ActiveWatch

A User's Guide

21 January 2022

Walnut Creek, CA

Staying On Top Of Unpredictable Text Data Streams
Through Language Expertise and Reliable Statistics

D r a f t

Table of Contents

1. What Is ActiveWatch?

ActiveWatch (AW) is an open-source software tool for exploring large dynamic collections of short English text segments like email messages, social network postings, or brief news items. It employs various kinds of statistical analysis to map out the major content areas of such collections and to identify segments that stand out in various ways. It can be useful in intelligence monitoring, legal discovery, or tracking of trends in web content.

AW is “active” in that it looks for information on its own; it is unlike a “passive” system, which requires a user to ask explicitly for everything. Querying is a critical capability for any information system, but when data is continually changing, we can miss important events by failing to ask the right question at the right time. AW digs into dynamic data on its own and can warn users when they need to pay attention to shifts in content.

AW is NOT a search engine or a search engine replacement. Nor can it read and understand natural language text like a personal assistant. It is a purely statistical system that counts selected features in text data. The trick here is knowing what to count and how to count it. This is a major challenge requiring familiarity with language plus information theory. AW draws upon reliable probabilistic modeling of text and extensive built-in familiarity with English.

AW was written entirely in the C language in the 1980’s and rewritten in Java at the turn of the 21st Century. This software demonstrates what could be done when indexing text data with a fixed, finite set of text features. The Java code was never released, but in 2021, it was recovered from a backup CD-ROM and cleaned up to compile with the latest versions of Java. The current AW system consists of 13 processing modules plus 3 support modules; together, these implement a distinctive way to cluster text items by content to delineate its major topics.

About another 8 legacy processing modules written in Java are in the pipeline to extend AW functionality. The old Java AW system also included a simple graphical user interface based on the Java AWT (Abstract Window Toolkit), a library of interactive display widgets. A new AW GUI may come at some point. The current AW package is mainly for expert demonstration, running from a command line under Unix (including macOS 10+). Linux, or Windows.

2. How ActiveWatch Works

Systems like ActiveWatch revolve around determining how much one text item is like another one. This typically involves a numerical function S(x,y) as a measure of similarity between two text items x and y, but such a value is often hard to interpret. That is especially so when a similarity measures is normalized to fall within a specific interval such as 0 and 1. This looks cleaner, but disregards the differences in length for two items being compared

Suppose that we have four items x, y, u, and v, where $S(x,y) \approx S(u,v)$. That suggests that the similarity between x and y is about the same as the similarity between u and v. If both x and y are long documents, while both u and v are short, however, then this theoretical near equality of similarity may not be evident to someone reading the text of u and v versus that for x and y. People do tend to take length into account in their own assessments of text similarity.

Inconsistencies in the interpretation of similarity measures can be tolerated in some areas of text analysis. For example, consider a ranked search based on S(x,q) for an item x versus a query q, treated as a short text item. A system needs only to determine whether $S(x,q) > S(y,q)$ or $S(x,q) \leq S(y,q)$ for all items x and y. This produces a ranked retrieval, and the top of the final ranking supposedly will be the best match, but “best” may not be “good.” A user always has to read the top item to see whether it is really worth reading.

When monitoring a live text input stream, however, there will never be any final ranking of items for a query; and if many standing queries are turned on, then someone has to keep watching the top matches of every query to see if it has found anything. What we really want here

instead is to set some fixed similarity threshold so that a system can notify users only when a query has made a significant match. This calls for a properly scaled similarity measure.

The thresholding problem arises in a different way in an application like automatic clustering of text items by content. With a normalized similarity measure between 0 and 1, we can try to set a minimum match threshold to some value like 0.7, but this will have a different interpretation depending on the actual pair of text items being compared. Clustering is still possible with such fuzzy similarity measures, but we cannot control the process as closely as we might want.

ActiveWatch addresses all such concerns with a new kind of text similarity measure based on statistical scaling. Actually, AW will define two slightly different measures for the two applications of standing queries and of automatic clustering. Statistically speaking, the first measure involves fewer degrees of freedom and can be computed more simply. Knowing which measure to use for which problem sets AW apart from other text processing systems.

3. Fixed, Finite Indexing With N-Grams

A similarity measure S(x,y) in general will compare features of item x with those of item y; having more features in common should result in a higher similarity score. For best results here, we have to choose the right features to work with. Ideally, they would be fairly frequent, though not ubiquitous, and be largely independent of each other. They should cover all of the text content of interest to us, but for statistical analyses, we need them to be finite in number.

In general, good features should all have about the same frequency in target text data. When a small subset of features are much more frequent than others, it means that we effectively have fewer features helpful in telling text items apart. To gauge the effectiveness of any index set, we can compute a standard information entropy score for the set, which will be maximum when all features do have the same frequency. This computation requires a finite set of features.

Although this all might seem like an impossible indexing wishlist, we can get a reasonable feature set by thinking outside the box. The default for statistical text processing system has been to take whole words as text features, but nothing mandates that this must always be so. In ActiveWatch, we will index with word fragments of some maximum length n. These are finite in number and can also be carefully selected for frequency, independence, and coverage.

Whole words has always been problematic for indexing. They do cover all content of interest if we include all words as features for indexing, but they will fall short in our other requirements. Words are not independent; they are often synonyms or antonyms to varying degrees. Their frequencies are also highly skewed, with most of them being quite rare. They are effectively infinite in number, because new words keep popping up in any live data source.

The Oxford English Dictionary currently lists about 170,000 words in the English language. Practical text processing, however, must also contend with inflected forms like DOGS and MATING, new morphological inventions like INCENTIVIZE or BALLER, names like JAMES T. KIRK or THERANOS, technical designations like U-235 or 401(K), nominalized numbers like 1960'S or 1-800-CARTALK, acronyms like TMI or POTUS, jargon or slang, and on and on.

Literate humans somehow manage such complexity, often in more than one language, but statistical text processing systems can be swamped by them. Although modern computers can easily keep track of billions of words or word-like entities, a statistical system must compute reliable probabilities. As a general rule, however, one has to see a feature many times for any good estimation of its likelihood of occurrence. With a live text data input stream, however, we would expect to see many words or word-like tokens for the first time on any given day.

Consequently, ActiveWatch instead works with a selected finite subset of short text fragments for indexing: all alphanumeric sequences of two characters, about 5,600 alphabetic sequences of three letters, over 2,000 selected sequences of four letters, and over 600 selected

sequences of five letters. Users may also specify up to 2,000 alphanumeric sequences of any length, either beginning or ending a word. Such a set of text features will be called **n-grams**.

A finite set of text features will of course be a disadvantage when looking for documents containing a particular word that is not a selected feature. That is what any decent search engine will will do for you. AW is instead intended for when you do not know what to ask for and need some prompting. AW can work on its own to explore unfamiliar data and give you insight into how it is changing. No queries are needed.

The analytic power of AW comes from its finite set of n-grams selected specifically for English text. With only about 10^4 fragmentary indices, both built-in and user-defined, AW can estimate their probabilities of occurrence quite well from only about 10^8 bytes of text data and in a pinch could get by with much less. This opens the way for more rigorous statistical modeling of English text data, allowing us to tackle practical problems of text analysis in novel ways.

In particular, when a text item is represented as a (finite) vector of the frequencies of feature occurrence, we can compute the expected value and variance of a theoretical multinomial distribution of the inner products of the vectors for random pairs of items. We can use this noise model to gauge rthe statistical significance of any particular inner product. AW scoring of similarity will be reported in units of standard deviations (abbreviated as σ).

An AW user need not understand all these underlying technical details. All one needs to know is that a scaled similarity score of 3σ or more is unlikely to be by chance. When actually running AW, a text processing application would usually set match thresholds of 6σ or even higher, which is highly unlikely by chance. In other so-called statistical systems, we never know how much a given similarity score is expected by chance.

4. Information Engineering With Finite Modeling

When building a structure like a bridge in the real world, an engineer must figure out whether its design will hold up under expected loads long before starting construction. This problem can be quite complex if one had to take into account for the performance of every bolt, cable, support beam, metal plate, and rivet. Instead, engineers turn to simplified models of a structure for which calculations can be more manageable.

In text processing, the representation of items as numerical vectors is such an engineering model. AW is particularly radical in its simplification, however. To begin with, it skips over grammatical function words like THE and AND, strips inflections form words, and ignores most markings of part of speech. This still leaves infinitely many possible indices to contend with, but AW then will count only n-grams, text fragments of at most n consecutive characters.

The set of all n-grams with a finite n will always be finite. What happens when a word or word-like token is longer than n characters? We can just allow the text token to be represented by more than one n-gram. For example, a long word might simplistically be broken into 5-grams:

```
SUPERCALIFRAGILISTIC
super
     calif
          ragil
               istic
```

So, we might count four 5-grams here instead of one long word. This is actually a poor solution, however, because it requires keeping track of about 12 million alphabetic 5-grams. This is excessive, given that there are fewer than a million words in English, but we can refine our indexing by working with a carefully designed mix of n-grams of different lengths.

The main point the example above is that, if a text item contained our full word, we still would have a good chance of finding the item by looking for all four of its 5-grams as analyzed here. Some noise is inevitable, but we can control it with the right indexing features. In general, we can approximate full-word indexing ever more closely by resorting to ever longer n-grams, though we want to be quite selective about them.

How long should our indexing fragments become so as to support a usable engineering model for text analysis? This has to be answered by running experiments, and the answer turns out to be surprising. For short items of only a few thousand characters, we can get effective indexing with a mix of selected alphabetic 3-grams plus all alphanumeric 2-grams. The 2- and 3-grams will be allowed to overlap in words to maximize the information in indexing.

Here is the AW 2- and 3-gram indexing for SUPERCALIFRAGILISTIC:

```
SUPERCALIFRAGILISTIC
sup
 upe
  per
   erc
    rca
     cal
      ali
       lif
        ifr
         fra
          rag
           agi
            gil
             ili
              lis
               ist
                sti
                 tic
```

The overlapping of features lets us count every occurrence of indexing 3-grams in a text segment regardless of its relative position in a word. All the 3-grams above come from the set of about 5,600 built into AW. Finding an indexing 3-gram at every position in a word is not unusual; it means that a word is recognizably English even though it is completely made up.

We can see, however, that indexing with common alphabetic 3-grams is more noisy than with 5-grams. People seeing a listing of indexing 3-grams in alphabetic order will have a hard time reconstructing the text they came from. This is good from the standpoint of independence, however, and having over 5,500 common alphabetic 3-grams plus all 1,296 alphanumeric 2-grams will allow us still to make good distinctions between different segments of English text.

We saw no 2-grams in our analysis above, but we shall need them to break down a word of like CHUTZPAH, which is of Yiddish origin and hardly English-like. Its AW analysis will be

```
CHUTZPAH
chu
 hut
  utz
    zp
     pah
```

We still manage to index with mostly common 3-grams for English, but TZP and ZPA are uncommon and missing from the AW built-in 3-gram indices. In their place, AW inserted the 2-gram index ZP to fill out its indexing, which calls for a bit of explanation.

5. The Non-Redundant Indexing Rule

ActiveWatch will recognize all occurrences of its indexing n-grams in text, but when a shorter one is inside a longer one, we have indexing redundancy. In that case, AW will ignore the shorter n-gram. For example, the word MINIMART begins with the 2-gram MI or the 3-gram MIN or the 4-gram MINI. These are all AW built-in indices, but only MINI will be counted in an analysis. In the case of CHUTZPAH, the 2-gram ZP overlaps with UTZ and also with PAH, but is not all inside either; so it does contribute its own bit of information to AW indexing.

With the full default complement of AW built-in n-gram indices, the non-redundant analysis of the word SUPERCALIFRAGILISTIC becomes:

```
SUPERCALIFRAGILISTIC
super
  perc
    rca
     cali
       lif
        ifr
         frag
           agi
            gil
             ili
              list
                stic
```

You can see our overall indexing strategy at work here. First, by adding frequent longer n-grams in English, we can improve the specificity of our indexing by text fragments and make it more closely approximate indexing by whole words. Second, when longer n-grams subsume shorter ones in text, this will reduce the counts of shorter ones and help to increase the overall entropy of indexing, since shorter n-grams are usually more frequent than longer ones.

We can continue this way to improve its performance incrementally. More alphabetic 4-grams seem to be our best bet here. There are 456,976 of them to choose from, and they are still short enough so that few will be recognizable as words. This avoids most of the problems of indexing by words, although it will take time to find the best 4-grams to work with. Too many indices make it harder to estimate all their probabilities of occurrence, since most will be rare.

Currently, AW built-in indices include only up to 5-grams. These are starting to look like words even when incomplete. For example, the built-in 5-gram REPOR is almost certainly from REPORT. Longer n-grams will be even more meaningful in this way, making it harder to assume independence for n-gram indices. AW can tolerate a few words as indices, but we want the bulk of our indexing to be with true word fragments for which we can assume independence.

6. Built-In N-Grams for Indexing

When a text item is only about 10,000 chars long, representing it with a numerical vector of dimensionality in the millions is grossly inefficient, especially if much of each vector is counting nonsense features. From an engineering standpoint, a finite system design should be lean and mean. The solution here is to note that only a subset of all n-grams is needed in an AW built-in index set. Choosing them requires more than a little effort, but pays off when done properly.

The current AW built-in index set has 1,296 alphanumeric 2-grams, 5,616 alphabetic 3-grams, 2,460 alphabetic 4-grams, and 700 alphabetic 5-grams. The 2- and 3-grams have been in AW since its earliest days and were the basis of the first experiments to show that fixed, finite indexing conveyed enough information to support useful text analysis. Alphabetic 4- and 5- are more recent. For a more detailed discussion of how everything came to be, see Appendix D.

The central problem of AW text processing is in capturing the content of arbitrary English text with only a limited set of n-grams indices. Compromises are unavoidable, but it has been possible to come up with a reasonably workable solution through a combination of intuition, serendipity, and trial and error. This can undoubtedly be improved upon, but it is sufficient to build an entire system around.

7. User-Defined Literal N-Grams

It is possible to run ActiveWatch on any English text data with just its built-in index set of all alphanumeric 2-grams and selected alphabetic 3-, 4-, and 5-grams. In competitive situations, however, users can sharpen their indexing of specific target content by defining up to 2,000 of their own n-grams, called literals. These can be employed to approximate full-word indexing more closely, but literal indices will always be treated as n-grams — word fragments.

Literal n-grams will be fully integrated with built-in indices, but will follow somewhat different rules in actual AW indexing.

- Literals may be alphanumeric character sequences of at least length 3 with no upper limit; a 2-gram may never be a literal.
- They must either start or end a word or word-like token, but not both. The former will be called leading literals and are represented with a final hyphen (e.g. PROTO-); the latter, trailing literals, represented with an initial hyphen (e.g. -VISE).
- A literal may cover an entire word. This is how users might achieve something like indexing with whole words.
- If a leading literal and a trailing literal both cover a full word, then only the leading literal is counted.
- If a trailing literal covers a full word, and leading literal covers only part of a word, the only the trailing literal is counted.
- A leading or trailing literal CANNOT be subsumed by a longer built-in n-gram. It will always be counted, subject to non-redundancy. If the literal is the same length as a built-in n-gram, then only the literal will be counted
- The non-redundancy rule still applies otherwise. Even if a leading and trailing literal overlap, AW may find one or more built-in n-grams that subsumes the overlap, but is not entirely contained within either literal.

AW users will define literal n-grams by listing their character sequences, one per line, in a text file called `literal`. A sequence ending with a hyphen (–) will be a leading literal; one starting with a hyphen will be a trailing literal. If there is no hyphen, the literal is assumed to be leading. For example, here are some default literals in the AW distribution package:

```
astro-
astron-
infra-
-onomy
-structure
```

Some analyses with these literals:

```
ASTRONOMY
astron-
   -onomy

ASTROEONOMY
astro-
   roe
     eon
     -onomy

INFRASTRUCTURE
infra-
   rast
    -structure
```

ASTROEONOMY is a fake word, but the example above shows how AW would analyze it with its rule for non-redundancy of indexing. Note that if ONOMY had also been defined as an AW built-in 5-gram index, it would not be counted at the end of the word ASTRONOMY. The literal n-gram takes priority.

8. How to Count

A full AW index set will include N = 10,000 to 12,000 built-in and user-defined n-grams. We can then represent every text item as a N-dimensional vector of the respective frequencies of indices in the text of item. Except for finiteness, an AW vector looks just like a vector from whole-word indexing. Of course, no numerical vector will ever capture everything in a text item, but anything done with a full-word vector can also be done with an n-gram vector.

AW will count feature occurrences differently, however. That is because of Zipf's Law: in any natural language, only a small subset of features for indexing will usually account for most occurrences of those features. This is a variant of the well-known ten-percent rule, where ten percent of the people in an organization usually do ninety percent of all the work. When most features make only a small indexing contribution, their overall usefulness in distinguishing between types of content will suffer.

Imagine being in a kitchen with a pantry full of many ingredients, but mostly potatoes, plus turnips and parsnips. A good chef can perform wonders here, but how many different ways can one cook root vegetables? Before long, every meal will start looking and tasting the same despite a thousand small jars of various herbs and spices in the pantry. A similar situation arises with text features and numerical vector representations. Zipf's Law is unyielding.

Nevertheless, we can get off fairly well with a better assortment of starting ingredients and some better recipes. For the latter, AW starts off by counting overlapping n-grams in a word with its rule of non-redundant indexing. Furthermore, AW counting is not a simple 1, 2, 3, … because we want to compensate for Zipf's Law. The idea is to adjust high counts in vectors so that indexing contributions of their non-zero n-gram indices can be more noticeable.

Such transformation is a common practice among statisticians working with text data. Instead of a raw count, they often use the square-root or the logarithm of the count, which will lessen the disparities of Zipf's Law. AW employs a logarithmic transflrmation here, but to avoid floating-point numbers, it will count 1, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4.… This flattening of counts will increase the entropy of any kind of indexing.

9. Getting ActiveWatch Source Code and Jarred Modules

Finite N-Gram indexing is more than an academic exercise. You can actually see it work on your own computer with your own text by downloading the new Java implementation of ActiveWatch from GitHub, including its source code. This is free software. You could write your own code for n-gram indexing, but we have had many decades to think about how to do this. Rather than reinvent the wheel, you can just rejigger ActiveWatch to do exactly what you want.

The easiest way to get a copy of AW on your own computer is to clone the ActiveWatch repository on GitHub, which is a place on the Worldwide Web where developers can share open-source code. The URL for ActiveWatch is

`https://github.com/prohippo/ActiveWatch`

If you direct your browser to this URL, you will see the top page for the GitHub repository **`prohippo/ActiveWatch`**. There should be a prominent green button labeled "Code ⌄". Clicking this will show several options. The first is "Clone", which will re-create the entire AW repository on your computer, including all revision histories. If this is too much, you can choose "Download ZIP", which will provide just the latest AW files in their hierarchy of file directories.

Given that AW in Java is still evolving, you should probably clone if you want to work with the latest version. Cloning does not automatically give you privileges to put your own changes into the AW repository.

10. ActiveWatch Basic Elements

We now can talk about how to use ActiveWatch to explore unfamiliar text data. AW will operate on files of persistent data objects to create files of other such data objects. The files will persist over AW update cycles as new text data arrives in batches; the main AW data objects include

text Item	a subsegment of input text assigned an ID consisting of a batch number and a sequential accession number in the form b:n; e.g. 2:101.
index vector	as already described above. These vectors will represent subsegments of text items. AW will split long text item into multiple subsegments having sequential IDs of the form b::m; e.g. 2::123.
profile	essentially a vector, but with floating point weights for n-gram indices instead of integer counts. Profiles will have only a few indices with non-zero weights. They function like queries to be matched against vectors.

match list showing subsegments similar to a profile above a minimum threshold, sometimes sorted in descending order of match score.

sequence an ordered listing of subsegment IDs, grouped into runs of consecutive IDs in the same batch.

attribute data various information associated with saved profiles. This will include descriptive keywords.

AW will save persistent objects in files containing mostly one type of object. Their names will indicating the type of object and the update batch in which they were generated. For example, vectors will be saved in the files `vector00`, `vector01`, `vector02`, ... , `vector30`, and `vector31`. AW currently will keep only up to 32 batches; if more data arrives for processing, then the older batch is deleted, and its number is recycled for the newest batch.

AW also generates various statistics also stored in files. The most important will be current probabilities of occurrence for n-gram indices, kept in a binary file called `probs`. That file will change with each update cycle, but will persist indefinitely. Noise model parameters for statistical scaling of similarity will also be saved in files, but usually in temporary ones written by one AW module and read by another.

All AW files have to be in a single working directory, although input text files can be anywhere. Individual AW modules may be outside the working directory. Each module should be built as a Java archive, a jar file with a six-letter uppercase name like SEGMTR. To run this module, a user would enter the command

```
java -jar path/SEGMTR
```

where `path` indicates where the jar file `SEGMTR` can be found. AW jar files should run on any computer platform that has installed the latest version of Java and supports a command line.

11. ActiveWatch Architecture

ActiveWatch began as a series of separate modules written in the C language. This was to make it easier to experiment with different algorithms and data formats. It also allowed different functionality to be supported by a reconfiguration of modules. When AW was rewritten in Java around 1999 for greater portability, it kept the various modules of the earlier C version and added many new ones.

In 2021, a subset of the 1999 system was converted to compile and run with the latest version of Java. The modules currently fall into four groups: setup, core processing, application processing, and reporting.

setup STPBLD, SUFBLD, LITBLD

These build the stopword, suffix, and literal n-gram tables for customizing AW text processing.

Stopwords are text words excluded from AW n-gram indexing, usual grammatical function words like THE , OF, and AND. The STPBLD module reads in a text file `stopword`, which contais a list of stopwords, one per line, and writes a binary file `stps` for AW modules to read. You may edit the input however you want, but it is probably a bad idea not to stop words like THE , OF, and AND.

The SUFBLD module reads in text files `suffix` and `action` and writes a binary file `sufs` for AW modules to read. This controls AW morphological stemming, which will remove endings like -MENT from words like ESTABLISHMENT from n-gram indexing. but not from words like

CEMENT. The suffix file contains stemming rules, one per line, and if a rule is matched, AW executes one of a list of possible actions defined in a text file `action`. The AW morphological stemming algorithm is quite complex and probably will left alone except by the most exacting of AW users. Users can refer to Appendix A of this guide to find out how to customize morphological stemming.

User-defined literal n-grams have already been described above. The LITBLD module reads in a text file `literal` and writes a binary file `lits` for AW modules to read. This is probably where most users will want to customize. AW built-in indices were chosen according to statistics for English in general. To sharpen indexing for specific kinds of English text, you may add up to 2,000 of your own alphanumeric n-gram indices that either start or end words. Just edit the `literal` file in your working directory and run LITBLD there.

core SEGMTR, INDEXR, UPDATR

These implement the basic processing required by almost all AW applications. The three modules should always be run with SEGMTR first, followed by INDEXR and then UPDATR.

SEGMTR reads in multiple text input files and divides them up into shorter subsegments for n-gram indexing. The major segmentation is controlled by a finite-state automaton defined by the text file `delims`. Appendix B will give th details on how to specify the logic for this automaton. If the text of items found by the automaton is long, then SEGMTR will divide the text into multiple shorter subsegments.

INDEXR computes an n-gram index vector for each subsegment found by SEGMTR, using AW built-in 2-, 3-, 4-, and 5-grams plus user-defined alphanumeric leading and trailing n-grams of arbitrary n. The indexing procedure was discussed above.

UPDATR updates n-gram probabilities for a new batch of text subsegments and starts an empty next batch. It also computes indexing entropy to gauge how well a set of n-gram indices can discriminate between the content of different subsegments. With the current AW demonatration system, entropy is about 11.6 bits, but this can be improved with literal n-grams defined for a user's target text.

application SEQNCR, SQUEZR, MLTPLR, CLUSTER, SUMRZR, CLSFYR

These are the modules required for automatic clustering of text subsegments by content. They have to run in the order shown above.

SEQNCR creates a sequence file that lists all subsegments in a current batch being processed plus those of subsegments unassigned to any cluster in previous batch. In this way, we can discover clusters of subsegments spread over multiple batches.

SQUEZR creates a compact file of index vectors for all subsegments in a sequence. It will keep only the first subsegment from a long item, since subsegments from the same item will strongly resemble each other and produce new clusters of no real interest. Indexing n-grams with fewer than 10 occurrences will also be ignored. A compact vector file will speed up the computation of similarity measures for every pair of index vectors for a sequence of text items. SQUEZR will also precompute

intermediate values (mean, variance, and covariance) for n-gram probabilities, which will speed up computation of noise distribution parameters for scaling raw inner product similarity scores.

MLTPLR computes simlarity measures between every pair of vectors in a squeezed index vector file. It will write out a file of links for pairs of vectors with a scaled similarity score above a given minimum significance. The AW demonstration system operates with a minimum link significance of 8σ or more.

CLUSTR applies a minimal spanning tree algorithm to a file of links to get a set of highly interlinked cluster seeds. The link density of a seed is is the minimum number of cluster links for an item in a seed divided by one less than the number of subsegments in it. If this density falls below a given threshold (default = 0.5), then the weakest links in that seed are dropped; and the seed then is broken up and reclustered. When all cluster seeds are small enough and lhave high enough link density, they are written out in a binary file.

SUMRZR reads a file of cluster seeds and produces a profile describing each one. It also precomputes noise model parameters for statistical scaling of similarity scores for each profile. This differs from the noise model for MLTPLR, which assumes that both sides of an inner product for vectors can vary. For profile matching, the profile weights are constant; only the vectors vary. It sets a significance threshold for a vector match with the profile; the default is 6σ.

CLSFYR compares all the subsegment index vectors identified in a sequence file with all profiles newly generated from cluster seeds and assigns subsegments to a cluster if its vector matches the profile for the cluster. Even items in the seed for a cluster must be assigned in this way. Results are saved in a match list file for cluster. Item not matching any profile will be put on a residual list, which will structured as sequence file.

user application PROFLR, EXMPLR

These set up user-defined standing profiles when specific target content is known beforehand.

PROFLR defines multiple profiles from user-supplied keywords in an input file. These are saved in the same places as profiles created by SUMRZR.

EXMPLR is like PROFLR, but defines them from user-supplied examples of items, specified as subsegment ID's in an input file. This can provide helpful indexing information even when you do not plan to run resulting profiles with CLSFYR.

reporting KEYWDR, WATCHR

These two modules run independently to show different aspects of automatic clustering of text items by content.

KEYWDR looks at all the words occurring in a cluster and assigns them a importance score according to the weighting of its n-gram indices in the profile generated for the cluster. The words with a non-zero importance score are then shown in descending order of that score.

WATCHR looks at the subsegments in a specified sequence file (default is the latest list of residuals from clustering). Each subsegment in the sequence is given a generality score computed as the sum of probabilities for the non-zero n-gram indices in its index vector. AW then reports the subsegments with the lowest generality scores when its total number of counted n-gram index occurrences in their vectors above a given threshold (default = 50).

Other modules can be expected in the **application** and **reporting** categories for upcoming AW releases.

12. ActiveWatch File Environment

For the command version of ActiveWatch on a computer running Unix, Linux, or Windows, a user should first set up an empty working directory for all AW analysis. It will fill up fast, and any other files kept there will probably become hard to find, especially after multiple batches of data have been processed. Here are the files produced when running the AW demonstration system with a single batch of Google News text

```
action    attributes  clusters  control   counts    delims
heads     index00     links     lists     literal   lits
maps      offset00    probs     profiles  range     residual
segmn00   sequence    source00  stopword  stpats    stps
suffix    sufs        svectors  vector00
```

The `control` file is created by AW to keep track of the batches currently accumulated in a text data collection.

STPBLD reads the `stopword` text file to write the binary file `stps` to be loaded by AW processing and reporting modules. SUFBLD reads the `suffix` and `action` text files to write the binary file `sufs` to control AW morphological stemming. LITBLD reads the `literal` text file to write the binary file `lits` to define literals, leading and trailing n-grams. A file `stpats` defines patterns for recognizing other word-like tokens to stop, but are not explicitly listed out in `stopword`; `stpats` will be read directly by AW modules and does not have an AW module to convert it into binary.

Copy the `stopword`, `suffix`, `action`, `literal` and `stpats` text files from a AW GitHub download into your working directory so that they can be edited and customized for a particular text data collection.

The binary files `source00`, `index00`, `segmn00`, `vector00`, and `offset00` are associated with batch 0 of text input. which may come from multiple input text files.

SEGMTR reads each file, records it in `source`*, divides it into items through a finite-state automaton defined by the text file `delims` (by default) and records each item in `index`*, and finally divides each item into one or more subsegments recorded in `segmn`*.

INDEXR reads the output of SEGMTR and produces an index vector for each subsegment recorded in `segmn`*. Index vectors will be appended to the file `vector`*, with the start of each vector appended in the file `offset`*.

UPDATR reads in all the vectors of the current batch and updates n-gram probabilities stored in `probs`, n-gram subsegment counts stored in `counts`, and lower and upper limits on non-zero probabilities stored in `range`. It also automatically sets the number of the next batch for processing any subsequent input data.

SEQNCR generate by default a sequence including all index vectors of the current batch. This is saved in the file `sequence`.

SQUEZR reads in all subsegments in a sequence and keeps only the first one for each text item in a batch. These are stored in a file `svectors`, which has a header with precomputed multinomial model parameters for statistically scaling inner product similarity measures.

MLTPLR computes a scaled similarity measure between every pair of vectors in `svectors` and writes out to the file `links` a record for each pair having similarity equal or greater to a specified significance threshold.

CLUSTR reads in `links` and writes out a file `clusters` of the cluster seeds found.

SUMRZR reads in `clusters` and produces a profile for each cluster seed. The profiles are saved in the `profiles` file and the creation of each is recorded in the binary file `maps`. A separate record describing each profile is saved in the file `attributes`.

CLSFYR compares the vector of every subsegment listed in `sequence` against newly created profile in `profiles`. Every match for a profile is saved in a match list stored in the `lists` file and indexed by the `heads` file. The module will also create a new `residual` sequence file identifying the the text subsegments that failed to match any profile.

KEYWDR reads in match list for each cluster profile and identifies words containing highly weighted n-grams in the profile. The words with the highest sum of weights for their n-grams will be displayed and also saved in the `attributes` file record for the profile.

WATCHR computes the sum of probabilities for n-grams appearing in the vectors of subsegments listed in the `residuals` sequence file. It will show the subsegments with lowest sum of probabilities when the total count of n-gram occurrences is greater than or equal to a specified threshold. Nothing is written to any AW file.

The AW clustering demonstration system more or less will run all the modules above in the order given. Actual results for each module, however, will vary according to the arguments passed to the module in its command-line invocation.

PROFLR and EXMPLR work with `maps`, `attributes`, and `profiles`, as well as their given definition text files. These are optionally run after UPDATR, but are not required for the basic AW dmonstration.

13. Controlling AW Modules

ActiveWatch is open-source code with a BSD license. You can freely modify anything in its processing. The object-oriented Java code may be confusing because of its outdated style and multi-level encapsulation of data structures, but anyone familiar with Java can reconfigure AW operation. You can even add alphabetic 4- and 5-grams to its built-in index set by editing the source file `gram/GramMap.java` plus also `gram/Gram.java` if the numbering base for 5-grams has to raised to make room for new 4-grams.

Most of the time, users should just run the currently available jarred AW modules. When these are invoked from a command line, you can control them by passing optional arguments to them. The most important current options are as follows:

Module	Arguments		Comments
STPBLD	(none)		
SUFBLD	(none)		
LITBLD	(none)		
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	–n N	set maximum built–in n–gram length	argument for sepecial testing only
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 sequence to cluster; default is ALL	used for debugging
SQUEZR	MinM MaxM MinL	minimum prob as multiple of lowest, default=10 maximum prob as mutiple of highest, default=0.5 minimum vector lenght, default=25	these determine which n-grams will have a role in vector similarity
MLTPLR	Minimum	threshold for pairwise link; default=6σ	
CLUSTR	Dthr Nitr Maxsz	threshold for link density of seed; default=0.5D number of full iterations maximum cluster seed size	link density is 1.0 when every item in a cluster seed is linked to every other one
SUMRZR	MinThr Multp MinLen	threshold for cluster assignment; default is 6σ minimum prob as multiple of lowest, default=10 minimum n-gram count for creating profile, default=16	null profile might be created
CLSFYR	Type	which profiles to apply, default=‘a’ for ALL; otherwise, ’n’ for new profiles, ‘u’ for user-defined priofiles only	
KEYWDR	Maximum	number of items to get keywords from; default=6	
WATCHR	MaxLis MinLen FileN MaxSum	number of standout items to display; default is 24 minimum vector length for items to show; default=50 name of sequence file to work from maximum probability sum to accept	
PROFLR	FileN	name of definition file; default is topics	See Appendix C.
EXMPLR	FileN	name of definition file; default is examples	See Appendix C.

AW modules coordinate their processing through the `control` file, which keeps track of the number of data batches known to the system. All new AW text data must come in as a text file read by SEGMTR. If no `control` file exists in a working directory, SEGMTR will create one. All input files will be assigned to the designated current batch.

When enough text data has been accumulated, an AW user can run INDEXR to create an n-gram index vector for each subsegment in the designated current batch. This must immediately be followed by UPDATR, which will update AW n-gram probabilities with the new data in the current batch and will start the next batch.

Once a batch of input text has been indexed, you can run AW automatic clustering by content. This can be done by invoking the modules SEQNCR, SQUEZR, MLTPLR, CLUSTR, SUMRZR, CLSFYR, and KEYWDR in that order on separate command lines. It is all right to specify no arguments; default values will suffice to demonstrate AW clustering.

If you want to experiment with clustering, you can delete the `control` file and redo the entire series of AW modules starting from SEGMTR. This time, try increasing the `minimum` argument for MLTPLR to 7 or higher or the `MinThr` argument for SUMRZR to 7 or higher. These will make clustering more restrictive, finding fewer clusters and assigning fewer items to them.

To save time in retries, delete just the `maps` file and start run from MLTPLR in the series of AW modules for clustering. With the Google News text data provided with the AW GitHub distribution, the entire demonstration should take less than ten seconds to produce results. With your own data, processing time will be proportional to the square of the number of items.

14. An Example of AW Clustering

Here are the top 6 clusters produced from the Google News demonstration data with all default arguments except for telling SEGMTR what text file to process and setting the minimum link significance in MLTPLR to 8 standard deviations. The output below was produced by the KEYWDR module in AW version v1.1.4. It has been truncated for brevity; usually, there will be about 80 different clusters.

The clusters will vary with the actual AW index set employed, but the top clusters will be fairly stable, with only slight variation in their listed order. The choice of an 8 standard deviation significance for cluster seed links and 6 standard deviation matching of cluster assignment profiles give us confidence that the results are not purely random.

The descriptive keywords for each cluster may not all be whole words. Some are more like word roots, although there is no standard for this. Such roots are an artifact of the morphological stemming algorithm employed by AW, which tries to eliminate n-grams serving mainly to mark the part of speech of a word. These have less value when indexing text for content. It should not be hard to relate word roots to actual words, however.

```
Active Watch release v1.1.4 (2021) Java - Keyworder
Copyright 1997-2002, 2021, CPM

deriving descriptive keys
------- cluster 1         (@ 1)
get keys from top 6 items out of 14
0::159 0::84 0::95 0::45 0::49 0::78
keys selected from 180 candidates

zimbabwe          mugabe            robert            preside
harare            succeed           independ          capital
lead              mnangagwa         emmerson          rule
army              took              africa            head
thousand          milite            address           know
struggle          decade            command           fall

  -other keys-

------- cluster 2         (@ 2)
get keys from top 3 items out of 16
0::320 0::553 0::485
keys selected from 112 candidates

philadelphia      massachusett      boston            north
york              snow              new               heavy
virginia          atlantic          snowfall          northeast
fall              east              nor'east          maine
part              baltimore         area              city
centr             issua             england           delaware
mid               metro             expect            coast

  -other keys-

------- cluster 3         (@ 3)
get keys from top 6 items out of 10
0::610 0::582 0::563 0::544 0::529 0::517
keys selected from 179 candidates

cambridge         facebook          analyt            firm
data              zuckerberg        mark              profile
employ            committee         politic           scrutiny
usa               obtain            brian             share
giant             blow              work              act
investige         year              world             suspend

  -other keys-
```

```
------- cluster 4         (@ 4)
get keys from top 6 items out of 15
0::540 0::502 0::552 0::562 0::349 0::334
keys selected from 130 candidates

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

  -other keys-

------- cluster 5         (@ 5)
get keys from top 6 items out of 13
0::322 0::300 0::287 0::337 0::276 0::263
keys selected from 175 candidates

ventura           santa             fire              evacuate
california        acre              county            thoma
barbara           scorch            flame             burn
carpinteria       communite         angele            new
order             destruct          blaze             reside
wildfire          spokesman         diego             south
highway           year              time              people
coast             move

  -other keys-

------- cluster 6         (@ 6)
get keys from top 6 items out of 9
0::127 0::86 0::113 0::160 0::97 0::143
keys selected from 126 candidates

submarine         argentine         crew              juan
navy              san               report            south
vessel            miss              coast             membr
ara               atlantic          spokesman         plata
communice         search            know              balbi
base              disappear         country           area
explode           found             locate            mile

  -other keys-
```

The Google News text data is from late 2017 to early 2018. The top 6 clusters here are about (1) a coup in Zimbabwe, (2) a nor'easter winter storm hitting the Northeast of the U.S., (3) the role of a British company influencing the 2016 U.S. presidential election through Facebook user data obtained illicitly, (4) a bomb exploding in Austin, TX, (5) a huge wildfire in Southern California, and (6) the disappearance of an Argentine submarine in the South Atlantic.

The clusters found by AW do seem to identify major topics in the Google News data. You can check on this by examining all seventy or so clusters in the demonstration output using diagnostic tools described in Section 16. The number of clusters here will depend on the actual n-gram index set employed. Lately, the list of alphabetic 4- and 5-gram indices have been growing in an experiment to understand how we can reduce indexing noise.

Intuitively, we would think that 4-gram indexing should be less noisy than 3-gram indexing. It may therefore be surprising how much AW clusters will remain more or less the same even after adding thousands of 4-grams to a mixed index set of about seven thousand alphabetic 3-grams and alphanumeric 2-grams. Do not judge AW just of the demonstration, however; see for yourself what AW will do on a collection of your own text data.

15. An Example of AW Residuals

AW statistically scaled n-gram similarity measures allow the setting of interpretable thresholds. The default significance for forming cluster or for matching cluster pofile is 6 standard deviations (σ), which should produce results highly unlikely by pure chance. Users may, however, crank up thresholds to get only the most significant clusters in a batch of data or crank them down to get more more items assigned to some clusters.

For example, a low threshold of 2σ will be quite noisy, but a user may sometimes want to be more inclusive in results. With any positive threshold, however, some items will remain unassigned to any cluster. Some similarity scores have to be below the mean score expected by chance because the mean of any distribution will never be the same as its minimum.

The latest release of AW has a WATCHR module that looks at items that failed to be assigned to any cluster and so would missing from any cluster map for a text data batch. These are called residual items; and they can account for as much as a quarter of a batch. WATCHR tries to find items that stand out in the sum of the probabilities for its n-gram indices. This may or may not have enduring importance, but it offers yet another window on dynamic data.

Here is the output of WATCHR for the Google News sample data after cluster assignments.

```
Active Watch release v1.1.4 (2021) Java - Watcher
Copyright 1997-2002, 2021, CPM

scanning segments in residual file
24 segments selected from 138 candidates
  1)  0::399   0.02738252 (  88)
  2)  0::171   0.03223459 (  96)
  3)  0::104   0.03600021 ( 131)
  4)  0::427   0.03864775 (  86)
  5)  0::155   0.03991351 ( 110)
  6)  0::24    0.04122145 (  88)
  7)  0::212   0.04124255 (  89)
  8)  0::69    0.04163282 ( 142)
  9)  0::351   0.04242392 ( 122)
 10)  0::183   0.04252940 ( 107)
 11)  0::60    0.04271926 ( 154)
 12)  0::12    0.04324666 ( 103)
 13)  0::231   0.04373187 ( 126)
 14)  0::570   0.04390064 ( 127)
 15)  0::283   0.04442804 ( 116)
 16)  0::339   0.04506091 ( 147)
 17)  0::343   0.04536681 ( 120)
```

```
18)  0::453   0.04548283 ( 122)
19)  0::70    0.04551448 ( 143)
20)  0::324   0.04652708 ( 137)
21)  0::573   0.04708612 ( 130)
22)  0::387   0.04756078 ( 107)
23)  0::269   0.04761352 ( 109)
24)  0::533   0.04770845 ( 154)
```

The second column is a subsegment ID, the third is the sum of probabilities of the n-gram indices in a subsegment, and the fourth is the total sum of counts in the index vector for a sunsegment. To see the text of a subsegment, you can run the DTXT tool described in the next section; just pass the subsegment ID as an argument.

16. Diagnostic Tools

AW diagnostic tools were originally developed to debug the processing of AW modules. This is still an important purpose, but the curious user can also run them to get a peek under the hood of AW. In the absence of a graphical user interface, the tools also provide the most convenient way to view text items.

Seventeen tools are currently included in the latest AW distribution from GitHub. These have been compiled and put into executable jar files with 3- and 4-letter file names. Some of them take arguments as indicated.

CTRL	shows the contents of the binary AW control file
DCLU	shows the cluster seed file produced by the CLUSTR module
DIXS	shows the index records identiying individual text items ih a batch of text data processed by the SEGMTR module
DLNK	shows the links computed by the MLTPLR module
DLSS	shows the match lists produced by CLSFYR
DLST n	shows the match list for profile n in descending order of match significance with major gaps indicated
DMAP	shows the allocation of profiles defined by SUMRZR and eventually also by users
DPRB m	selects to the m n-gram indices with the highest probability in a set of text data batches; if m is omitted, 16 is assumed
DPRO n	show the n-gram index weights for profile n
DNGM	shows the relative importance for indexing of user-defined literal n-grams, alphanumeric 2-grams, and alphanumeric 3-, 4-, and 5-grams.
DSCN ng	finds text segments containing an n-gram index expressed as an integer code ng
DSEQ	shows an AW sequence produced by SEQNCR
DSQV	shows squeezed vectors produced by SQUEZR
DTXT id	shows the text of an item or a subsegment given its item or subsegment id (i.e. b:n or b::n)

DVEC id	shows the index vector of a specified subsegment or the first subsegment of an item
GMx tok	shows the n-gram analysis of a given token string with no stemming
TGx < in	shows the analyses of text from standard input with stemming

If the diagnostic tools are stored in a file directory `path`, then a tool XXXX would be invoked in a working directory with the command line

```
java -jar path/XXXX
```

More tools will migrate over from the old AW system as time permits.

17. Using Diagnostic Tools to Explore Clustering Results

Diagnostic tools separately implement what an AW graphical user interface might bring together, although perhaps with more detail than most users would ask for. These tools will show how well AW is working.

Here is sample output produced by DNGM in AW v1.2 with the Google News data. It shows how much the various types of n-grams are contributing to the work of indexing content.

```
% java -jar path/DNGM
analysis of n-gram contributions

Literal N-Grams
total prob=0.106284 for 342 indices
min=0.000011, max=0.004539
--
Alphanumeric 2-grams
total prob=0.056611 for 522 indices
min=0.000011, max=0.002204
--
Alphabetic 3-grams
total prob=0.395429 for 2777 indices
min=0.000011, max=0.003720
--
Alphabetic 4-grams
total prob=0.348113 for 1809 indices
min=0.000011, max=0.004222
--
alphabetic 5-grams
total prob=0.093563 for 449 indices
min=0.000011, max=0.003338
—
%
```

This indicates that alphabetic 3-grams are pulling the most weight in indexing according to the sum of their probabilities. They are closely followed by 4-grams, then less closely by literals, 5-grams, and 2-grams, in that order. For noise control, we probably want more built-in 4-grams; it takes about another hundred new 4-grams to raise indexing entropy by about .01 bits. Note that, out of over 2,000 built-in 4-grams defined for AW v1.2, only 1,809 actually occurred in the Google News demonstration data for AW v1.2. Later versions will be higher.

We can look at n-gram probabilities in another way by running the DPRB tool to get the top indices for the Google News data (the default is 24):

```
% java -jar path/DPRB
analysis of probabilities

probability range= 0.000011 : 0.004467

5604 non-zero indices with sum of probabilities=1.000000
based on 93127 total occurrences of indices
computed entropy = 11.5 bits, 92.3 percent of maximum
(saved entropy= 92.3)

24 indices with highest percentages of occurrence

 preside-         (  337):  percent=0.4467
 trum             (10652):  percent=0.4156
 rump             (10383):  percent=0.4134
 new              ( 5117):  percent=0.3662
 office-          (  313):  percent=0.3361
 state            (11409):  percent=0.3286
 year             (10771):  percent=0.3146
 pro-             (  339):  percent=0.2856
 report-          (  364):  percent=0.2652
 time             (10604):  percent=0.2545
 people-          (  322):  percent=0.2191
 20               ( 8735):  percent=0.2169
 lead             ( 9835):  percent=0.2115
 us               ( 8439):  percent=0.2040
 ore              ( 5671):  percent=0.1976
 act              ( 2046):  percent=0.1825
 -son             (   45):  percent=0.1815
 day              ( 2909):  percent=0.1815
 north-           (  308):  percent=0.1772
 house            (11196):  percent=0.1750
 -russia          (    1):  percent=0.1675
 01               ( 8664):  percent=0.1664
 man              ( 4822):  percent=0.1632
 -nation          (   42):  percent=0.1600
%
```

The numbers in parentheses are the AW codes for each n-gram; the percent values are just 100 times the probability of an index n-gram. Note that many n-grams are longer than 3 in the top 24; it indicates a good balance of indices for the reduction of noise. The 3-grams listed here are mostly common 3-letter words. A closer examination of a longer listing of top n-grams suggests that indexing will be more finely tuned if we added the literals TRUMP-, KOREA-, and -BAMA (for ALABAMA) to our AW index set.

To see the text of any item, we can run the DTXT tool, which can be used for both text items and individual subsegments. For example, to show item 0:111,

```
% java -jar path/DTXT 0:111
dumping text
** item 0:111 full length=1238 chars, text length=1238 chars versus 1238 bytes stored
--HEAD



--BODY


A report from Politico this week, which found that the special counsel Robert Mueller is
gearing up to interview the White House communications director, Hope Hicks, indicates
that a significant part of the Russia investigation is probably moving into its final
stages.
Hicks has long been one of President Donald Trump's most trusted advisers, and she was
present during some events that are key to the special counsel's investigation.
Mueller's investigation includes multiple components. In addition to looking into whether
members of the Trump campaign colluded with Moscow to tilt the 2016 election in Trump's
favor, the special counsel is also investigating Trump on suspicion of obstruction of
justice related to his decision to fire James Comey as FBI director.
As part of that investigation, ABC News reported on Sunday, Mueller has asked the
Department of Justice for all emails connected to Comey's firing.
Mueller has also requested documents related to Attorney General Jeff Sessions' recusal
from the Russia investigation. Sessions announced his recusal in March after it emerged
that he had failed to disclose contacts with Sergey Kislyak, then Russia's ambassador to
the US, in his Senate confirmation hearing in January.
--
DONE
%
```

Note that the command line above has the argument 0:111. This will cause the output to show the full item with its various parts marked. A subsegment index like 0::22 as an argument will show only the text for that subsegment.

With DTXT, we can get the first sentences of the top 4 residuals from the WATCHR output of the previous section.

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::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::427 In December, a team of U.S. government scientists released a “report card” on the Arctic.

The residuals tend to be stable with the addition of alphabetic 4-grams to the AW built-in index set, athough their ordering in WATCHR output may vary a little.

The diagnostic tools DPRO and DLST let a user look at the profile for a given cluster or the match list for that profile in descending order of significance of match. A profile is really an n-gram index vector with fewer indices having non-zero weights; here is the output for profile 66.

```
% java -jar path/DPRO 66
profile 66
lexical  threshold= 2
phonetic threshold= 0
significance threshold= 6.0
expected value= 0.654833
variance      = 29.27065

no filters

weights
  0: america-    25 (   89) (p=0.00130*) (c=  95)
  1: committee-  46 (  131) (p=0.00053 ) (c=  37)
  2: launch-     72 (  272) (p=0.00028 ) (c=  21)
  3: north-      21 (  308) (p=0.00180*) (c= 103)
  4: pro-        16 (  339) (p=0.00290*) (c= 199)
  5: strategy-  101 (  395) (p=0.00016 ) (c=  13)
  6: weapon-     50 (  436) (p=0.00059 ) (c=  38)
  7: ced         98 ( 2654) (p=0.00017 ) (c=  13)
  8: dur        111 ( 3032) (p=0.00015 ) (c=  11)
  9: edu        111 ( 3139) (p=0.00015 ) (c=  12)
 10: gal         43 ( 3728) (p=0.00059 ) (c=  42)
 11: leu         57 ( 4673) (p=0.00051 ) (c=  29)
 12: nuc         56 ( 5305) (p=0.00052 ) (c=  30)
 13: ore         20 ( 5671) (p=0.00201*) (c=  97)
 14: rea         22 ( 6161) (p=0.00160*) (c=  84)
 15: rlo        127 ( 6253) (p=0.00013 ) (c=   9)
 16: ure         50 ( 7283) (p=0.00045 ) (c=  35)
 17: way         30 ( 7485) (p=0.00091 ) (c=  75)
 18: ko          27 ( 8075) (p=0.00149*) (c=  71)
 19: us          21 ( 8439) (p=0.00207*) (c= 122)
 20: bomb        53 ( 9186) (p=0.00025 ) (c=  15)
 21: rike        48 (10596) (p=0.00029 ) (c=  22)
 22: star        45 (10821) (p=0.00034 ) (c=  26)
 23: ucle        37 (11006) (p=0.00051 ) (c=  29)
 24: proce       63 (11863) (p=0.00017 ) (c=  13)
 25: strik       48 (11983) (p=0.00029 ) (c=  22)
nominal vector matches
 250:  14.4 standard deviations
 500:   8.8 standard deviations
1000:   4.3 standard deviations
%
```

The second column shows the actual n-gram, the third is its weight in the specified profile, the fourth is the n-gram index number, the fifth is the n-gram probability, and the sixth is the number of subsegments containing it. This profile relates to the North Korean atomic bomb.

To see the match list for profile 66 with DLST, we can enter the command line

```
% java -jar path/DLST 66
  1)  0:96      32.25
     ----------------
  2)  0:39      22.38
  3)  0:328     16.31
  4)  0:480     11.69
  5)  0:282     10.81
  6)  0:6       10.50
  7)  0:461     10.38
  8)  0:151      9.69
  9)  0:103      8.56
 10)  0:36       8.00
 11)  0:388      7.81
 12)  0:360      7.75
 13)  0:537      7.44
 14)  0:416      7.19
 15)  0:55       7.13
 16)  0:293      6.88
 17)  0:372      6.63
 18)  0:353      6.44
 19)  0:465      6.31
 20)  0:211      6.00
 21)  0:451      6.00
%
```

This shows matches in descending order of statistical significance, the lowest being 6.00 standard deviations. The highest match has a significance score that much greater than those immediately below it, indicated by the line of hyphens. The output is different from DLSS.

18. Tuning ActiveWatch For Target Text Data

Text in any given language can be quite diverse. You only have to compare a business memo to a scientific paper or to a social network posting on hip hop. They might all be written in English, but they may be barely intelligible to someone familiar with English only as taught in schools. The built-in n-gram indices in AW were intended for English in general.

An AW user, however, often has a particular target subset of text in mind; for example, American political partisanship, vaccine research, weather forecasts and warnings, or international terrorism. These narrower areas of interest often have distinctive vocabularies, which may be indexed less well by AW right out of the box.

There are three ways that an AW user can reorient n-gram indexing and make it more specific without changing any of its Java code:

- Morphological Stemming — usually avoided by AW users, but adjustments can often be handy, such as when target text has foreign names or technical nomenclature that is mishandled by normal English stemming rules.
- Stopwords — In information retrieval, English grammatical function words like THE, AND, or FOR are often called stopwords. In AW, they are normally filtered out of text data input so as to make n-grams like THE, AND, and FOR more helpful for indexing important text content. You may, however, add or remove individual stopwords on the AW master list; and if you want to do a stylistic analyis of text, you might eliminate all stopwords.
- Literal N-Grams — allow an AW user to define up to 2,000 special n-gram indices to augment AW built-in 2-, 3-, 4-, and 5-gram word fragment indices. These are called literal n-gram indices, and they are a way of getting indices with 6 or more letters and numbers. An AW literal n-gram must always begin or end a word, but otherwise they work just like regular n-grams in how they are counted up. A text word may both leading and trailing literal n-grams as well as regular n-grams for indexing.

The GitHub AW distribution includes default tables for for morphological stemming, for stopwords, and for literal n-grams. These should be adequate for demonstrating basic AW capabilities, but serious users will want to customize AW processing to particular target text data. For details on editing AW tables, see Appendix A.

19. Other ActiveWatch Directions

AW revolves around the control of noise in text indexing. This might suggest that noise is an enemy to be obliterated, but it is actually necessary for statistical scaling of similarity to work. In any event, some level of noise is unavoidable in any kind of text indexing. It shows up even when our indices are whole words.

For example, consider the word POST, not the 4-gram POST. One of its meanings is a tall physical object with a compact cross-section: e.g. a fence POST. Another indicates a sense of afterwards in time: e.g. POST-war. Another refers to a function of national government: e.g. POST office. It also may be about making information public: e.g. a Facebook POST. Simply counting an occurrence of the word POST in a text item will be noisy.

ActiveWatch wholly embraces noise to achieve its fixed, finite vector indexing of text. Noise allows different vector dimensions to be more independent. The demonstration system on GitHub shows how such independence can be exploited for automatic clustering, but that is only part of the story. As it turns out, one person's noise can often be someone else's signal; and AW can be reconfigured to suit everyone's purposes. Here are two other possibilities:

- Plagiarism is a major problem in the Internet age, especially in the essays required from students applying for admission to some colleges and universities. With just 2- and 3-grams, AW can find highly significant matches between a submitted essay and a reference collection of previously successful essays. This probably should not be the only criterion for judgment, but it can winnow out candidates for extra scrutiny.
- Documents handed over in the course of legal discovery for major litigation may be extra noisy on purpose. Complete gibberish would be contempt of court, but providers of dcuments have been known to use inferior methods of optical character recognition when required to furnish electronic copies of printed documents. If OCR errors are not random, however, AW can still find significant groupings in the data.

As time permits, the GitHub AW distribution will expand with modules to support other types of text processing.

20. The Bottom Line

How well does ActiveWatch actually work? The answer is complicated, like explaining how a casino can expect to be profitable. It all depends on stacking the odds in one's own favor. For AW, this goes beyond training blackjack dealers to hold on 17, but the principle is the same. Noise can never be entirely eliminated, but we can b oost the separation between signal and noise in n-gram indexing to squeeze the most information out of dynamic text data.

Consider the example of the alphabetic 4-gram RENT. This looks like something associated with landlords and leases, but to AW, it is just one of a possible half million sequences of four letters. It can be the word RENT, but it also can occur in curRENT, torRENT, paRENT, and tRENTon. Such ambiguity is the main source of noise in AW n-gram indexing of text.

A little noise, however, allows n-gram indices to be more independent, more so than with whole word indices. Such independence allows definition of a multinomial model of vector similarity. Vectors are really what we want to focus on, not individual n-grams like RENT. AW vectors will usually be fairly long, but finite, with about 10^4 dimensions. Through such vectors, we can discover major topics in a message stream, classify messages reliably with only occasional user intervention, and identify messages that stand out.

If you just want all messages containing the word RENT, then n-gram indexing is unnecessary and unwanted. The original Java AW even included its own conventional search engine that allowed users to run whole-word searches if they needed to; this is easy to do. What is hard is the "active" part of ActiveWatch, which is what AW on GitHub demonstrates. You have to see it before you can believe it.

Early AW experimental results showed that indexing with 1,296 alphanumeric 2-grams plus 5,616 selected alphabetic 3-grams had adequate separation of signal and noise for simple applications. For heavier industrial work. the latest AW versions expand indexing with selected alphabetic 4- and 5-grams. The 4-grams make the biggest difference: with more than 2,000 of them, they now account for almost as many occurrences in vectors as AW 3-grams.

(If you want to see the current complete listing of built-in alphabetic 4- and 5-grams, see the AW source file `gram/GramMap.java`. A listing of all alphabetic 2-gram seeds for 3-gram indices can be found in `gram/GramDecode.java`.)

All of this of course applies only for English text data. If you have to process Finnish, Croatian, Swahili, Malay, Quechua, or Hawaiian, then all bets are off. This contrasts with natural language text processing technology that seems to require minimal adaptation to the lexical characteristics of any particular language. AW recognizes the peculiarities of English and leans into them, which may be more consistent with how human children learn language.

In any event, play with ActiveWatch on your own data. Theoretic discussion of fixed, finite indexing is fine, but the proof of the pudding is in running AW with many different kinds of English text. It may well be that n-gram indexing needs to be developed beyond alphabetic 4- and 5-gram indices and alphanumeric literal n-grams. What we can see so far, however, is real. It may be surprising to many skeptics, but no smoke and mirrors are involved.

AW works as a system because of its strong attention to detail: extended inflectional and morphological stemming, blending of diverse built-in n-grams, counting of n-gram occurrences to increase indexing entropy, and providing two distinct statistically scaled similarity measures for different kinds of vector matching. AW tries to solve a few, though not all, problems well.

The GitHub AW release is still in progress, but decades of experience with its underlying framework has given us confidence in its efficacy. It should give you novel insights into your English text information resources. Although robots can now whup our tails in chess, go, poker, and other games, humans are still the supreme experts in natural language. We can whup convoluted neural nets on our home turf; and our deep linguistic savvy could even help them.

Appendix A. User-Defined Tables to Adjust Indexing

In theory, the finite indexing framework defined in preceding sections should perform quite well right out of the box. Our modern information universe, however, rewards people who can discover things before some other guy does. Everyone wants to glean some hidden nugget first from mountains or rivers of text data.

So, an information tool has to allow for customization and sharpening to give its users a competitive edge. AW accomplishes this through three user-editable tables to guide its finite indexing; these are defined mainly by three text files in a file directory from which AW is running: `stopword`, `suffix`, and `literal`.

A `stopword` table tells AW what words to ignore in building its text index vectors. For example, here are some actual entries in the default table for the GitHub download:

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

AW also supports an alternate way of defining stopwords with the file `stpats`. This allows a user to define patterns for stopping large classes of text tokens like telephone numbers. The patterns in `stpats` take the same form as those in Appendix B. Here are some examples of stop patterns:

```
##
#@
#####*
```

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 to add 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
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 code 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. A user can freely edit these three files, although new AW users probably should them all alone until they can understand how how the files will affect overall indexing.

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 up to 128 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 current actions 77 through 81; the actual numbers are only for the convenience of users. 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 (|) indicating 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 just to change the part of speech for a word or 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`, `stpats`, `suffix`, `action`, 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 may or may not increase the total number of clusters found in the AW demonstration, but may decrease the total number of clustered text items.

Appendix B. The AW Segmentation Automaton

ActiveWatch was originally designed to support watch officers monitoring streams of text messages arriving at a command center. In the old days, the officers sat in front of teletypes printing out arriving messages on fanfold paper. If something of interest showed up, an officer would have to tear it off and put it on one of many labeled clipboards hanging from hooks on a wall; sometimes, a message would be so important that the officer will call a superior immediately and make copies for distribution.

Command centers were noisy and often chaotic workplaces, but as computers invaded their operation, work stations with large displays replaced clackety teletypes and disk drives replaced the clipboards on the walls. The form of the messages remained the same, however: an indicator for the start of a message, a header with routing information and usually a subject line, the start of text, a text body, the end of text, a miscellaneous trailer, and an indicator for the end of a a message. A similar format was later adopted by Internet email.

AW at its heart is still a message processing system. Its first task was always to detect an incoming message and then divide it into parts and index only the text body. This process is guided by a delimiter file (default = `delims`), which contains transition rules for a simple finite-state automaton for scanning a message line by line, looking for where to make divisions. Such an automaton runs in the current AW SEGMTR module.

A state for an automaton remembers what it has seen so far, which will be quite limited. These states are identified by numbers, which can be assigned arbitrarily by a user, except for state 0, which must be the starting state for any automaton. At a given state and at a given input line, an automaton will check for matches with certain patterns. If one is made, the automaton can make a specified change in state for the pattern, signal an event, and possibly go to the next input line. Otherwise, the automaton will read the next input line without changing state.

The possible events in a message are

`NIL`	none yet to signal
`SOH`	start of header, which will begin a message
`SBJ`	found subject line in header
`SOT`	start of text body
`EOT`	end of text body
`EOM`	end of message

Transitions between states of an automaton will be defined in a delimiter file by a series of one-line rules expressed in the form

```
x y e a pat
```

where

`x`	is a beginning state
`y`	is a ending state
`e`	is an event type signaled for a delimiter match
`a`	is an action (+ . ?) to advance the input line or not
`pat`	is a pattern given as a string to be matched by an input line

(1) states are integer values from 0 to 99, determining which delimiter patterns are pertinent at each stage of processing by an segmentation automaton

(2) event types are `NIL`, `SOH`, `SBJ`, `SOT`, `EOT`, `EOM`

(3) possible actions on an arc are as follows

+	go to the next line
.	stay on the same input line
?	raise a fatal error

(4) patterns will be expressed as strings with certain characters having special significance:

[	define a class of specified chars
]	close a class of specified chars
~	take complement of class
@	match any alphabetic char
#	match any numeric char
*	match 0 or more arbitrary characters
?	match any char except \n
_	match a space or tab char
/	match end of a line
&	match 1 or more chars of an immediately following class
\	escape any character after it, negating any special significance

For example, the pattern

```
*ABC&[~12345]??\?/
```

matches the line

```
—ABC8::?
```

or the line

```
-!!!!!!-ABC999?
```

Input text is assumed to be ASCII. If you are processing Unicode, then you have the responsibility for doing any necessary conversion.

Most AW users will have no use for complex matching, but support for it is available if ever needed. The default AW delimiter file `delims`, used for the Google News input text, is in the GitHub AW distribution and is as follows:

```
0 0 NIL + /
0 1 SOH . #*
0 1 SOH . @*
1 2 SBJ . @*
1 2 SBJ ? #*
2 3 SOT + ?*
3 4 EOT . /
4 0 EOM + /
```

This assumes that an input text file has messages with no actual header or trailer, each separated only by one or more empty lines. Note that each definition line must end with either a `/` or a `*`. An entire input line must always be matched by a pattern.

Appendix C. Standing Profile Definition Files

ActiveWatch classifies text items by matching up their index vectors against an array of standing content profiles. These profiles usually are generated by AW itself from clusters of items found in a data input stream, but AW users can also add their own standing profiles specified either by a set of descriptive keywords or by a listing of examples of what to look out for. These profiles are produced by two AW modules: PROFLR for keyword definitions and EXMPLR for definitions by example.

Both PROFLR and EXMPLR take their input from text files of similar format. Each such input file will define multiple standing profiles, with the starting line of each definition as follows:

```
—— = xx.xx , nn
```

This header line has to start with at least four consecutive hyphens, optionally followed by a minimum significance match score and a specific profile ID number. For example,

```
——
—— =6.5
—— = 4 , 1
—— , 2
```

If a match score is omitted, 7.5 standard deviations is assumed; if an ID number is omitted, AW will assign the first available ID number. Users will probably want to reserve a specific range of ID numbers for their own standing profiles and specify those numbers in definition files.

AW will ignore all text lines before the first profile definition; and these might be used for commentary. For definition files read in by PROFLR, a header line will be followed by descriptive keywords; for example,

```
These will be comment
lines
—— =8 ,1
president donald trump
white house
—— =6 ,2
roy moore
alabama
senate
teenage girl
```

This file will define standing profiles 1 and 2.

The definition file ror EXMPLR would specify AW subsegment ID's instead of keywords; for example,

```
—— =7,3
0::454 0::435 0::385 0::381 0::423
—- -7.4
0::535 0::523 0::413
```

which will define standing profiles 3 and 4. When assigning profile ID numbers yourself, make sure that they are unique; otherwise, one definition will overwrite another. The CLSFYR module will compare user-defined standing profiles agalinst an input stream by default.

Appendix D. Where Built-In N-Gram Indices Came From

Historically, ActiveWatch began with indexing only by all 1,296 ASCII alphanumeric 2-grams just to see how far we could get. So, for the bit of English text PARTY, we get the 2-gram indices PA, AR, RT, and TY. AW allows indices to overlap so as to maximize the information in indexing. It will also disregard the relative position of any n-gram within a given word; the 2-gram PA will be the same in PAT or in SPAT or in DISPATCH.

Such indexing is quite noisy, but it can distinguish between small numbers of short text items in English and identify the similar ones. We can improve such 2-gram performance by preprocessing text to drop grammatical function words like THE and AND and to remove inflectional and morphological word endings, but in the end, 2-grams alone carry too little information for most text processing applications.

A logical next step is to index with 3-grams, but there are 46,656 alphanumeric 3-grams. Most of these will be nonsense: XXQ, LZR, A2J, 33L, etc., which will produce highly inefficient index vectors. When working with computational mechanisms like artificial neural nets, we typically want to limit the dimensionality of vector data input to something on the order of 10^4 or less. This is in line with the vocabulary of most people people, about 10^4 words.

To get a smaller index set, and to reduce index vector dimensionality, AW adopted a hybrid scheme with all alphanumeric 2-grams plus selected alphabetic 3-grams. The idea is to index everything with 2-grams, but where a selected 3-gram can be found, we go with it instead. For example, if ART is a selected 3-gram, then the text PARTY would be indexed with the 3-gram ART plus the 2-grams PA and TY. We no longer have to recognize AR and RT.

For a workable subset of alphabetic 3-grams, ActiveWatch takes the K most frequent alphabetic 2-grams in English as seeds and then appends a single letter to each, giving us a subset of K×26 alphabetic 3-grams. There are 676 alphabetic 2-grams; working from a published frequency list for cryptanalysis, we chose K = 216 so as to include QU as a seed. This produces 5,616 3-grams to index with. Together with our base 1,296 alphanumeric 2-grams, our hybrid index set grows to 6,912 n-grams.

We still gey nonsense 3-grams like QUJ, QUK, THJ, and OOU, but can now index long extents of English text almost entirely with 3-grams. This is done with well under 10^4 indices in all. It eliminates much of the noise arising from 2-gram indexing, but not all. Two completely unrelated text segments can still have many of our selected alphabetic 3-grams in common.

We can further reduce noise by adding alphabetic 4-grams to an index set. There are 456,976 possibilities, but most are nonsense. To keep within our practical limit of 10^4 indices, we need to work harder at selecting them, but this requires no skills beyond familiarity with English. The current AW list of 2,460 4-grams took many months to find and probably could be developed further. They mostly are true fragments, but some could be read as full words by themselves.

For example, AW selected 4-grams include: AGGL (hAGGLe), BLOO (BLOOm), CAVA (exCAVAte, CAVAlry), DORS (DORSal, enDORSe), ELTE (shELTEr, svELTE), FORD (afFORD), GRIN (GRINd), HOME (HOMErun), and so forth. All AW 4-gram indices were individually vetted to make sure that they show up in multiple English words. Each 4-gram had to pull its own weight in the indexing of content; none could be nonsense.

At a count of almost 2,500, 4-grams will play a major role in overall indexing, but 2- and 3-grams are still important. With the latest indexing of Google News data (v1.3.4), the total fraction of occurrences for 3-grams was about 36% versus 37% for 4-grams. For 2-grams, it was only about 6%, and for literal n-grams, it was only about 11%.

With its selection of about 9,300 2-, 3-, and 4-grams for AW built-indices, there is still room for further extension. One possibility is to have more 4-grams, but it has become hard to find suitable new ones in quantity. The logical direction here was to move up to alphabetic 5-grams,

of which there are 11,881,376 possibilities to choose from. Their indiviual probabilities of occurrence in text will be generally be lower than for 4-grams, however, and so we would have to add many thousands of them for any major impact on overall AW indexing.

Nevertheless, a subset of common English 5-grams can still serve a useful function in AW. A 5-gram occurrence in text will subsume occurrences of shorter 2- 3-, and 4-grams and reduce indexing noise. For example, the text CRYPT might analyze into the n-grams CRY, YP, and PT, but the 5-gram CRYPT (inCRYPT, CRYPTogram) would avoid spurious matches with CRY, YP, and PT when comparing two index vectors.

Alphabetic 5-grams in the latest AW indexing of Google News data now account for 11% of all n-gram occurrences. This is about the same as for default literal n-grams, but much lower than for 3- or 4-gram indices. Furthermore, more than half of AW selected 5-grams can be read as words, which is much less the case for 4-gram indices. There are many non-word 5-gram indices like DYNAM, GENER, and PREHE, but most are like COUNT, LUNCH, or METER.

For n-gram indexing, the ideal is for the indices to be frequent and widely scattered in text. That is because AW text analysis is really more about working with entire index vectors than about any single dimension of those vectors. When indexing with full words as in a search engine, however, we usually want indices to be less frequent and less scattered for better precision in matching. As a resut, the kinds of 5-letter words chosen for full-word indexing will have little in common with those chosen to be AW n-gram indices,

We can see this reaized in the AW demonstration system with Google News data. Making 5-gram indices out of an ESL list of the most common 5-letter English words had a surprising effect. It reduced the total number of AW output clusters by about ten percent while increasing the number of text items left unclustered. One might have expected that more indices would provide more information and more chances of finding links and more clustering of text items.

On the whole, alphabetic 5-grams frequent enough to serve as indices are probably too general to be helpful in finding clusters. Their main purpose in AW is in subsuming shorter n-grams more susceptible to noisy matching. Even here, their impact is limited because Zipf's Law tells us that there will be few 5-grams frequent enough to be worth bothering with. This suggests we probably should forgo adding alphabetic 6-grams to the AW built-in index set.

(Someone familiar with Java programming can change the AW built-in index sets. somewhat. Changes to 4- and 5-grams can be done by editing their definition lists in the AW source file `gram/GramMap.java` in the GitHub ActiveWatch repository; the current maximum number of 4-grams is 2,500. Changes to 3-grams will be too much trouble to be worth describing or doing, and 2-grams must include all possibilities in order to support complete indexing.)