ActiveWatch

A User's Guide

7 March 2023

Walnut Creek, CA

A Tool for Staying On Top Of
Unpredictable Text Data Streams
Through Reliable Statistical Modeling

D r a f t

Table of Contents

1. What Is ActiveWatch?

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

AW is “active” in that it can look for information on its own; this contrasts with a “passive” system, where a user has to ask explicitly for everything. Querying is a helpful capability for any information system, but when its data is dynamic, we can miss important events by failing to ask the right question at the right time. AW will automatically dig into fast-changing data and can warn users when they need to pay attention to unexpected shifts in content.

AW is NOT a search engine or a search engine replacement. Nor does it read and understand natural language text like a person. It is a purely statistical system that counts selected features in text data. The trick here is in knowing what to count and how to count it. This is a major challenge requiring knowledge of both natural language and information theory. AW excels at built-in familiarity with English and reliable probabilistic modeling of text.

AW was written entirely in the C language in the 1980’s and rewritten in Java at the turn of the 21st Century. It shows what anyone can accomplish by indexing text data with a fixed, finite set of linguistic features. The early 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 15 processing modules and 8 support modules, plus diagnostic tools; together, these can organize text items by content and find its major topics.

Other legacy AW processing modules written in Java may be added as time permits. The early Java AW system also included a simple graphical user interface based on the Java AWT (Abstract Window Toolkit), a standard library of interactive display widgets. A new AW GUI may come at some point. The current AW package is mainly for demonstration. It runs from a command line under Unix (including macOS 10+), Linux, or Windows on computer system supporting Java.

2. How ActiveWatch Works

Statistical text information systems 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 measure is normalized to fall within a specific interval such as [0 , 1]. This may look clean, but disregards the lengths of the 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, however, x and y are both long documents, while u and v are both short, then such theoretical near similarity may be much less evident to someone reading the text of u and v versus that for x and y. People all do tend to take length into account when assessing 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 any item x versus a specific query q, treated as a short text item. We only have 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 always be “good.”

When monitoring a live text stream, no ranking of items for a query will ever be final; and if many standing queries are turned on, then someone has to keep checking the top matches of each one to see whether it has really found anything yet. What we want here is to allow a system itself to know when a query has made a significant match and notify users as appropriate. This calls for a properly scaled similarity measure our system can itself interpret.

The threshold problem arises in a different way in an application like automatic clustering of text items by content. With a simple normalized similarity measure between 0 and 1, we can try to set a minimum match threshold 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 fuzzier similarity, but we cannot control the process as closely as we might want.

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

3. Fixed, Finite Indexing With N-Grams

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

Ideally, 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 have about the same frequency.

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

Whole words have always been problematic for indexing. They do cover all content of interest, but only when all words are features for indexing. Words will also fall short in our other requirements. They are not independent, often being synonyms or antonyms to some degree. Their frequencies are also highly skewed, with most words being quite rare. They are effectively infinite in number, because new words will keep popping up in any live data stream.

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 so 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, 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 currently indexes instead with a selected finite subset of short text fragments: all alphanumeric sequences of two characters, about 5,600 alphabetic sequences of three letters, 2,600 selected sequences of four letters, and 700 selected sequences of five letters. Users may also specify up to 2,000 alphanumeric sequences of any length, either beginning or ending a word. All such 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. For that, you should turn to any decent search engine. AW is instead intended for when you do not know what to ask for and need more than a little guidance. AW can work on its own to analyze unfamiliar data and give you insight into how it is changing. Random queries in the dark are never your only option.

The 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, letting us tackle practical problems of text analysis in novel ways.

In particular, when a text item is represented as a finite vector of the frequency of n-gram occurrences, 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. This noise model will let us gauge the statistical significance of any particular inner product. AW scoring of similarity will be in units of standard deviations (abbreviated as σ) above the expected value.

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 should usually set its match thresholds at 6σ or even higher, which is highly unlikely by chance. In other so-called statistical systems, we have no idea how much any 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 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, letting us simplify load calculations while maintaining reasonable safety margins.

In text processing, representing 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 from words, and ignores most word endings marking part of speech. We still have infinitely many stripped words 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 for 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 for finite modeling, given that unabridged English dictionaries have fewer than a million words, but we can refine our indexing by using a carefully selected mix of n-grams of short length.

The main point of 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 this is controllable. In general, we can approximate full-word indexing ever more closely by resorting to ever longer n-grams, though we always want to keep their total number reasonably low.

How long should our indexing fragments be in order 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 just of selected alphabetic 3-grams plus all alphanumeric 2-grams. The 2- and 3-grams will be allowed to overlap in words to maximize their 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 such a 3-gram at every position in a word is common; it just means that a word is recognizably English even though it might be 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, though, and having over 5,600 common alphabetic 3-grams plus all alphanumeric 2-grams will allow us still to make good distinctions between different segments of English text.

We saw no 2-grams in our example above, but need them to break down a word like CHUTZPAH, which is of Yiddish origin and far from 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 of either; so it does contribute its own bit of information to AW indexing.

With the full complement of AW built-in n-gram indices, including 4- and 5-grams, 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, adding frequent longer n-grams in English can improve the specificity of 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, which will make eethem less noisy and help to increase the overall entropy of indexing. Shorter n-grams will otherwise be more frequent than longer ones.

We can continue this way to improve AW 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 extend 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 near-words as indices, but we want the bulk of our indexing to be with true word fragments so that we can reasonably assume statistical independence.

6. AW 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 will be for nonsense features. From an engineering standpoint, a finite system model should be lean and mean. The solution here is to note that only a small subset of all n-grams is needed in an AW index set. Choosing them properly requires much effort, but pays off in better indexing.

The current AW built-in index set has 1,296 alphanumeric 2-grams, 5,616 alphabetic 3-grams, 2,600 alphabetic 4-grams, and 700 alphabetic 5-grams. The 2- and 3-grams have been in AW since its earliest days and facilitated the first experiments to show that fixed, finite indexing conveyed enough information to support useful text analysis. Alphabetic 4- and 5-grams are 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 repeated trial and error. The current selection can undoubtedly be improved, but it is sufficient for demonstrating a wide variety of AW functionality.

7. User-Defined Literal N-Grams

We can 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 in AW, they 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 overall 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, will be 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; for example, -DONALD.
- 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, then 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, except when subsumed by a longer literal. 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 have 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 its text. 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 one counting n-grams.

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 all 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, however, the overall usefulness of an index set for distinguishing between types of content will suffer.

Imagine a kitchen with a pantry full of many ingredients, but mostly potatoes, plus some turnips and parsnips. A good chef can perform wonders here, but how many different ways can we 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 other non-zero n-gram indices can become more significant.

Such transformation is a common practice among data scientists 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 transformation 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 should increase the entropy of any kind of indexing. A lower proportion of potatoes.

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 data by downloading the latest 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, a popular place on the Worldwide Web for developers to 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 all the latest AW files in their hierarchy of directories.

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

10. ActiveWatch Basic Processing 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 a long text item into multiple subsegments having sequential IDs of the form b::m; e.g. 2::123 (double colon).
profile	essentially a vector, but with floating point weights for n-gram indices instead of integer counts. Profiles will usually have at most dozens 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, often 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 indicate 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 oldest batch is deleted, and its number will be 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 out by one AW module and read in 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 algorithms and data formats. It also allowed different functionality to be supported by a reconfiguration of modules. When AW was subsitten in Java around 1999 for greater portability, it kept the various modules of the earlier C version and added 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 five groups: setup, core processing, clustering, stream monitoring, and reporting.

setup	STPBLD, SUFBLD, LITBLD, PATBLD, ENDBLD
	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 contains a list of stopwords, one per line, and writes a binary file `stps` for AW modules to read. You may edit the input file however you want, but it is probably a bad idea not to stop English grammatical 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 before n-gram indexing. but not from words like

CEMENT. The suffix input 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. You 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.

ENDBLD and PATBLD support AW phrase extraction by determining the part of speech of words undefined in the AW file WORDS. Later versions of this user guide will describe these modules in more detail.

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 details how to specify the logic for this automaton. If the text of an item found by the automaton is long, then SEGMTR will divide its 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 demonstration system, entropy is about 11.6 bits, but this can be improved with literal n-grams defined for a user’s target text.

clustering 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 a previously processed 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 similarity measures between every pair of vectors in a squeezed index vector file. It will then write out a file of links for pairs of vectors with a scaled similarity score above a given minimum significance. The AW demonstration system usually 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 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 have 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 a subsegment 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 a cluster. Items not matching any profile will be put on a residual list, structured as sequence file.

monitoring 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; see Appendix C for details. User profiles 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; see Appendix C for details. This can provide helpful indexing information even when you do not plan to run the resulting profiles with CLSFYR.

reporting KEYWDR, WATCHR, ANALZR, PHRASR

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).

ANALZR produces a phrase analysis of every text item through simple word lookup and syntactic analysis. This ii usually done in preparation for reporting of descriptive phrases for AW clusters.

PHRASR ranks the phrases in the items for each cluster according to the n-gram index weights in the descriptive profile for that cluster. The top N phrases will be reported for each cluster.

Other modules for **monitoring** and **reporting** may be in upcoming AW releases (see Appendix E). The shell file `awdemo` will run the sequence of modules for the AW clustering demonstration.

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 AW data 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 clustering demonstration with a single batch of Google News text

`action`	`attributes`	`clusters`	`control`	`counts`	`delims`
`ending`	`endss`	`heads`	`index00`	`links`	`lists`
`literal`	`lits`	`maps`	`offset00`	`parse00`	`pattern`
`patss`	`probs`	`profiles`	`range`	`residual`	`segmn00`
`sequence`	`source00`	`start00`	`stopword`	`stpats`	`stps`
`substitutions`	`suffix`	`sufs`	`svectors`	`vector00`	

AW creates a `control` file to track 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 to determine what text words not to index. SUFBLD reads the `suffix` and `action` text files to write the binary file `sufs` tcontrol 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.

To set up a directory for AW operations, 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. Files for other batches will automatically be numbered: e.g. `index01`, `index02`, and so forth.

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 items into one or more subsegments as 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 up to hard limit of 16,384. 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 yet.

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 demonstration.

ANALZR produces the `start`* and the `parse`* files identifying the phrases found in each text item of an input batch. PHRASR will use the output of ANALYZR plus the AW `maps,` `profiles`, `segmn`*, and `text`* files to derive descriptive phrases for each cluster and each match list for a user-defined standing profile.

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 more 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)		
ENDBLD	(none)		for phrase extraction
PATBLD	(none)		for phrase extraction
SEGMTR	-d D -l L -ll LL -ul UL File1 File2 …	use delimiter file D, defaults to `delims` set debugging output level to L, defaults to 0 set lower subsegment limit to LL, defaults to 1000 set lower subsegment limit to LL, defaults to 2500 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	for debugging; max of 16,384 set in code to limit clustering times
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

Module	Arguments		Comments

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
			a cluster will be dropped if no profile can be created for it

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.
ANALZR	`(none)`		Still experimental.
PHRASR	`(none)`		Still experimental.

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 7σ or higher. These will make clustering more restrictive, finding fewer clusters and assigning fewer items to them.

You can also reduce the minimum link threshold for MLTPLR. The algorithm in CLUSTR for obtaining densely linked subsets of items as seeds will make a low link threshold less noisy than one might expect. Processing time will increase because it will be proportional to the square of the total number of links abpve threshold, but you may want to go this route to minimize the rlsk of missing an important cluster.

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 approximately be proportional to the square of the number of items.

14. An Example of AW Clustering

Clustering is a common technique in data mining, but has a special status in AW because it is at the heart of what makes AW "active." The AW clustering algorithm is closely tied to statistically scaled similarity measures and is set up so that text items can be assigned to more than one cluster. After all, a sports story can also be about international politics.

Poor scaling of measures of similarity between documents can be problematic for text data. For example, Salton's cosine measure, which models a documents as a vector of word counts, can range in value between 0 and 1; but what does a cosine of 0.6 really mean? This does imply some similarity between two vectors, but proper interpretation requires knowing more about those vectors.

With AW fixed, finite indexing, similarity at 6 standard deviations means more or less the same statistically for any pair of documents. This makes it easier to set thresholds for matching and gives a user more control over the clustering of unfamiliar text documents. AW clustering is in two stages: (1) it obtains compact, highly interlinked cluster seeds indicating a significant area of content in documents; and (2) each seed can then be converted into a profile for actually assigning the documents to the cluster. Documents can be in more than one AW cluster.

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 v2.4. The listing of clusters has been truncated; there were actually 78 different clusters obtained with AW v2.4 for the Google News data.

KEYWDR will score stemmed words in a clustered item according to the n-gram weights for those words in the assignment profile for a cluster. The words are shown in descending order of their scores. For example, the highest scoring word in cluster 1 was `zimbabwe`, followed by `mugabe`. The lowest non-zero scoring words for the cluster were `old` and `resign`. These are less specific to the cluster, but still can help to indicate its content.

If n-gram indexing had no relation to content, then clustering of the vectors for text items would be random, and the descriptive keywords for the clusters would also look random. This has not been the case for any collection of text analyzed so far by AW. The keywords for a particular clustering run, however, will depend on how AW is stemming of words and the particular set of builtin and user literal n-grams used for indexing.

We can also make AW cluster seeds broader or narrower by adjusting parameters for in the command line for the module CLUSTR. These will set the minimum similarity for items in a cluster seed and for the minimum density of links required for a set of items to qualify as a cluster seed. AW cluster seeds should generally be compact and densely interlinked, making it easier to create assignment profiles for them.

The clusters will vary with the actual index set in effect for AW and with the current probabilities for the indices in a collection. The top clusters should be fairly stable, with only minor variation in their listed order. The choice of an 8 standard-deviation significance for cluster seed links and default 6 standard-deviation matching of cluster assignment profiles give us confidence that the results are real.

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. These are artifacts of the morphological stemming algorithm inherited by AW, which tries to eliminate letter sequences serving mainly to mark the part of speech of a word. These grammatical markers have less value when indexing text for content; and it is fairly easy to relate resulting word roots to actual words.

```
Active Watch release v2.4 (2022) Java - Keyworder
Copyright 1997-2002, 2022, CPM

deriving descriptive keys
------- cluster 1         (@ 1)
get keys from top 6 items out of 15
0::159 0::95 0::49 0::121 0::45 0::84
keys selected from 266 candidates

zimbabwe          mugabe            robert            preside
mnangagwa         emmerson          succeed           independ
harare            capital           rule              vice
lead              zanu              africa            fla
serve             new               state             address
head              struggle          event             took
thousand          know              decade            resign
------- cluster 2         (@ 2)
get keys from top 6 items out of 14
0::540 0::502 0::552 0::562 0::349 0::355
keys selected from 225 candidates

explode           suspect           injury            police
detonate          austin            person            dead
bomb              pack              connect           blast
treat             federal           texas             kill
people            hour              night             device
set               fedex             report            rock
authorit
------- cluster 3         (@ 3)
get keys from top 6 items out of 10
0::610 0::544 0::563 0::582 0::529 0::517
keys selected from 336 candidates

cambridge         facebook          analyt            firm
million           data              politic           company
zuckerberg        mark              found             profile
executive         committee         research          employ
develop           report            inform            chief
co                digit             usa               share
collect           app               work
```

```
------- cluster 4         (@ 4)
get keys from top 6 items out of 10
0::127 0::86 0::160 0::113 0::97 0::143
keys selected from 169 candidates

argentine         submarine         crew              juan
navy              san               report            south
miss              coast             membr             ara
atlantic          plata             communice         locate
search            del               contact           sub
know              base              balbi             disappear
area              country           novembr           mile
------- cluster 5         (@ 5)
get keys from top 3 items out of 25
0::493 0::406 0::531
keys selected from 121 candidates

douglas           marjory           stoneman          peterson
depute            scot              school            sheriff
florida           officer           resource          broward
gunman            shoot             parkland          build
county            office            month
------- cluster 6         (@ 6)
get keys from top 4 items out of 15
0::28 0::44 0::89 0::149
keys selected from 192 candidates

soldier           defect            korea             south
north             wound             surgery           condition
lee               border            jong              medic
guard             escape            cross             hospital
shot              university        person            milite
lead              fellow            suwon
```

The Google News text data is from late 2017 to early 2018. Its top 6 clusters are about (1) a coup in Zimbabwe; (2) package bomb terrorism in Austin, TX; (3) a British consulting company influencing the 2016 U.S. presidential election through illegally obtained Facebook data; (4) the disappearance of an Argentine submarine in the South Atlantic; (5) a mass shooting in a Florida high school; and (6) the defection under fire of a North Korean soldier.

The clusters found by AW do identify major topics in the Google News data. You can check on this by examining all seventy-plus clusters in the demonstration output using diagnostic tools described in Section 16. AW analysis of the data did not introduce any literal indices that an actual AW user might have immediately thought of: e.g. TRUMP, KOREA, or FACEBOOK.

Adding 4-grams to AW indexing should generally make it less noisy than with 3- and 2-gram indexing alone. Surprisingly, AW clusters remained more or less the same with 4-, 3-, and 2-grams compared to 3- and 2-grams, although the ordering of clusters did change. Actual AW users have found such clustering helpful for interpreting their data. You now can see for yourself what AW can do on your own text data.

15. AW Phrase Summaries for Clusters

AW phrase extraction dates back to its original C language implementation. The ANALZR and PHRASR modules together operate like the KEYWDR module to describe the results of AW automatic clustering. Phrases provide some context to keywords, which should then be easier for users to interpret. The algorithms for phrase extraction have to be fairly complex, however, and are still being tuned to improve their quality.

The current overall goal for Java ActiveWatch is achieve at least the capabilities of its earlier commercial C version, including phrase extraction. That capability has been much harder than just straight code translation, since Java is less friendly for text processing and also requires much greater attention to the architectural details in software. Release v2.0 of Java AW on GitHub was the first crack at the problem.

Here are the six top-scoring phrases extracted by the PHRASR module for each of the first eight clusters of the AW clustering demonstration with the Google News text sample. Compare with the keyword summaries of the same clusters in the previous section above.

```
Active Watch release v2.4 (2022) Java - Phraser
Copyright 1997-2002, 2022, CPM

phrase summarization for item clusters
------- cluster 1= 0::159 0::95 0::49 0::121 0::45 0::84
Zimbabwe
President Robert Mugabe
capital Harare
successor
Mutsvangwa
celebrate his stunning
------- cluster 2= 0::540 0::502 0::552 0::562 0::349 0::355
explosion
Texas
device
two suspicious packages
package bombs
injured
```

```
------- cluster 3= 0::610 0::544 0::563 0::582 0::529 0::517
develop psychological profiles
50 million user profiles
co-founder Brian Acton
scrutiny from federal investigators
data analytics
brokerage Pivotal Research
------- cluster 4= 0::127 0::86 0::160 0::113 0::97 0::143
ARA San Juan submarine
submarine San Juan
Atlantic coast south
south Atlantic
navy submarine carrying
far southern Argentina's
------- cluster 5= 0::493 0::406 0::531
Peterson
Deputy Scot
school resource officer
massacre
Florida’s
Sheriff’s
------- cluster 6= 0::28 0::44 0::89 0::149
Korean soldiers
Surgeon
defect by crossing
escape released
University Medical Center
University Hospital
------- cluster 7= 0::268 0::257 0::542 0::209 0::304 0::30
deal negotiated
Parliament
exit from Brexit
Theresa
Northern Ireland
Varadkar
------- cluster 8= 0::432 0::469 0::444 0::457 0::564 0::589
aluminum imports
steel imports
impose steep tariffs
new U.S. tariffs
trade measures
steel products
```

No artificial intelligence is involved in selecting these descriptive phrases for clusters. AW employs the same algorithm as in its KEYWDR module to assign a score to individual keywords in the text of items. Phrases are then scored by summing the scores of the keywords contained in them. If multiple phrases have the same keywords in the same order regardless of any non-keywords, then only one of them will be reported.

If the sequence of keywords in a descriptive phrase is a subsequence of the keywords of a longer descriptive phrase, then AW will report only the longer phrase. A phrase may consist of only a single keyword, but PHRASR will report this only if that keyword is in no other longer descriptive phrase. Keywords are always stemmed for any matching; e.g. DEPUTY will match DEPUTIES and DETONATE will match DETONATION.

16. AW Reporting of Residuals

AW statistically scaled n-gram similarity measures allow the setting of interpretable match thresholds. The default significance for matching cluster profile is 6 standard deviations (σ), which should be highly unlikely by chance. You may crank this up or down to get tighter or looser item assignments to clusters.

For example, a low threshold of 3σ will be noisy, but a user may want to avoid missing something. With any positive threshold, however, some items may remain unassigned to any cluster. Depending on the match threshold, as many as a fifth of all items could be unclustered. These will be called residual items

AW keeps track of all residuals and will allow them to form clusters with later items in an input stream. Meanwhile, the AW WATCHR module will look for items that stand out in the sum of the probabilities for their n-gram indices. This will give users yet another way to discover items of possible interest.

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

```
Active Watch release v2.4 (2022) Java - Watcher
Copyright 1997-2002, 2022, CPM

scanning segments in residual file
24 segments selected from 145 candidates
  1)  0::399   0.02184183 (  80)
  2)  0::266   0.02470781 (  85)
  3)  0::171   0.02589450 (  85)
  4)  0::104   0.02877166 ( 124)
  5)  0::69    0.03365277 ( 130)
  6)  0::283   0.03408938 ( 106)
  7)  0::570   0.03501859 ( 119)
  8)  0::427   0.03527607 (  83)
  9)  0::70    0.03563432 ( 124)
 10)  0::155   0.03607093 ( 106)
 11)  0::60    0.03628364 ( 145)
 12)  0::351   0.03656352 ( 118)
 13)  0::231   0.03680982 ( 114)
 14)  0::324   0.03921678 ( 128)
 15)  0::327   0.04036989 ( 143)
 16)  0::40    0.04190363 ( 114)
 17)  0::295   0.04231785 (  87)
 18)  0::374   0.04262013 ( 103)
 19)  0::573   0.04263132 ( 125)
 20)  0::343   0.04310152 ( 117)
 21)  0::139   0.04321347 ( 154)
 22)  0::453   0.04328064 ( 119)
 23)  0::402   0.04359411 ( 117)
 24)  0::254   0.04361650 ( 108)
```

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 subsegment. 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.

17. Diagnostic Tools

AW diagnostic tools were developed to debug the code of AW modules. This is still important, but curious users 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.

Twenty-two tools are currently included in the latest AW distribution from GitHub. These have been compiled and put into executable jar files with 3-, 4-, and 5-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
`DINF w …`	shows English inflectional stemming of each word argument
`DIXS`	shows the index records identifying individual text items in a batch of text data processed by the `SEGMTR` module
`DKYW id …`	select descriptive keys for profile 0 from text items given as commond line arguments
`DLNK`	shows the links abo (does nor save)ve threshold 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 also by users
`DNGM`	shows the relative importance for indexing of user-defined literal n-grams, alphanumeric 2-grams, and alphabetic 3-, 4-, and 5-grams.
`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` [-s] n	show the n-gram index weights and thresholds for profile n; -s option shows expected significance of matching all indices with item vectors of shorter nominal length (e.g. tweets)
`DQBE m l id …`	builds profile from item ID arguments (see below) where m is the min count for a profile index and l is minimum profile index count
`DQBK w …`	builds profile from word arguments (see below)
`DSCN ng`	finds text segments containing an n-gram index expressed as an integer code ng
`DSEQ`	shows an AW sequence produced by `SEQNCR`
`DSIM id id`	shows details of scaled similarity compution for two items
`DSMX n id`	shows details of scaled similarity compution for item id versus profile n
`DSQV`	shows squeezed vectors produced by `SQUEZR`
`DSRV n l t b`	scans the first `l` items in batch b to find matches above threshold `t` for profile n, with defaults n = 0, `l` = 100, `t` = 6.0, b = 0 (see below)

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 –n k str	shows the n-gram analysis of a given text string str with no stemming and, if a -n k argument is given, do only for n ≤ k (default k = 5)
TGx < in	shows the n-gram analyses of text from standard input with stemming
TKNZR file	shows stemming and stopword removal in getting successive tokens from a text file

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 need requires and time permits.

Unlike its original C implementation, the current Java version includes no search engine. This is because plenty of options are available elsewhere; it is hardly necessary to duplicate such functionality in AW itself. Yet, basic searching is already supported by methods in the Profile and Vector classes of Java AW, and having access to them is helpful for system testing and for understanding details of a deta set analyzed by AW.

So, AW allows a batch of text data to be scanned for matches with either a profile generated by AW automatic clustering or by a standing profile defined by an AW user. These are assigned unique identifying numbers 1 through M by AW. So, we can in effect employ the DSRV tool search a batch of text data for the best L matches. The DSRV tool allows the significance threshold stored with a profile to be overridden to do what-if testing.

In addition, AW defines a profile number 0, which a user can create dynamically from a command line with the DQBK and DQBE tools. DQBK approximates querying with keywords, while DQBE approximates querying by example. They do not, however, operate as one might expect from a search engine. The main complication here comes from AW reliance on statistically scaled match thresholds.

In particular, if a query word is quite common in a data set, then a simple query with it will return nothing at the AW default of six standard deviations. It may be necessary to go with any match above zero standard deviations, which might feel strange, but is consistent with how statistical significance works. This is how a command line search might look

```
java -jar path/DQBK state
java -jar path/DSRV 0 1000 1.0
```

Profile 0 is created by DQBK. DSRV matches this profile against the first 1,000 items of batch 0 with a threshold of only 1 (!) standard deviation. Query by example with DQBE, is slightly easier; we may want to leave the profile match threshold at its default of 6 standard deviatioons.

18. 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. The tools will show how well AW is working.

Here is sample output produced by DNGM in AW v2.4 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=0.110038 for 342 indices
min=0.000011, max=0.004657
--
Alphanumeric 2-grams
total prob=0.056032 for 520 indices
min=0.000011, max=0.002261
--
Alphabetic 3-grams
total prob=0.352033 for 2727 indices
min=0.000011, max=0.002385
--
Alphabetic 4-grams
total prob=0.374468 for 2086 indices
min=0.000011, max=0.004333
--
alphabetic 5-grams
total prob=0.107429 for 563 indices
min=0.000011, max=0.003426
--
6238 total non-zero n-gram indices
```

This indicates that alphabetic 4-grams are pulling the most weight in indexing according to the sum of their probabilities, which means that we selected them well. The more numerous alphabetic 3-grams are still important and come in a close second. Less closely afterwards are literals, 5-grams, and 2-grams, in that order. For better 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 2,580 built-in 4-grams defined for AW v2.4, only 2,086 actually occurred in the Google News demonstration data included with the GitHub AW repository.

We can look at n-gram probabilities in another way by running the DPRB tool to get the top 24 indices for the Google News data (the default is 16):

```
% java -jar path/DPRB 24
analysis of probabilities

probability range= 0.000011 : 0.004657

6238 non-zero indices with sum of probabilities=1.000000
based on 89324 total occurrences of indices
computed entropy = 11.6 bits, 91.9 percent of maximum
(saved entropy= 91.9)

24 indices with highest percentages of occurrence

 preside-         (  333):  percent=0.4657
 trum            (11381):  percent=0.4333
 rump            (10995):  percent=0.4310
 office-          (  310):  percent=0.3504
 state           (12323):  percent=0.3426
 year            (11569):  percent=0.3280
 pro-             (  335):  percent=0.2978
 report-          (  361):  percent=0.2765
 time            (11311):  percent=0.2664
 new              ( 6417):  percent=0.2385
 people-          (  319):  percent=0.2295
 20               ( 3035):  percent=0.2261
 lead            (10200):  percent=0.2205
 us               ( 2739):  percent=0.2127
 ore              ( 6971):  percent=0.2015
 -son             (   45):  percent=0.1892
 day              ( 4209):  percent=0.1881
 act              ( 3346):  percent=0.1858
 north-           (  305):  percent=0.1847
 house           (12025):  percent=0.1825
 -russia          (    1):  percent=0.1746
 01               ( 2964):  percent=0.1735
 man              ( 6122):  percent=0.1702
 -nation          (   42):  percent=0.1679

total percentage of occurrences for top 24 =  6.14
```

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 indexing. 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::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.

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

The diagnostic tools DPRO and DLST let a user look at the profile for a given cluster and the match list for that profile in descending order of significance. A profile is really an n-gram index vector with fewer indices having non-zero weights, which will no longer be counts of n-gram occurrences in a particular text item. The highest n-gram weight always will be 127, with those for other n-grams proportionally scaled down. The expected value and variance are that of the noise distribution for the statistical scaling of match scores; they have to be multiplied by the total count of n-gram index occurrences for an item being matched.

Here is the output for profile 64.

```
% java -jar path/DPRO 64
profile 64
significance threshold= 6.0
expected value= 0.810857
variance      = 56.34201

weights
  0: -iate       62 (   21) (p=0.00072 ) (c=  55)
  1: accuse-     66 (   73) (p=0.00083 ) (c=  63)
  2: harass-    101 (  233) (p=0.00036 ) (c=  22)
  3: report-     26 (  361) (p=0.00277*) (c= 172)
  4: alt        102 ( 3476) (p=0.00019 ) (c=  16)
  5: apo         78 ( 3549) (p=0.00032 ) (c=  23)
  6: duc         60 ( 4317) (p=0.00054 ) (c=  41)
  7: fel        127 ( 4846) (p=0.00012 ) (c=  10)
  8: opr        116 ( 6958) (p=0.00015 ) (c=  12)
  9: orw         76 ( 6989) (p=0.00034 ) (c=  25)
 10: pol         78 ( 7264) (p=0.00032 ) (c=  23)
 11: pri         80 ( 7313) (p=0.00030 ) (c=  25)
 12: sex         44 ( 7848) (p=0.00102 ) (c=  62)
 13: stu         75 ( 8027) (p=0.00035 ) (c=  28)
 14: tun        102 ( 8280) (p=0.00019 ) (c=  17)
 15: uct         56 ( 8364) (p=0.00063 ) (c=  48)
 16: cond        49 ( 9396) (p=0.00082 ) (c=  61)
 17: diff        91 ( 9533) (p=0.00024 ) (c=  20)
 18: ffer        84 ( 9789) (p=0.00028 ) (c=  23)
 19: iffe        91 (10040) (p=0.00024 ) (c=  20)
 20: misc        79 (10377) (p=0.00031 ) (c=  24)
 21: ondu        60 (10559) (p=0.00055 ) (c=  41)
 22: prop        85 (10778) (p=0.00027 ) (c=  20)
 23: riat       112 (10905) (p=0.00016 ) (c=  13)
 24: scon        68 (11035) (p=0.00043 ) (c=  31)
 25: ology       72 (12134) (p=0.00038 ) (c=  28)
 26: speak       59 (12293) (p=0.00056 ) (c=  43)
nominal vector matches
 250:  16.0 standard deviations
 500:  10.1 standard deviations
1000:   5.4 standard deviations
```

The second column of the main profile listing shows the actual profile n-gram, the third is its weight in the specified profile, the fourth is the AW 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 accusations of sexual harassment by wormen.

To see the match list for profile 64, we can enter the command line with DLST

```
% java -jar path/DLST 64
  1)  0::332     24.75
  2)  0::277     22.19
  3)  0::64      22.00
    ----------------
  4)  0::286     16.25
  5)  0::73      13.69
  6)  0::325     13.31
  7)  0::202     10.63
  8)  0::392      8.81
  9)  0::335      8.31
 10)  0::482      7.75
 11)  0::271      7.63
 12)  0::301      7.50
 13)  0::308      7.38
 14)  0::27       7.25
 15)  0:::261      6.81
 16)  0::361      6.75
 17)  0::132      6.56
 18)  0::362      6.13
```

Matches are of vectors for text subsegments in descending order of significance, The lowest is 6.13 standard deviations; the highest is at 24.75 standard deviations. The top three, separated by the line of hyphens, are much more significant than those just below. These are probably the seed for cluster 64.

19. Tuning ActiveWatch For Your 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 hiphop. They might all be written in English, but they may be barely intelligible to someone familiar only with English as formally taught in schools. The AW n-gram indexing approach will be advantageous here because it is less sensitive to both tbe style adn the content of iext data.

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. For best performance in critical situations, it can helpful to adjust AW processing in various ways.

An AW user can reorient n-gram indexing for specific target text without changing any of its Java code. This can be accomplished by editing tables loaded by AW at startup.

- 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 analysis 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 get maximum indexing entropy fo particdular data. For details on editing AW tables, see Appendix A.

20. Other ActiveWatch Directions

AW revolves around the control of noise in text indexing. One might think 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 documents 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 through its n-grams.

As time permits, the GitHub AW distribution will add on modules from the old 20th Century AW system to support other types of text processing.

21. The Bottom Line

ActiveWatch had both a government and a commercial user in the 1980's and 1990's, but was never a shrink-wrapped software product. The rapid advance of computer architectures and operating systems made AW difficult to maintain. This led to an effort to reimplement AW in Java for portability, but that work had to be abandoned. The current Java AW finally completes the reimplementation, bringing its code into the 21st Century as free open-source software.

How well does AW actually work? The answer is complicated, but is like how a casino can expect to be profitable consistently. We only have to stack all the odds in our own favor. For AW, that goes beyond training blackjack dealers to hold on 16, but the principle is the same. Some randomness and luck is inevitable, but we can do much boost the separation between signal and noise in n-gram indexing to wring more information out of dynamic text data.

On first hearing about finite indexing of text data with word fragments, most people think that it will be too noisy ever to be useful. It is no magic bullet to be sure, but as any crossword puzzle enthusiast can tell you, word fragments do carry non-zero information. With the fragment RENT, we can have curRENT, paRENT, RENTal, or tRENTon; but modern computational power can perform wonders in piecing information together out of vector representations of text.

A little noise also allows n-gram indices to be more independent than whole words as indices. Words carry meanings, and these often will be the same or the opposite of those for other words, making them far from independent as vector dimensions. In contrast, n-grams are mostly meaningless: is ALLE related to THQ or ZZ in any obvious way? Some n-grams like SEA and WATER are sometimes related, but then we also find them in SEArch and WATERbaby.

Greater independence of indexing features allows AW to define a multinomial model of text vector similarity. We will focus on those vectors, not on individual n-grams like RENT. The vectors will be fairly long, but finite, with about 10^4 dimensions. Through them, we can discover major topics in a message stream, classify messages reliably with only occasional user intervention, and identify messages that stand out from their background.

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. It is really unlike any other text analysis system.

Although finite indexing is unnecessary for matching text stream segments to standing profiles or for clustering them by similarity of content, it will give AW users greater control over such processing. Statistical control can provide a critical edge in applications where users want to observe how a text stream is evolving and avoid nasty surprises.

Early AW experimentation 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-gram indices. The 4-grams make the biggest difference: only 2,600 of them now can account for a bit more indexing occurrences in normal text than AW 3-grams.

(To view 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 works only for English text data. If you want to tackle Finnish, Croatian, Swahili, Malay, Quechua, Hawaiian, or Klingon, then all bets are off. This contrasts with natural language text processing that applies machine learning to text data with minimal adaptation to the known characteristics of any particular language. AW recognizes the peculiarities of English and leans into them, consistent with how human children learn a language.

AW is and remains a unique kind of natural language processing system. This ranges from its overall architecture down to what it does at the lowest levels of analysis. Unique is not necessarily good, but sometimes we want to look at text data in a fresh way. Everyone should have more choices in an analytic tool belt than just a hammer.

In any event, try ActiveWatch on your own data. Theoretic discussion of fixed, finite indexing is fine, but the proof of the pudding is in running it on your own kinds of English text. It has already proved effective in both commercial and military text processing. What AW can do is quite real. No smoke and mirrors are involved in its application to profile matching and automatic clustering.

AW excels because of close attention to detail: its inflectional and morphological stemming to reduce words to root forms, its curation of 10^4 built-in n-gram indices for efficiency, its blending of built-in and user-defined literal n-grams for better coverage, its logarithmic counting of n-gram occurrences to increase indexing entropy, and its two distinct statistically scaled similarity measures for different degrees of freedom in vector matching. AW is still imperfect in its mastery of English, but within the realm of statistical expertise, it shines.

(If you want to dive into the mathematics of ActiveWatch and the technical details of n-gram indexing, please refer to the paper "How To Index" (`howtoindex.pdf`). This is included in the top directory of the ActiveWatch repository on GitHub. The writeup also recounts some of the history of ActiveWatch, including early experiments to confirm signal-to-noise separation in indexing in a wide range of text.)

AW in Java is still only a demonstration system, but its organization of classes for an object-oriented architecture should provide a skeleton for future operational systems with full-fledged graphical user interfaces. The problem of exploiting information in unpredictable text data streams remains hard, but still needs addressing. AW fixed, finite indexing offers only an incomplete solution here, but its finite vectors are friendly to statistical analysis and can be a good core for wrapping other technologies around.

AW users need not understand all the details of how text data is indexed. Familiarity with Gaussian and multinomial distributions will be helpful, but It is enough just to have a practical sense of what the σ scale means for setting thresholds to control automatic clustering or data stream matching. With reliable match thresholds, AW can operate with less supervision and can be truly more active in dealing with unpredictable data streams.

The GitHub AW release remains a work in progress after many decades, but experience with its underlying framework has given us confidence in its effiectiveness. AW can give you broader 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 go-to experts in natural language. Much of our linguistic savvy can be built into knowledge-based systems like AW to produce vector models of text that even the deepest neural net can love.

Appendix A. User-Defined Tables to Adjust Indexing

Usually, 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 shown 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 in English usually change the part of speech for a word or a word root, but in AW can serve to nornalize a word. They can be applied recursively.

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 hung on 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 . @*
0 1 SOH . "*
1 2 SBJ . @*
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. The final line of an input text file will have to be have to be an empty line 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. User Standing Profile Definition

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 for 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 against an input stream by default.

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

The term “n-gram” in AW refers to a sequence of n letters or digits in text. This usage is common in cryptanalysis, but for Google, n-grams are sequences of n words. Since AW is older than Google or even the Internet, we have kept the cryptanalysis meaning of n-grams to be consistent with previously published documentation.

Historically, ActiveWatch began by 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.

With a finite set of 2-gram indices, we can represent any segment of text as numerical vectors in a space of 1,296-dimensions, where each vector component is the count of a particular 2-gram in a text segment. This is a severely simpified reresentation of a segment, but we can show that there is still information for useful analyses of content. The AW demonstration system will let users try out indexing with 1 < n < 5 by passing an optional `-n` n argument to the module INDEXR. For example, you can set n = 2 to see what happens.

Indexing with only 2-grams is quite noisy, but can distinguish between small numbers of short text items in English and identify similar ones. We can improve indexing performance here 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. This is clear from indexing entropy scores.

A logical next step is to index with 3-grams, but there are 46,656 alphabetic 3-grams. Most of these will be nonsense: XXQ, LZR, AZJ, IIL, etc., which will produce highly inefficient indexing. 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, about 10^4 words.

To get a more efficient index set and to reduce index vector dimensionality, AW adopts a hybrid scheme with all alphanumeric 2-grams plus selected alphabetic 3-grams. The idea is to index everything with all 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 here.

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 get nonsense 3-grams like QUJ, QUK, THJ, and OOU, but can now index long extents of English text almost entirely with 5,616 3-grams. This is all still 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 our index set. There are 456,976 possibilities, again mostly 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 spelling. The current AW list of 2,600 4-grams took months to compile and could be developed further. Most are true fragments, but some could also be read as full words by themselves.

For example, currently selected AW 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 such 4-gram has to pull its own weight in the indexing of content; none could be nonsense.

At a count of 2,600, 4-grams will play a major role in overall AW indexing, but 2- and 3-grams are still important. In the indexing of Google News data with AW v2.4, the total fraction of non-zero vector occurrences for 3-grams was about 36% versus 37% for 4-grams. For 2-grams, it was only about 6%, and for default 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. It was logical here to think about alphabetic 5-grams, of which there are 11,881,376 possibilities to choose from. Their individual probabilities of occurrence in text will be generally be lower than for 4-grams, however, and so we have to add many hundreds of non-nonsense ones 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 some 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 many are like COUNT, LUNCH, or METER.

For n-gram indexing, the ideal is for the indices to be frequent and widely present 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 result, the kinds of 5-letter words chosen for full-word indexing may be quite different from those chosen to become AW 5-gram indices,

We can see this realized 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. This is reassuring; with more indexing information, we should be able to distinguish between text segments better.

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 will be in subsuming shorter n-grams as described above. Even here, their impact will be limited because Zipf's Law tells us that there will be few 5-grams frequent enough to be worth bothering with. This rarity problem probably would get worse if we try to add alphabetic 6-grams to our indexing.

Abyone familiar with Java programming can change the AW built-in index set. This can get complicated, however; and most users will probably just want to add indices and can do it most easily by defining them as leading or trailing literal n-grams. AW 4- and 5-grams are listed in the AW source file `gram/GramMap.java`. Changes to 3-grams will be too much trouble to be worth describing or doing, and 2-grams must always include all combinations in order to support complete indexing.

Appendix E. More AW Modules

We can extend ActiveWatch capabilities by implementing new modules tapping into the data files produced by older modules. Those older modules were described in Section 13; they will typically make up an update sequence bringing a new batch of text into a data set. AW release v2.7 introduced three analytic modules to help users explore further the content of a large collection of unfamiliar text data:

`PLOTTR`	generates a `.csv` file showing the match of successive items in a batch versus a reference cluster profile; the actual data plotting should be done with a spreadsheet app. The plots will show how items associated with a topic cluster are distributed over time in an input batch seen as a data stream.
`RANKER`	identifies the items that match the most cluster profiles, i.e. that cover the most major content areas. These items will be a good place to look in order for a quick sense of the overall content of a text data set.
`HUBBER`	identifies the items with the most links to other items; these will be the major hubs in the graph of all items in a data set and their connections. The data view is different from RANKER, since it ignores any AW topic clustering in its reporting.

Their command line arguments are as follows:

Module	Arguments		Comments

PLOTTR	`Batch`
	`Cluster`
	`binWidth`
	`minSig`
		number
		number
		how to bin counts to be plotted, default is w=1
		in standard deviations for cluster profile match
			produces of matches versus a profile for plotting

RANKER	`minSig` `minNoC`	in standard deviations for cluster profile matches minimum cluster count for reporting	
HUBBER	`minSig` `minNoC`	in standard deviations for pairwise link minimum link count for reporting	a standard kind of graph analysis

PLOTTR bin width currently is specified with an integer argument preceded by “`w=`“ with no intervening space; for example “`w=20`”. Or the argument may be the name of a file assigning differerent widths fo each bin, one bin per input line. This may change.

The HUBBER module was part of the original port of AW to Java over twenty years ago. The others are new, though they mainly exploit existing capabilities of AW automatic clustering.