

# ActiveWatch

## A User's Guide

16 December 2021

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

D r a f t

## Table of Contents

1. What Is ActiveWatch?	3
2. How ActiveWatch Works	3
3. Fixed, Finite Indexing With N-Grams	4
4. Information Engineering With Finite Modeling	5
5. A Rule for Non-Redundant Indexing	7
6. User-Defined Literal N-Grams	7
7. How to Count	9
8. Getting ActiveWatch Source Code and Jarred Modules	9
9. ActiveWatch Basic Elements	10
10. ActiveWatch Architecture	10
11. ActiveWatch File Environment	13
12. Control of Processing in AW Modules	14
13. Example of AW Clustering	16
14. Example of AW Residuals	19
15. Diagnostic Tools	20
16. Tuning ActiveWatch For Your Particular Text Data	23
17. Other ActiveWatch Directions	24
18. The Bottom Line	25
Appendix A. User-Defined Tables to Guide Indexing	26
Appendix B. The AW Segmentation Automaton	28

# 1. What Is ActiveWatch?

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

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

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

AW was written entirely in the C language in the 1980’s and rewritten in Java at the turn of the 21st Century. This software demonstrates what could be done when indexing text data with a fixed, finite set of text features. The Java code was never released, but in 2021, it was recovered from a backup CD-ROM and cleaned up to compile with the latest versions of Java. The current AW system consists of 11 processing modules plus 3 support modules; together, these implement a distinctive way to cluster text items by content to delineate its major topics.

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

# 2. How ActiveWatch Works

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

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

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

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

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

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

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

### **3. Fixed, Finite Indexing With N-Grams**

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

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

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

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

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

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

Consequently, ActiveWatch instead works with a selected finite subset of short text fragments for indexing: all alphanumeric sequences of two characters, about 5,500 alphabetic sequences of three letters, about 2,000 sequences of four letters, and over 500 sequences of five letters.

Users may also specify up to 2,000 alphanumeric sequences of any length, which either begin or end a word. This set of text features will be called n-grams.

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

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

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

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

## 4. Information Engineering With Finite Modeling

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

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

The set of all n-grams up to some given 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 could be broken into 5-grams

```
SUPERCALIFRAGILISTIC
super
  calif
    ragil
      istic
```

So, we might count four 5-grams here instead of one long word. This adds some noise to our indexing, but 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. If the noise is unacceptable, we could move up to 6-grams. In general, we approximate full-word indexing ever more closely as n increases.

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

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

```
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 3-gram occurrence in a text segment regardless of its relative position in a word. All the 3-grams above are from the set of about 5,500 built into AW. Finding an indexing 3-gram at every position in a word is not unusual; it indicates that the word is recognizably English even though it is completely made up.

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

We saw no 2-grams in our analysis above, but we shall need them to break down a word of foreign origin like CHUTZPAH, which is hardly English-like. Its overlapping 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 for indexing, which calls for a bit of explanation.

## 5. A Rule for Non-Redundant Indexing

ActiveWatch can 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 MINIMUM 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 inside either; so it does contribute a bit information to our AW indexing.

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

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

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

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

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

## 6. User-Defined Literal N-Grams

It is possible to run ActiveWatch with just its built-in index set of all alphanumeric 2-grams and selected alphabetic 3-, 4-, and 5-grams. This should be adequate for any English text data, but users can sharpen their indexing of specific target content by defining up to 2,000 of their own n-grams, called literals. Literal n-grams will be integrated with built-ins, but will follow different rules in AW indexing.

- Literals may be alphanumeric character sequences of at least length 3; a user may not define any 2-gram as a literal, since these are already built into AW indexing.
- They must either start or end a word or word-like token. The former will be called leading literals; the latter, trailing literals.
- A literal may cover an entire word. This is how users might approximate indexing with whole words.
- If a leading literal and a trailing literal both cover a full word, then only the leading literal is counted.
- If a trailing literal covers a full word, and leading literal covers only part of a word, the only the trailing literal is counted.
- A leading or trailing literal CANNOT be subsumed by a longer built-in n-gram. It will always be counted, subject to non-redundancy. If the literal is the same length as a built-in n-gram, then the literal WILL be counted instead of the built-in.
- The non-redundancy rule still applies in full. Even if a leading and trailing literal overlap, AW may find a longer built-in n-gram that subsumes the overlap.

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 is how AW would analyze it with its non-redundancy of indexing rule. 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.



## 7. How to Count

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

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

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

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

Such transformation is a common practice among statisticians working with text data. Instead of a raw count, they often use the square-root or the logarithm of the count, which will dampen out 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 will increase the entropy of any kind of indexing.

## 8. Getting ActiveWatch Source Code and Jarred Modules

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

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

<https://github.com/prohippo/ActiveWatch>

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

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

## 9. ActiveWatch Basic Elements

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

<b>text item</b>	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.
<b>index vector</b>	as already described above. These vectors will represent subsegments of text items. AW will split long text item into multiple subsegments having sequential IDs of the form b::m; e.g. 2::123.
<b>profile</b>	essentially a vector, but with floating point weights for n-gram indices instead of integer counts. Profiles will have only a few indices with non-zero weights. They function like queries to be matched against vectors.
<b>match list</b>	showing subsegments similar to a profile above a minimum threshold, ranked by significance.
<b>sequence</b>	an ordered listing of subsegment IDs, grouped into runs of consecutive IDs in the same batch.
<b>attribute data</b>	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 older batch is deleted, and its number is recycled for the newest batch.

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

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

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

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

## 10. ActiveWatch Architecture

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

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

## setup

### STPBLD, SUFBLD, LITBLD

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

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

The SUFBLD module reads in text files `suffix` and `action` and writes a binary file `sufs` for AW modules to read. This controls AW morphological stemming, which will remove endings like -MENT from words like ESTABLISHMENT from n-gram indexing. but not from words like CEMENT. The suffix file contains stemming rules, one per line, and if a rule is matched, AW executes one of a list of possible actions defined in a text file `action`. The AW morphological stemming algorithm is quite complex and probably will left alone except by the most exacting of AW users. Users can refer to Appendix A of this guide to find out how to customize morphological stemming.

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

## core

### SEGMTR, INDEXR, UPDATR

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

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

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

UPDATR updates n-gram probabilities for a new batch of text subsegments and starts an empty next batch. It also computes indexing entropy to gauge how well a set of n-gram indices can discriminate between the content of different subsegments. With the current AW demonatration system, entropy is about 11.5 bits, but this can be improved with literal n-grams defined for a user's target text.

## application

### SEQNCR, SQUEZR, MLTPLR, CLUSTER, SUMRZR, CLSFYR

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

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

SQUEZR creates a compact file of index vectors for each subsegment listed 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 so will produce new clusters of no real interest. Indexing n-grams with fewer than 10 occurrences will also be ignored. Making a compact vector file will speed up the computation of similarity measure 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 write out a file of links for pairs of vectors with a scaled similarity score above a given minimum significance. The AW demonstration system operates with a minimum link significance of 8 $\sigma$  or more.

CLUSTER 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 is different from the noise model used by MLTPLR, which assumes that both sides of an inner product for vectors can vary. For profile matching, the profile weights are the same for all vectors compared against it; only the vectors vary. It sets a significance threshold for a vector match with the profile; the default is 6 standard deviations;

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

## reporting

## KEYWDR, WATCHR

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

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

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

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

## 11. ActiveWatch File Environment

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

<code>action</code>	<code>attributes</code>	<code>clusters</code>	<code>control</code>	<code>counts</code>	<code>delims</code>
<code>heads</code>	<code>index00</code>	<code>links</code>	<code>lists</code>	<code>literal</code>	<code>lits</code>
<code>maps</code>	<code>offset00</code>	<code>probs</code>	<code>profiles</code>	<code>range</code>	<code>residual</code>
<code>segm00</code>	<code>sequence</code>	<code>source00</code>	<code>stopword</code>	<code>stopts</code>	<code>stps</code>
<code>suffix</code>	<code>sufs</code>	<code>svecs</code>	<code>vector00</code>		

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

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

Copy the `stopword`, `suffix`, `action`, `literal` and `stopts` 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`, `segm00`, `vector00`, and `offset00` are associated with batch 0 of text input. which may come from multiple input text files.

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

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

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

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

SQUEZR reads in all subsegments in a sequence and keeps only the first one for each text item in a batch. These are stored in a file `svector`s, 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 `svector`s and writes out to the file `links` a record for each pair having similarity equal or greater to `s` 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 entry of one is recorded in the binary file `maps`. A separate record describing each profile is saved in the file `attributes`.

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

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

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

The AW clustering demonstration system more or less will run all the prodeessing modules above in the order presented. The actual results, however, will vary according to the choice of arguments to pass to a module in its command-line invocation.

## 12. Control of Processing in AW Modules

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

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

Module	Arguments		Comments
<b>STPBLD</b>	(none)		
<b>SUFBLD</b>	(none)		
<b>LITBLD</b>	(none)		
<b>SEGMTR</b>	-d D File1 File2 ...	use delimiter file D, defaults to <code>delims</code> first input text file; default is <code>text</code> second	can process multiple text input files specified as successive command line arguments
<b>INDEXR</b>	(none)		
<b>UPDATR</b>	Mode	= "+", "-", "+-", or "-+"; default is "+"	"+" means to add the new batch of text to statistics, "-" means to drop the old batch of text from statistics
<b>SEQNCR</b>	Count	how many items of sequence to cluster; default is ALL	used for debugging
<b>SQUEZR</b>	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
<b>MLTPLR</b>	Minimum	threshold for pairwise link; default=6 $\sigma$	
<b>CLUSTR</b>	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
<b>SUMRZR</b>	MinThr Multp MinLen	threshold for cluster assignment; default is 6 $\sigma$ minimum prob as multiple of lowest, default=10 minimum n-gram count for creating profile, default=16	null profile might be created
<b>CLSFYR</b>	Type	which profiles to apply, default='a' for ALL	
<b>KEYWDR</b>	Maximum	number of items to get keywords from; default=6	
<b>WATCHR</b>	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	

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

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

## 13. Example of AW Clustering

Here are the top 6 of the 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 instead of the default 6. The output was produced by the KEYWDR module in AW version v1.1.4. It has been truncated for brevity; typically, the Google News data results in about 80 different clusters.

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

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



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

deriving descriptive keys

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

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

-other keys-

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

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

-other keys-

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

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

-other keys-

```

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

```

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

-other keys-

```

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

```

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

-other keys-

```

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

```

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

-other keys-

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

## 14. Example of AW Residuals

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

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

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

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

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

```
scanning segments in residual file
20 segments selected from 138 candidates
 1) 0::399 0.02738252 ( 88)
 2) 0::171 0.03223459 ( 96)
 3) 0::104 0.03600021 ( 131)
 4) 0::427 0.03864775 ( 86)
 5) 0::155 0.03991351 ( 110)
 6) 0::24 0.04122145 ( 88)
 7) 0::212 0.04124255 ( 89)
 8) 0::69 0.04163282 ( 142)
 9) 0::351 0.04242392 ( 122)
10) 0::183 0.04252940 ( 107)
11) 0::60 0.04271926 ( 154)
12) 0::12 0.04324666 ( 103)
13) 0::231 0.04373187 ( 126)
14) 0::570 0.04390064 ( 127)
15) 0::283 0.04442804 ( 116)
16) 0::339 0.04506091 ( 147)
17) 0::343 0.04536681 ( 120)
18) 0::453 0.04548283 ( 122)
19) 0::70 0.04551448 ( 143)
20) 0::324 0.04652708 ( 137)
```

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 DTEXT tool described in the next section; just pass the subsegment ID as an argument.

## 15. Diagnostic Tools

AW diagnostic tools were originally developed to debug the processing of AW modules. This is still an important purpose, but the curious user can also run 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.

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

CTRL	shows the contents of the binary AW control file
DCLU	shows the cluster seed file produced by the CLUSTER module
DIXS	shows the index records identifying individual text items in a batch of text data processed by the SEGMTR module
DLNK	shows the links computed by the MLTPLR module
DLSS	shows the match lists produced by CLSFYR
DMAP	shows the allocation of profiles defined by SUMRZR and eventually also by users
DPRB m	selects to the m n-gram indices with the highest probability in a set of text data batches; if m is omitted, 16 is assumed
DNGM	shows the relative importance for indexing of user-defined literal n-grams, alphanumeric 2-grams, and alphanumeric 3-, 4-, and 5-grams.
DSCN ng	finds text segments containing an n-gram index expressed as an integer code ng
DSEQ	shows an AW sequence produced by SEQNCR
DSQV	shows squeezed vectors produced by SQUEZR
DTXT id	shows the text of an item or a subsegment given its item or subsegment id (i.e. b:n or b::n)
DVEC id	shows the index vector of a specified subsegment or the first subsegment of an item
GMx tok	shows the n-gram analysis of a given token string with no stemming
TGx < in	shows the analyses of text from standard input with stemming

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

```
java pTH path/DNGM
```

Here is sample output produced by AW v1.1.8 with the Google News data

analysis of n-gram contributions

Literal N-Grams

total prob=0.104610 for 342 indices

min=0.000011, max=0.004467

--

Alphanumeric 2-grams

total prob=0.056407 for 523 indices

min=0.000011, max=0.002169

--Alphabetic 3-grams

total prob=0.433773 for 2806 indices

min=0.000011, max=0.003662

--

Alphabetic 4-grams

total prob=0.316986 for 1539 indices

min=0.000011, max=0.004156

--

alphabetic 5-grams

total prob=0.088224 for 394 indices

min=0.000011, max=0.003286

--

This indicates that alphabetic 3-grams are pulling the most weight in indexing according to the sum of their probabilities. They are closely followed by 4-grams, then less closely by literals, 5-grams, and 2-grams, in that order. This suggests that additional 4-grams might be a good goal for built-in indexing, although we might need about a thousand more of them to make a big difference in indexing entropy.

Here is another view of n-gram probabilities running the DPRB tool to get the top 24 indices for the Google News data

```
analysis of probabilities

probability range= 0.000011 : 0.004467

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

24 indices with highest percentages of occurrence

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

The bracketed numbers are the AW codes for each n-gram; the percent values are just 100 times the probability of an n-gram index. Note the predominance of longer n-grams in the top 24; it indicates a good balance in indexing for the reduction of noise.

To see the text of any item, we can run the DTXT tool. For example, to see the text of the 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
%
```

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

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

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

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

0 : :427 In December, a team of U.S. government scientists released a "report card" on the Arctic.

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

## 16. Tuning ActiveWatch For Your Particular Text Data

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

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

There are three ways that an AW user can redirect n-gram indexing and make it more specific:

- 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 in 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 particular target text data. For details on editing AW tables, see Appendix A.

## 17. Other ActiveWatch Directions

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

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

ActiveWatch actually 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 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.

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



## 18. The Bottom Line

The inevitable question is how well ActiveWatch actually works. The answer is complicated, like explaining how it is possible for a casino to be profitable. It all depends on stacking the odds in one's own favor. For AW, this involves much more than training blackjack dealers to hold on 17, but the principle is the same. We want to maximize the separation between signal and noise in every way we can.

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

A little noise can be good, however. It allows n-gram indices to be somewhat independent, unlike whole words, which will let us to define a multinomial model of vector similarity. The vectors are what we want to focus on, not individual n-grams like RENT. AW vectors will be 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.

If you just want all messages containing the word RENT, then n-gram indexing is unnecessary. The original Java AW even included its own conventional search engine that allowed users to run whole-word searches if they needed to; this was easy to do. What is hard is the “active” part of ActiveWatch, which is what the GitHub AW demonstration is all about.

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

All of this applies of course only for English text data. If you are processing Finnish, Croatian, Swahili, Malay, Quechua, or Hawaiian, then all bets are off. It seems odd, however, that many natural language text processing systems seem to be of the opinion that their technology requires minimal turning to the lexical characteristics of any particular language. AW takes the opposite approach, and it may be more consistent with how human children learn language.

In any event, play with ActiveWatch on your own data. Theoretic discussion of AW fixed, finite indexing is fine, but the proof of the pudding is in running it with a maximum diversity of English text. It may well be that n-gram indexing needs to be developed further, especially with its recent expansion to include large numbers of alphabetic 4- and 5-grams.

The GitHub AW release is still a work in progress, but decades of experience so far with its underlying framework has given us confidence in its efficacy. It should be able to give you novel insights into your English text information resources. Although robots can now whup our tails in chess, go, and poker, humans still know more about natural language, and English in particular, than any convoluted neural net. ActiveWatch, based on such knowledge, can help.

## Appendix A. User-Defined Tables to Guide Indexing

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

So, an information tool needs to allow for easy 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 `stprefs`. This allows a user to define patterns for stopping large classes of text tokens like telephone numbers. The patterns in `stprefs` 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 are to added to the AW built-in lexical n-gram index set, typically whole words or long prefixes and suffixes like `PROTO-` or `-OLOGY`. For example,

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

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

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

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

```
mility    3 81
```

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

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

```
|men      1 118
```

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

```
1an.      =118
```

Morphological rules need not always be applied only to change the part of speech for a word root.

The AW STPBLD, SUFBLD, and LITBLD modules generate respective binary files `stps`, `sufs`, and `lits` to be loaded by other AW modules at run time to guide finite indexing. For just a basic demonstration, it is enough just to use just the `stopword`, `stps`, `sufs`, `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 generally can 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, but sometimes, a message would be so important that the officer will call a superior immediately and make copies for distribution.

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

AW is still a message processing system. Its first task is 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 scanning message text line by line, looking for where to make divisions. This will run 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 will always 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 will make a specified change in state for the pattern, signal an event, and possibly go to the next input line. Otherwise, the automaton will advance one input line without changing state.

The possible events in a message are

NIL	none to signal
S0H	start of header, which will begin a message
SBJ	subject line in header
S0T	start of text body
E0T	end of text body
E0M	end of message

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

`x y e a pat`

where

<code>x</code>	is a beginning state
<code>y</code>	is a ending state
<code>e</code>	is an event type signaled for a delimiter match
<code>a</code>	is an action (+ . ?)
<code>pat</code>	is a delimiter pattern given as a string

(1) states are integer values from 0 to 99 determining which delimiter patterns are pertinent at each state of 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 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?

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

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

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