Word Frequency Analysis As an Authoring Tool

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

As an author, I seem to have a disconcerting habit of reusing the same word or phrase over and over again. Used once, a word can be concise and incisive. But repeated use first devalues and then ultimately annoys. Especially "getting in a rut" verbs or adverbs. Such as **being** or **typically**.

I feel any word has a normally **expected use frequency**. And that going beyond that frequency in any document degrades the quality of whatever it is you are trying to say. A word like **superb** is best used only a very few times. Unexpected words such as **plethora** or **sultry** should be limited to a single use per story.

By doing some simple **word frequency analysis**, you can easily flag any problem words and alter or correct them as needed…

> **WORD FREQUENCY ANALYSIS - Counting the number of times each word in a document is used and correcting any excess.**

Secondary uses of word frequency analysis include **extracting the keywords and search terms from websites**. Or as an alternate method of spell checking that easily flags wrongly used **homonyms** and such. Or for **index generation**. Or to determine if your **reading grade level** matches your word lengths.

Word frequency analysis can easily done by using the superb **PostScript** general purpose computing language. Especially since it is so good at reading virtually any file format in any language. From most any sourcecode. The usual route is to create your **PostScript** code as a standard ASCII textfile and then send it on to **Acrobat Distiller** which you use as a **host based PostScript interpreter**.

I've written a **WORDFRQ1.PSL** PostScript utility that you might like to explore. It presently works with **.PSL** sourcecode files based on my **Gonzo Utilities**. Since I'm not sure exactly where I am going with this, the code may remain a little rough around the edges. You should be able to readily adapt this code to **Acrobat** .PDF or another display format.

Depending upon your sourcecode and its format, extracting words can end up somewhat subtle or difficult. Here are the usual steps involved…

EXPAND any document compression.

ISOLATE text strings from formatting commands.

REMOVE punctuation, linefeeds, and force lower case.

FILTER embedded commands and very short words.

COUNT the words used into a data structure.

SORT the data structure by word popularity.

ANALYZE the results for excessive word use.

Let's look at these in turn. Details will vary with your document source…

EXPAND any compression

My **Gonzo** routines and most HTML code will not be compressed, so expansion should not be needed. **Acrobat** PDF on the other hand, is usually **Flate** encoded for compactness.

Your first step to open a .PDF doc for word frequency analysis would be to get an uncompressed output. You can do this by printing to disk as level 1, by using a disability output option, by using **FLATEVUE.PDF** in my **GuruGram** library, or by using the newer and much better **UcompressPFD.api** routine on the **Acrobat SDK Software Development Kit** detailed in **STARTSDK.PDF**. Which also can be found in my **GuruGram** library.

ISOLATE text strings

Except for ordinary text files, your source doc will probably be a **mix** of printable words and embedded formatting and layout commands. Your next goal should be to extract only those "words" in your doc that are actually going to appear on screen or in print. In the case of a HTML page, chances are you can remove most of the unwanteds by eliminating anything between pairs of < and > brackets. Other sources may require wildly different techniques.

For my **Gonzo** .PSL source files and for an uncompressed **Acrobat** .PDF file, your PostScript strings will often be identified by pairs of **(** and **)** parenthesis that will define a PS **string type**.

The PostScript **token** operator can be especially useful for stripping out **only** your prinable strings. Here is an example from **WORDFRQ1.PSL** that also should be adaptable to general .PDF word extraction…

```
/extracttextstrings  { 2000        % set looping limit
     {dup length 1 gt {            % till string empty
     token                         % extract token
     /donex false store            % clear exit flag
     {dup type (stringtype) eq     % is token a string?
     {getwordsfromstring}          %    yes, process words
     {pop} ifelse }                %    no, get next token
     { /donex true store}          % set exit flag
     ifelse }                      % if token is found
     {   exit} ifelse              % if string length > 0

     donex {exit} if               % force exit if done
     } repeat                      % loop if not
               } store
```

In normal use, a string that holds the .PSL page objects gets placed on the stack and sent to this routine. Strings representing the grouped page words are then isolated and sent to **getwordsfromstring** for further processing. Similar code should be useful to extract .PDF line strings.

Tokens are extracted one at a time from the input string. If a **stringtype**, they get output, rejecting all other commnds. The **donex** flag lets you exit from a loop within a loop. Input data longer than 65K can be handled by going to a file read rather than a string input.

REMOVE punctuation, linefeeds, and embedded codes.

Two stages of processing will normally be needed to convert the word strings into individual words. In the first of these, linefeeds and carriage returns and tabs are converted to spaces. An ending space is also added. It may also be desirable to switch to all lower case characters…

To REMOVE carriage returns, linefeeds, or tabs, convert any decimal 9, 10, or 13 string ASCII values to 32. Add an extra ending 32.

To CHANGE upper to lower case, ADD 32 to any decimal string ASCII value in the range of 65 to 90.

After spaces have been substituted for other formatting, the individual words can be extracted for further second stage processing.

Perhaps like so…

```
/getwordsfromstring {
    mark exch                   % start array
    {dup dup 10 eq exch         % lf to space
    13 eq or {pop 32} if        % cr to space
    dup dup 65 ge exch          % force lower case
    90 le and {32 add} if
    } forall
    ]  makestring               % array to string

    30000 {                     % word extraction loop
    ( ) search { exch pop
    procword}{pop exit}
    ifelse} repeat
          } store
```

Individual words are then output on the stack to **procword** for further processing. As before, **makestring** is our disgustingly elegant array to string converter…

```
/makestring {dup length string dup /NullEncode filter
3 -1 roll {1 index exch write} forall pop} def
```

FILTER commands and very short words.

At this point, each printing word has been individually extracted from the page strings. In the case of my **.PSL** files using my **gonzo utilities**, these words may still have embedded font changes or url links attached to them. Our second removal stage leaves only the isolated lower case word for further processing. Using this higher level sequence…

```
/filterword {
     striptrailinggonzo          % remove url links
     striptrailingpunct          % remove trailing punctuation
     striptrailingfontchange     % remove trailing font change
     striptrailingpunct          % and again

     4{stripleadingfontchange    % remove leading font change
     } repeat

     dup length 3 gt             % word more than 3 chars?
     {gotone}{pop} ifelse        %   yes, process
          } store
```

Helped along by these code details…

```
/striptrailinggonzo {              % remove url links
      (I/) search  {exch
      pop exch pop} if
          } store

/striptrailingfontchange {         % remove post font changes
      dup length 3 ge {
      dup dup length 2 sub
      get 124 eq {dup length
      2 sub 0 exch getinterval
      } if } if } store

/striptrailingpunct {              % remove post punctuation
      dup length 3 ge {
      dup dup  dup length
      1 sub get dup 46             % periods

      eq exch 44 eq or             % commas
      dup exch 188 eq or           % ellipses
      exch 208 eq or               % em dashes

      {dup length 1 sub            % shorten by one char
      0 exch getinterval}
      if } if } store

/stripleadingfontchange {          % remove pre font changes
      dup length 3 ge {            % must be long enough
      dup 0 get 124 eq {           % vertical shash?
      dup length 2 sub             % remove first two chars
      2 exch getinterval}
      if } if } store
```

Words less than three characters are ignored. Additional filtering could be added here to block very common words. But these are also likely to be used in excess, so I've purposely left them all in.

COUNT the words used into a data structure.

At this point, each word has been isolated. We are finally ready to place the words in a data structure and count them.

A dictionary is a good initial choice because PostScript's **known** command does fast and convenient searches. We'll use a **/word nn** format where **/word** is the name of the word and **nn** is the word count.

If the word is **not** known, we add it to the dictionary with a count of one. If the word **is** known, we bump the count of the existing entry…

```
/gotone {cvn /curword              % change word to name
      exch store
      worddict curword known       % is name in worddict?

      {worddict dup curword        %    yes, bump count
      get 1 add curword exch
      put}

      {worddict begin curword      %    no, add new
      1 def end } ifelse
        } store
```

Note that **worddict** has to be previously defined with a **/worddict 100 dict def**.

SORT the data structure by word popularity.

PostScript dictionaries do not sort very well because of the hashing used for fast access. So, we will convert to a second data structure that we have used quite a few times before. This will be an array of arrays, with each subarray having two entries of **[(word) nn]**, where **nn** is the count.

Format conversion will be handled by our upcoming **reportworddict** routine. We have looked at several **PostScript** sorts in our previous **GuruGrams**. This plain old bubble sort by popularity should work just fine…

```
/popbubblesort2 { /curmat1 exch store curmat1 length
1 sub -1 1 {curmat1 0 get exch 1 exch 1 exch {/posn exch
store curmat1 posn get 2 copy 1 get exch 1 get lt {exch}
if curmat1 exch posn 1 sub exch put} for curmat1 exch
posn exch put } for curmat1 } bind store
```

ANALYZE the results for excessive word use.

A **reportworddict** routine can combine data reformatting, sorting, and set up the actual output reporting…

```
/reportworddict {mark              % start new array
      worddict  { exch /cn         % save the name
      exch store                   % save the count
      /cv exch store
```

```
mark cn 100 string              % name to posn 0
cvs cv ]                        % count to posn 1
} forall ]                      % complete array

popbubblesort2                  % sort array by count

{ dup 1 get 3 ge                % start log report
{formattedprint}{pop}           %  using formatter
ifelse} forall
  } store
```

Your **formattedprint** can go to the Distiller log file or can be written to a fancier disk file. Here is a simple log reporter…

```
/formattedprint {dup 0 get /wordx exch store 1 get /num
exch store (jn) wordx mergestr ( - ) mergestr num 10
string cvs mergestr print flush
```

Which borrows this mergestr string merger from my gonzo utilities…

```
/mergestr {2 copy length exch length add string dup dup
4 3 roll 4 index length exch putinterval 3 1 roll exch
0 exch putinterval} def
```

An Example

Our **WORDFRQ1.PSL** utility includes a **tempstring** collection of the three printing pages of **AZAUCT1.PDF** found in **GuruGram #44**. Running the utility produces these (and lesser) word frequencies in your output log file…

auction 27	sales 7	many 5	houses 4	several 3
auctions 27	while 6	some 5	surplus 4	digital 3
arizona 18	does 6	find 4	sierra 4	listings 3
their 13	have 6	also 4	based 4	community 3
ebay 12	items 6	tucson 4	help 4	good 3
with 11	other 6	school 4	resource 4	library 3
here 11	your 5	typically 4	area 3	gurugram 3
college 8	this 5	often 4	along 3	found 3
yard 8	work 5	well 4	utility 3	looked 3
these 8	that 5	estate 4	from 3	postscript 3

We immediately see that our subject matter of **Arizona Auctions** is way overloaded. But this may be tricky to reduce because most of the references needed have one or both words in them. Still, any additional uses should be ruthlessly stomped out.

there and **these** also seem to have a tad too much use. Two "in a rut" words down on the list are **based** and **typically**. And are in my usual problem area. **other** may be near the edge. While **yard** appears a bit high, it is a needed modifier to this story.

A Caution

Word frequency analysis can be a most useful tool to make your work more readable and more concise. It is also extremely useful for index generation, extracting HTML keywords, and checking your reading grade level.

Just be sure you don't go too far the other way. An author has a bad case of **Roget's Syndrome** when they use the wrong word in the wrong domicile.

Reduce any excessive word usage as best you can. But be sure to carefully avoid synonoyms for synoymns sake.

For More Help

I'm working on an upgrade that should let you work directly with output **Acrobat** PDF files. By reading input docs of any length and writing to an output reporting diskfile. This may be the subject of a future **GuruGram**. It can also be made available to you on a **Custom Consulting** basis.

Additional GuruGrams are found **here**, PostScript topics **here**, and Acrobat info can be found **here**.

Further **GuruGrams** await your ongoing support as a **Synergetics Partner**.