Text Mining Approaches with Historical Newspapers, Part 1

Alex Leslie

October 30, 2019

Contents

Welcome! In this workshop we'll be exploring one technique and one kind of dataset: fuzzy string matching in R and a batch of digitized newspapers from Chronicling America.[1]

The first step is to download the newspaper batch we'll be using from Chronicling America. The batch we'll be working with is called `allspice`; it consists of the *Perth Amboy Evening News* from March 1903 to March 1907. Once downloaded, unzip the tarball; this will take a few more minutes. The result will show up as a single directory with the name "sn85035720": this is Chronicling America's ID for the *Perth Amboy Evening News*, so we'll leave it that way. Any other single-paper Chronicling America batch will do just as well if you'd prefer to work with data from a different newspaper. Just make sure that [1] the batch includes enough page data to produce a good number of search results (`allspice` includes nearly 9,000 pages) and [2] you change all instances of "sn85035720" in this code to the ID of whatever other paper you choose.

Before we get any further into the news, though, we should familiarize ourselves with fuzzy string matching.

Levenshtein Distances and Fuzzy String Matching

The first obstacle to doing quantitative analysis on digitized newspapers - or even just finding a search term - is the messiness of newspaper Optical Character Recognition (OCR). This is a problem for all periodicals, but it's an especially big one for newspapers, many of which were printed on low-quality paper with small fonts and received minimal preservation attention. If we're going to make use of this wealth of data, we need to be able to work around persistent, inconsistent OCR errors.

Say we're trying to find a name, the surname of postbellum American author Mary Murfree, from among a *vector* of *strings* (a series of words).

```
test_vector <- c("murfree", "muurfree", "nurfree", "murfre", "murrfee", "murphee",
                 "durpree", "free", "smurffree", "murfreesboro", "marymurfree")
```

At least one of these words is perfectly correct ("murfree") and some of them are obviously not what we're looking for ("durpree"), but some of them are quite probably just OCR errors ("nurfree"). R has a conventional search function for exact *character patterns* (for our purposes, words): `grep`. Unsurprisingly, it doesn't do us much good here.

```
grep("murfree", test_vector)
```

```
## [1]  1 10 11
```

[1] My thanks to Nicole Sheriko and Andrew Goldstone for their feedback and advice on portions of the code included here.

`grep` returns a vector of the *numeric position* of each *element* in the specified vector in which our desired character pattern ("murfree") occurs. To return a vector of the strings themselves - the elements of `test_vector` and not just their numeric position - we *index* our vector of `search_hits` positions *into* our `test_vector` of strings, with brackets.

```
search_hits <- grep("murfree", test_vector)
test_vector[search_hits]
```

```
## [1] "murfree"      "murfreesboro" "marymurfree"
```

Note that `grep` also includes strings longer than our search pattern. This is basically the result we'd get if we searched for a particular word via the Chronicling America's search API. Thankfully, R has something much better for us: `agrep`.

```
search_hits <- agrep("murfree", test_vector, max.distance=2)
test_vector[search_hits]
```

```
##  [1] "murfree"      "muurfree"     "nurfree"      "murfre"
##  [5] "murrfee"      "murphee"      "durpree"      "smurffree"
##  [9] "murfreesboro" "marymurfree"
```

`agrep` allows us to get fuzzy; it returns all elements of a vector in which an approximate match to our character pattern occurs, within a specified Levenshtein Distance (`max.distance=`). Levenshtein distance is simply the number of insertions (internal added characters), deletions (deleted characters), and substitutions (characters replaced by other characters) it takes to get from one string to another (Table 1).

Table 1: A few Levenshtein Distances

String	Difference	Distance	Edit
muurfree	+u	1	Insertion
murfre	-e	1	Deletion
nurfree	m->n	1	Substitution

A Levenshtein Distance of two is clearly a bit too capacious for this test dataset, though, so let's try again:

```
search_hits <- agrep("murfree", test_vector, max.distance=1)
test_vector[search_hits]
```

```
## [1] "murfree"      "muurfree"     "nurfree"      "murfre"
## [5] "smurffree"    "murfreesboro" "marymurfree"
```

By cutting the `max.distance` down to one, we're able to filter out a few more things we didn't want ("murphee" and "durpree") - but notice we also lost one string we probably *did* want to keep, "murrfee". Our goal in setting a `max.distance` is to minimize the number of false negatives (the strings we do want that our search fails to identify) *and* false positives (the search hits that we don't actually want).

We're still getting some garbage though, for the same reason `grep` gave us additional hits: `agrep` also includes strings longer than our search pattern so long as something within distance of that pattern occurs within the string. We can solve this issue by taking the results of our last `agrep` call and filtering out anything more than one character longer than "murfree":

```
search_hits <- which(nchar(test_vector[search_hits]) < nchar("murfree")+2)
test_vector[search_hits]
```

```
## [1] "murfree"  "muurfree" "nurfree"  "murfre"
```

By adding a length limit to our search hits, we can finally get rid of some more pesky strings we didn't want. Once again, though, this comes at a cost: we've lost "marymurfree," which is certainly a false negative. If

we wanted to be even more precise, we could address this. We might run an additional `grep` call that looks specifically for the pattern "marymurfree" and combine its results with the results of our `agrep` call. Perhaps we're worried that the OCR software has a tendency to split "murfree" into separate words, like "mur" and "free"; we could write some additional code to `paste` each string together with each string immediately adjacent to it before calling `grep` and then combine those results with the results of our `agrep` call.

Our code will run slower, however, the more intricate our search gets. On a small dataset or even a large dataset and a lot of time, this might not be a problem. But even working with a few years of a single paper is a sizeable chunk of data: 9,632 pages containing around 38 million words that can take up over 540 Mb of memory. Furthermore, no matter how precise we get, at the end of the day fuzzy string matching is always a gamble: with enough data, we will always have false positives and false negatives.

Data Familiarity and Setting Up an Input Vector

Now let's turn to the *Perth Amboy Evening News*. Chronicling America datasets come highly structured; open up your downloads folder and familiarize yourself with it for a bit. You'll notice the file directory is structured something like this: "E:/Alexander/Downloads/[paper ID]/[year]/[month]/[day]/[edition]/[page]". Once you get all the way to the bottom of things, you'll see two files: a plain old .txt file and an .xml file that indicates the position of the text on the page.

The .txt file is the only one we want - and in fact, we *do not* want the .xml files, which take up tons of space. This next line of code uses wildcards to remove all files with the .xml extension within the exact number of directories specified by asterisk. Change the names or number of the directories prior to "sn85035720" to match the file path on your computer (depending on your archive utility, you may have an additional directory in there by the name of "njr_allspace_ver02"). But **do not** remove "sn85035720" (unless replacing it with the ID for a different newspaper you downloaded instead) or change anything after it: we don't want to go removing other things from your computer. If you're not sure what the exact file path should look like, just find the "sn85035720" in your file viewer, right-click (Ctrl+click on Mac), and select "Properties" to view its location.

```
unlink(Sys.glob(file.path("E:/Alexander/Downloads/sn85035720/*/*/*/*/*", "*.xml")))
```

A brief note: downloading isn't the only way to access this data. Since Chronicling America hosts a plain text version of each page online as well, we could scrape those webpages instead. If you're interested in this approach, you might consult my workshop on webscraping techniques.[2]

In order to run our search, we need to figure out the file paths for each file of data. R has a convenient function for this: `list.dirs`. `list.dirs` returns every directory in a particular path. We can narrow things down, however, and only return the directories that are x degrees removed in our file tree. We don't even need to know exact names, because `list.dirs` supports wildcard matching with `*` to designate directories. An illustration should clarify:

```
filepaths <- list.dirs(Sys.glob("E:/Alexander/Downloads/sn85035720/*/*/*/*/*"))
filepaths[1:12]
```

```
##  [1] "E:/Alexander/Downloads/sn85035720/1903/08/01/ed-2/seq-1"
##  [2] "E:/Alexander/Downloads/sn85035720/1903/08/01/ed-2/seq-2"
##  [3] "E:/Alexander/Downloads/sn85035720/1903/08/01/ed-2/seq-3"
##  [4] "E:/Alexander/Downloads/sn85035720/1903/08/01/ed-2/seq-4"
##  [5] "E:/Alexander/Downloads/sn85035720/1903/08/01/ed-2/seq-5"
##  [6] "E:/Alexander/Downloads/sn85035720/1903/08/01/ed-2/seq-6"
##  [7] "E:/Alexander/Downloads/sn85035720/1903/08/03/ed-2/seq-1"
```

[2]While scraping is a good way to obtain your own copy of the data or even to run moderate/test searches, it is much more efficient - and much kinder to Chronicling America's servers - to run code on copies of files rather than scraping each page for every query.

```
##  [8] "E:/Alexander/Downloads/sn85035720/1903/08/03/ed-2/seq-2"
##  [9] "E:/Alexander/Downloads/sn85035720/1903/08/03/ed-2/seq-3"
## [10] "E:/Alexander/Downloads/sn85035720/1903/08/03/ed-2/seq-4"
## [11] "E:/Alexander/Downloads/sn85035720/1903/08/03/ed-2/seq-5"
## [12] "E:/Alexander/Downloads/sn85035720/1903/08/03/ed-2/seq-6"
```

This vector is just about all the input R needs to read in a single page of data. Let's test it by reading in a sample page with `readLines`. Rather than specifying the exact name of the file (which we could just as easily do in this case, since we know that they're all called "ocr.txt"), we'll grab any .txt file(s) in the directory with the combination of the `Sys.glob` and `file.path` functions.

```
test_page <- readLines(Sys.glob(file.path(filepaths[1], "*.txt")))
```

```
## Warning in readLines(Sys.glob(file.path(filepaths[1], "*.txt"))):
## incomplete final line found on 'E:/Alexander/Downloads/
## sn85035720/1903/08/01/ed-2/seq-1/ocr.txt'
```

The metadata pertaining to each of the page .txt files isn't contained in the file names or even in the files themselves: we need to extract each piece of information from each file path. This can be done with the `strsplit` function by splitting wherever there is a `/` character.

```
strsplit(filepaths[1:3], "/")
```

```
## [[1]]
## [1] "E:"         "Alexander"  "Downloads"  "sn85035720" "1903"
## [6] "08"         "01"         "ed-2"       "seq-1"
##
## [[2]]
## [1] "E:"         "Alexander"  "Downloads"  "sn85035720" "1903"
## [6] "08"         "01"         "ed-2"       "seq-2"
##
## [[3]]
## [1] "E:"         "Alexander"  "Downloads"  "sn85035720" "1903"
## [6] "08"         "01"         "ed-2"       "seq-3"
```

Note that the result is a series of vectors, one corresponding to each element of the filepaths vector, rather than one big vector with everything mashed together. This data format is called a *list*. When indexing a list, we use a double pair of brackets to index the vector and an additional single pair of brackets to index the element within that vector. Since the year is the fifth element of each vector, then, we would index the year of the first file as such:

```
strsplit(filepaths[1], "/")[[1]][5]
```

```
## [1] "1903"
```

Great: now we know how to get everything we want out of a single input vector. This will make our code efficient and clean.

There's one other useful thing to be done with file paths, and that is identifying the number of pages in each issue. Since each of our filepaths is the directory corresponding to a single page, we'll need to back up one directory. We can do this by manipulating each element of filepaths with `gsub`, to substitute everything (`(^.*)`) from the beginning of each file path to the part that reads `/seq-` followed by a digit (`\\d`) at the end (`$`). These are metacharacters that, in programming, are used to build *regular expressions*: expressions that refer to inexact yet consistent character patterns. No matter which element of filepaths we use this line of code on, it'll still work because they all follow the same pattern.

```
gsub("(^.*)/seq-\\d+$", "\\1", filepaths[1])
```

```
## [1] "E:/Alexander/Downloads/sn85035720/1903/08/01/ed-2"
```

Now, we just want to know how many files are in this directory. To do so we'll use `Sys.glob` again in conjunction with `list.files`. We don't really want to know the names of the files, though, just how many of them there are. This is what the `length` function does.

```
length(list.files(Sys.glob(gsub("(^.*)/seq-\\d+$", "\\1", filepaths[1]))))
```

```
## [1] 6
```

If we were working with multiple newspapers, we would simply change two things. First - and I do mean first - we would put all the newspaper directories into one big directory to make sure we didn't mess things up with our wildcards. Second, we would add the name of that big directory to our file path above and replace the "sn85035720" above with another `*`. (The indexed positions in the function below would need to change as well to reflect whatever additional changes were made.) But aside from that, the code would essentially remain the same.

Running a Search and Grabbing Collocates

Now it's time to start putting everything together. So far we've been using functions included in R. Now we'll write one of our own, `searchfun`, that takes an element of our input vector, reads in a page, finds all approximate matches using the `agrep` code we wrote two sections ago, and records the metadata for each result using the code from the previous section.

Our code needs to be written as a function because this way we can easily run it multiple times, i.e., for each page. But it does one other thing as well: it also returns the twenty strings on either side of each hit, or collocate strings. This takes little extra time and will come in very handy later; indeed, it's one of the biggest advantages to using our own code rather than relying on a search API. In this example, I've used postbellum American author Jack London as my search.

```
searchfun <- function (filepath) {
  # the name of our function and the variable it requires to run

  one_page <- readLines(Sys.glob(file.path(filepath, "*.txt")))
  # read any .txt file(s) in the filepath into memory as `one_page`

  one_page <- unlist(strsplit(one_page, "\\W+"))
  one_page <- tolower(one_page[one_page != ""])
  # split the page into words and make everything lowercase

  forename <- agrep("jack", one_page, max.distance=1)
  forename <- forename[which(nchar(one_page[forename]) < nchar("jack")+2)]
  # here are the two lines for fuzzy string matching from earlier

  surname <- agrep("london", one_page[forename+1], max.distance=1)
  surname <- surname[which(nchar(one_page[forename[surname]+1]) < nchar("london")+2)]
  # these lines are basically the same, except they only check the strings immediately
  # following the hits identified in our previous search, `forename`

  collocates <- lapply(surname,
                       function(x) if (forename[x] < 21) {
                          paste(one_page[1:(forename[x]+22)], collapse=" ")
                       } else {
                          paste(one_page[(forename[x]-20):(forename[x]+22)], collapse=" ")
                       })
  # this is where we grab collocate strings (for ease of storage, we'll paste them all
```

```
  # into one string for now); we're indexing with `surname` because we only want to keep
  # track of instances of forename immediately followed by surname. we use `if` in order
  # to avoid getting a negative subscript (i.e., when the desired string appears at the
  # start of a page)

  tot_pg <- length(list.files(Sys.glob(gsub("(^.*)/seq-\\d+$", "\\1", filepath))))
  # this finds the total number of page files for the issue; it'll be useful later

  rm(one_page, forename, surname)
  gc()
  # this clears our page data out of memory now that we're done with it

  if (length(collocates)==0) {
    return()
    # if there are no hits, nothing will happen

    } else {
    return(data.frame(LCCN="sn85035720",
                      Year=strsplit(filepath, "/")[[1]][5],
                      Month=strsplit(filepath, "/")[[1]][6],
                      Day=strsplit(filepath, "/")[[1]][7],
                      Page=strsplit(filepath, "/")[[1]][9],
                      Issue_Length=tot_pg, Collocates=I(collocates)))}
    # if there are hits, we return the date, page, issue length, and collocates for each
}
```

Now it's time to decide who or what two-word sequence to search for. To help decide, bring up Chronicling America before running your search. In the Advanced Search tab, select New Jersey under the State field and run a quick Phrase Search for the name or phrase you're considering to see how many hits you get. Ideally, in order to balance the amount of time this will take with the amount of data it'll produce, you should aim for someone or something with around 100 hits on Chronicling America.

Once you've done that, change both instances of "jack" and "london" to the two names or words you've decided on and run the batch of code above.

In order to make sure things don't take too long, we'll wrap our search function in another function that distributes the workload across multiple processor cores.

```
decade_par <- function (input_vector) {
  core_num <- detectCores()-2
  clust <- makeCluster(core_num, outfile="")
  clusterExport(clust, varlist=c("input_vector"), envir=environment())
  clusterExport(clust, varlist=c("searchfun"))
  result <- do.call(rbind, parLapply(clust, seq_along(input_vector),
                                function(x) searchfun(input_vector[x])))
  # this is where the magic happens: it runs `searchfun` once for each element of
  # `input_vector` and then binds the results together

  stopCluster(clust)
  return(result)
}
```

Finally, let's run the actual search! This will take a couple minutes.

```
hits <- decade_par(filepaths)
```

Now if all has gone well, the output of this function, `hits`, will be a data frame in which each row contains the metadata and collocate string for each hit:

```
hits
```

```
##          LCCN Year Month Day   Page Issue_Length   Collocates
## 1  sn85035720 1903    08  03  seq-5            6 won by m....
## 2  sn85035720 1903    08  07  seq-6            8 avenne o....
## 3  sn85035720 1903    09  04  seq-6            8 â by jan....
## 4  sn85035720 1903    10  21  seq-1            6 night th....
## 5  sn85035720 1903    12  08  seq-1            6 of the b....
## 6  sn85035720 1903    12  26  seq-3            6 st featu....
## 7  sn85035720 1904    02  03  seq-2            6 mon ey e....
## 8  sn85035720 1904    06  27  seq-4            6 s piquan....
## 9  sn85035720 1904    07  29  seq-7           10 form of ....
## 10 sn85035720 1904    11  14  seq-4            6 be short....
## 11 sn85035720 1904    12  08  seq-6            8 a h lewi....
## 12 sn85035720 1905    03  28  seq-7            8 depots d....
## 13 sn85035720 1905    11  03  seq-9           16 in tint ....
## 14 sn85035720 1905    11  24 seq-15           16 december....
## 15 sn85035720 1905    11  24 seq-15           16 instance....
## 16 sn85035720 1906    02  09  seq-4           16 were yil....
## 17 sn85035720 1906    03  29  seq-8           10 oils acq....
## 18 sn85035720 1906    09  07  seq-7           16 glengarr....
## 19 sn85035720 1906    11  17  seq-4            8 claim am....
## 20 sn85035720 1906    11  17  seq-4            8 tales a ....
## 21 sn85035720 1906    11  30  seq-7           16 macgrath....
## 22 sn85035720 1907    01  04  seq-6           14 miss m c....
```

Nice and tidy data: this will make further analysis a breeze. Our work has paid off!

You may be wondering: how useful was fuzzy string matching after all? We can see how many hits an exact search would've missed by doing an exact pattern search (with `grep`) to see how many of our collocate strings contain it and subtracting this number from the total number of hits:

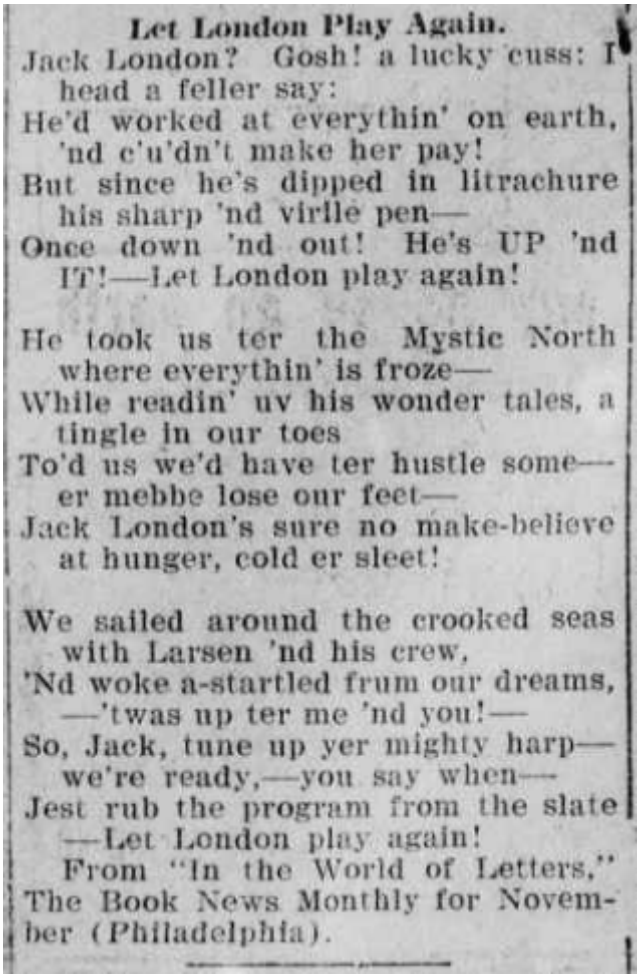
```
nrow(hits) - length(grep("jack london", hits$Collocates))
```

```
## [1] 2
```

And now the follow-up question: are these extra hits actually matches or are they false positives? Let's take a look at their collocate strings. To do so, we'll use the `-` sign for negative indexing, to return all collocate strings that don't contain the exact character pattern "jack london"; then we'll split the big collocates string (with `str_split`) into separate strings wherever there's a space (`" "`).

```
str_split(hits$Collocates[-grep("jack london", hits$Collocates)], " ")
```

```
## [[1]]
##  [1] "night"   "they"    "vrere"   "little"  "shep"    "herd"    "of"
##  [8] "kingdom" "come"    "kipliug" "s"       "five"    "nations" "and"
## [15] "the"     "call"    "of"      "the"     "wild"    "by"      "jack"
## [22] "lendon"  "ti"      "_"       "av"      "to"      "let"     "thing"
## [29] "111c"    "goâ"     "thatâ"   "s"       "wl"      "y"       "s"
## [36] "nf"      "fhp"     "many"    "colds"   "devel"   "â"       "op"
## [43] "luug"
##
## [[2]]
##  [1] "s"             "piquant"       "and"           "daring"
##  [5] "â"             "the"           "youth"         "of"
```



Let London Play Again.

Jack London? Gosh! a lucky cuss: I
head a feller say:
He'd worked at everythin' on earth,
'nd c'u'dn't make her pay!
But since he's dipped in litrachure
his sharp 'nd virile pen—
Once down 'nd out! He's UP 'nd
IT!—Let London play again!

He took us ter the Mystic North
where everythin' is froze—
While readin' uv his wonder tales, a
tingle in our toes
To'd us we'd have ter hustle some—
er mebbe lose our feet—
Jack London's sure no make-believe
at hunger, cold er sleet!

We sailed around the crooked seas
with Larsen 'nd his crew,
'Nd woke a-startled frum our dreams,
—'twas up ter me 'nd you!—
So, Jack, tune up yer mighty harp—
we're ready,—you say when—
Jest rub the program from the slate
—Let London play again!

From "In the World of Letters,"
The Book News Monthly for Novem-
ber (Philadelphia).

Figure 1: From the *Perth Amboy Evening News*, 11-17-1906, pg. 4

```
##  [9] "washing"       "ton"           "â"             "told"
## [13] "in"            "tho"           "form"          "of"
## [17] "an"            "antobio"       "grapliv"       "and"
## [21] "jaok"          "londonâ"       "s"             "vivid"
## [25] "and"           "adventurous"   "â"             "tho"
## [29] "sei"           "of"            "â"             "the"
## [33] "fiction"       "enlists"       "tin"           "illlnstrativo"
## [37] "talont"        "of"            "keller"        "sternor"
## [41] "orson"         "lo"            "well"
```

These look pretty good: four are explicitly about London's fiction (1, 3, 4, and 5), two are about London's own travel plans (6 and 7), and one particularly fun result is a poem about wanting London to write more (8) (Figure 1). Only one of these (2) is a false positive: it's a snippet called "London Tit-Bits" that just happens to be preceded by the word "back."

Seven out of eight is pretty good, especially when the collocates are as clean as they are here: this additional word data will enrich any future analysis. Our high success rate of recovering false negatives might lead us to try a slightly fuzzier search. We could, for example, change the `agrep max.distance` to 2 for "london"; this might open the garbage floodgates, but it could recover enough additional false negatives to be worthwhile. Fuzzy string matching is always a matter of trial and error.

The utility of fuzzy string matching can vary considerably based on [1] the distinctiveness of the desired character pattern, and [2] the messiness of the OCR data. The OCR for the *Perth Amboy Evening News* is comparatively clean for a newspaper - thanks to a lot of great work by Caryn Radick and the rest of the New Jersey Digital Newspaper Project team! - though still not as clean as, say, hathitrust.org scans of a magazine volume. It may be that fuzzy string matching isn't all that useful in a particular case, but you'll find that the payoff always increases as you increase the size of the newspaper dataset.

The `write.csv` function will save the results to a .csv (a bare-bones spreadsheet) for fast and easy access in the future: change the name (but not the ".csv" extension) as desired. If you're planning on coming to Part 2 of this workshop, save this file to a flashdrive or cloud drive and you'll be able to pick up right where you left off.

```
write.csv(hits, "name-this-file.csv")
```

If you'd like to look at this workshop in more detail or run the code on your own, visit https://github.com/azleslie/ChronAmQuant.

Finally, we would really appreciate it if you took a minute to fill out our brief feedback survey.

Thanks for participating!