Article extraction from historical newspaper archives

Olivier Binette + Cole Juracek, STA 490/690

"The Chronicle is Duke's independent, student-run newspaper. Founded December 19, 1905, it was called the Trinity Chronicle, as the school was called Trinity College at the time." - https://repository.duke.edu/dc/dukechronicle

Introduction

Duke has a long history of student journalism, dating back at least to 1905 when the Trinity Chronicle (now Duke Chronicle) was first published. The Duke University Library has digitized their collection which is now available at https://repository.duke.edu/dc/dukechronicle.

The undergraduate *Duke Applied Machine Learning* club has a team working on the analysis of this historical newspaper archive. While the archive is currently searchable – the newspaper scans have been read-in using optical character recognition (OCR) software – the current system does not provide information regarding the belonging of the text to individual article. The team is therefore tackling the technical challenge of *article extraction*. The goal is to identify individual newspaper articles, recovering both their title and continuous text content.

In this homework, you will work on the article extraction problem and learn to:

1. use web scrapping tools to download data (image scans) from digital archives,
2. apply optical character recognition software to read text from images, and
3. construct a rudimentary article extraction tool and evaluate its performance.

Throughout the homework, you may use the `TessTools` R package available at https://github.com/olivierbinette/TessTools. You may use the package directly or you may draw inspiration from its source code in order to solve the given tasks. The goal of this homework is to walk you through this case study, not to have you struggle with code; we will provide solutions as needed to help.

Task 1: Web scraping digital archives

This task walks you through the process of downloading images from the Duke Library website. Metadata for the Duke Chronicle digital archive has been provided to you in the file `chronicle_metadata.csv`. Each row represents one issue of the Chronicle, with the issue's webpage url under the `permanent_url` field, and the issue identifier under the `local_id` field.

Part 1. Navigate to the first issue (December 19, 1905) webpage and download the "ZIP file of JPGs" page scans.

- Notice how the "ZIP file of JPGs" is not a link to a particular address. Rather, it triggers the download by submitting an html POST request with authentifying information. (Look at the page's source code).

Part 2. Write a function which, given an issue's `permanent_url` or `local_id`, downloads the "ZIP file of JPGs" into a folder of your choice.

- To do this, use the `rvest` package to initiate an html session and submit the "ZIP file of JPGs" form. The zip file will be returned to you as part of the response to your form submission.

- You may use the function `TessTools::download_chronicle`. Make sure not to submit too many requests to the library website. Only submit one download request at a time.

Task 2: Optical Character Recognition

This task will walk you through the process of using the Tesseract tool for optical character recognition.

Part 1. Check if Tesseract is installed on your machine and available in PATH by typing `tesseract` in a terminal window. Follow the installation instructions at https://tesseract-ocr.github.io/tessdoc/Installation.html if tesseract is not installed on your machine.

- If unable to install Tesseract, let Olivier know and skip the rest of this task.

Part 2. Run the Tesseract command line tool on the JPG scans from the first issue, using the "hocr" option. In a terminal window, your call to Tesseract will have the form

```
tesseract inputfile outputfile hocr
```

where `inputfile` is the path to a JPG image and outputfile is the name of the of the result. You may also use the function `TessTools::hocr_from_zip()` or `TessTools::hocr_from_images()`.

- Tesseract's output will be a file with the "hocr" extension. This is an html file which contains the text read-in from the newspaper scan together with meta information such as layout segmentation, word bounding boxes and word confidence levels.

Part 3 (optional). Visualize one of the resulting hocr file using the `hocrjs` script available at https://github.com/kba/hocrjs. You may use the `visualize_html()` function of the `TessTools` package to help with this.

Part 4. Transform the hocr files into data frames, where each row represents one paragraph (division of class "ocr_par" in the hocr files) and with columns for the paragraph text and the paragraph's bounding box. You may use the function `TessTools::paragraph()`.

Task 3: Construct article extraction tool

This task will walk you through key steps for constructing a rudimentary article extraction tool.

The paragraphs extracted in Task 2 (part 4) for the first Chronicle issue have been annotated according to the article to which they belong. Paragraphs have also been categorized as being either a title, text or part of a publicity. You can find this data in the file "vol1_paragraphs_truth.RData".

Part 1. Determine performance evaluation metrics which your article extraction tool will try to optimize.

- In the dataset "vol1_paragraphs_truth" available in the `TessTools` package, each paragraph is assigned an "articleID" value. This articleID is a number refering to the article the pargraph is part of, or it is `NA` if the paragraph is part of no article (e.g. it is part of a publicity).

Proposed solution. For simplicity, we will focus on precision and recall, where paragraphs with `NA` articleID are ignored. Under this evaluation framework, publicity text may be assigned to articles at no penalty – we are entirely ignoring this aspect of the problem (which would require more advanced techniques).

Part 2. Use the annotated data from the first issue in order to construct an article extraction tool which targets the above evaluation metrics.

Possible solution:

- Use text capitalization and other features to identify possible titles and subtitles.
- Link paragraphs following possible titles in order to obtain article portions.
- Use a document clustering algorithm to cluster article portions into full articles.

- Tune this approach using the annotated ground truth.

Part 3. Apply your article extraction tool to the second issue of the Duke Chronicle and evaluate its performance.

Olivier's solution - modified by Cole

First let's load the annotated ground truth data.

```r
if (!require(pacman)) install.packages("pacman")
pacman::p_load(dplyr, purrr, textmineR, clevr, igraph, install=TRUE)
pacman::p_load_gh("OlivierBinette/TessTools")

vol1_paragraphs_truth = TessTools::vol1_paragraphs_truth
```

Now we identify possible titles and subtitles among paragraphs using text capitalization.

```r
title_regex = "^[A-Z0-9\\s\\.,]+$"
data = bind_rows(vol1_paragraphs_truth) %>%
  mutate(text = as.character(text),
         capital = grepl(title_regex, text, perl=TRUE))

pred_title = data$capital
```

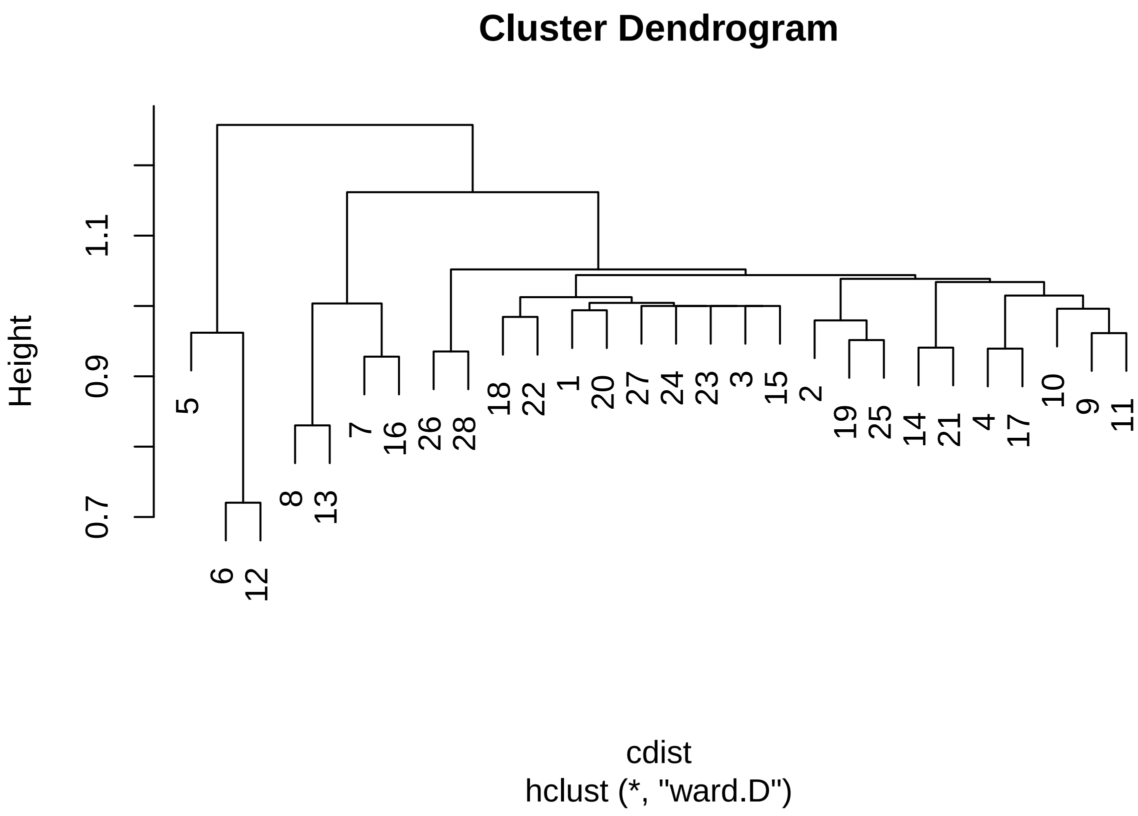
Every non-title paragraph following a title paragraph is linked to the title, thus creating article portions in the `article_parts` data frame.

```r
# Build edges between and a paragraph and the next unless it's the start of a new article
edges = t(sapply(1:(nrow(data)-1), function(row) {
  if (!pred_title[row+1]) return(c(row, row+1))
  return(c(NA, NA))
}))
edges = rbind(edges,
              matrix(c(1:nrow(data), 1:nrow(data)), nrow=nrow(data)))
edges = edges[complete.cases(edges), ]

# Determine the connected components of our article graph
g = igraph::graph_from_edgelist(edges)
articlepartID = igraph::components(g)$membership

# Group by alleged article, and then combine all same articles together
article_parts = data %>%
  mutate(articlepartID=articlepartID) %>%
  group_by(articlepartID) %>%
  summarize(text = paste0(text, collapse="\n"))
```

Now we run a generic document clustering algorithm based on "term frequency-inverse document frequency".

```r
# From the package vignette: https://cran.r-project.org/web/packages/textmineR/vignettes/b_document_clu
create_tfidf_vectors <- function(docs, doc_names) {
  dtm <- CreateDtm(doc_vec = docs, # character vector of documents
                   doc_names = doc_names, # document names
                   ngram_window = c(1, 2), # minimum and maximum n-gram length
                   stopword_vec = c(stopwords::stopwords("en"), # stopwords from tm
                                    stopwords::stopwords(source = "smart")), # this is the default valu
                   lower = TRUE, # lowercase - this is the default value
                   remove_punctuation = TRUE, # punctuation - this is the default
                   remove_numbers = TRUE, # numbers - this is the default
                   verbose = FALSE, # Turn off status bar for this demo
                   cpus = 2) # default is all available cpus on the system

  # Extract TF and IDF information
  tf_mat <- TermDocFreq(dtm)

  # Calculate tf-idf values for each token in each document
  tfidf <- sweep(dtm[ , tf_mat$term ], MARGIN = 2, tf_mat$idf, '*')
  return(tfidf)
}


calc_cosine_distance <- function(tfidf) {

  # Make each tfidf vector unit length
  vec_lengths <- apply(tfidf, 1, norm, type='2')
  tfidf_unit <- tfidf[vec_lengths > 0, ] / vec_lengths[vec_lengths > 0]

  # Dot product of 2 document unit vectors = cosine similarity
  csim <- tfidf_unit %*% t(tfidf_unit)

  # Convert cosine similarity to a distance
  cdist <- as.dist(1 - csim)
  return(cdist)
}


cluster_documents <- function(cdist, num_clusters, plot=FALSE, method='ward.D') {
  hc <- hclust(cdist, method)
  if(plot) {
    plot(hc)
  }
  clustering <- cutree(tree=hc, k=num_clusters)
  return(clustering)
}


evaluate_cluster <- function(clustering, na.rm=TRUE) {
  predID = unname(clustering)[articlepartID]
  articleID = data$articleID

  # Omit paragraphs where articleID is NA
  if(na.rm) {
    predID = predID[!is.na(articleID)]
    articleID = articleID[!is.na(articleID)]
  }

  predpairs = clevr::membership_to_pairs(predID)
  truepairs = clevr::membership_to_pairs(articleID)

  clevr::eval_report_pairs(truepairs, predpairs)
}
```

```r
tfidf <- create_tfidf_vectors(docs=article_parts$text, doc_names = article_parts$articlepartID)
cdist <- calc_cosine_distance(tfidf)
clustering <- cluster_documents(cdist, num_clusters=10, plot=TRUE)
```



```
evaluate_cluster(clustering)
```

```
## $precision
## [1] 0.7912491
##
## $recall
## [1] 0.7884669
##
## $specificity
## [1] NA
##
## $sensitivity
## [1] 0.7884669
##
## $f1score
## [1] 0.7898556
##
## $accuracy
## [1] NA
##
## $balanced_accuracy
## [1] NA
```

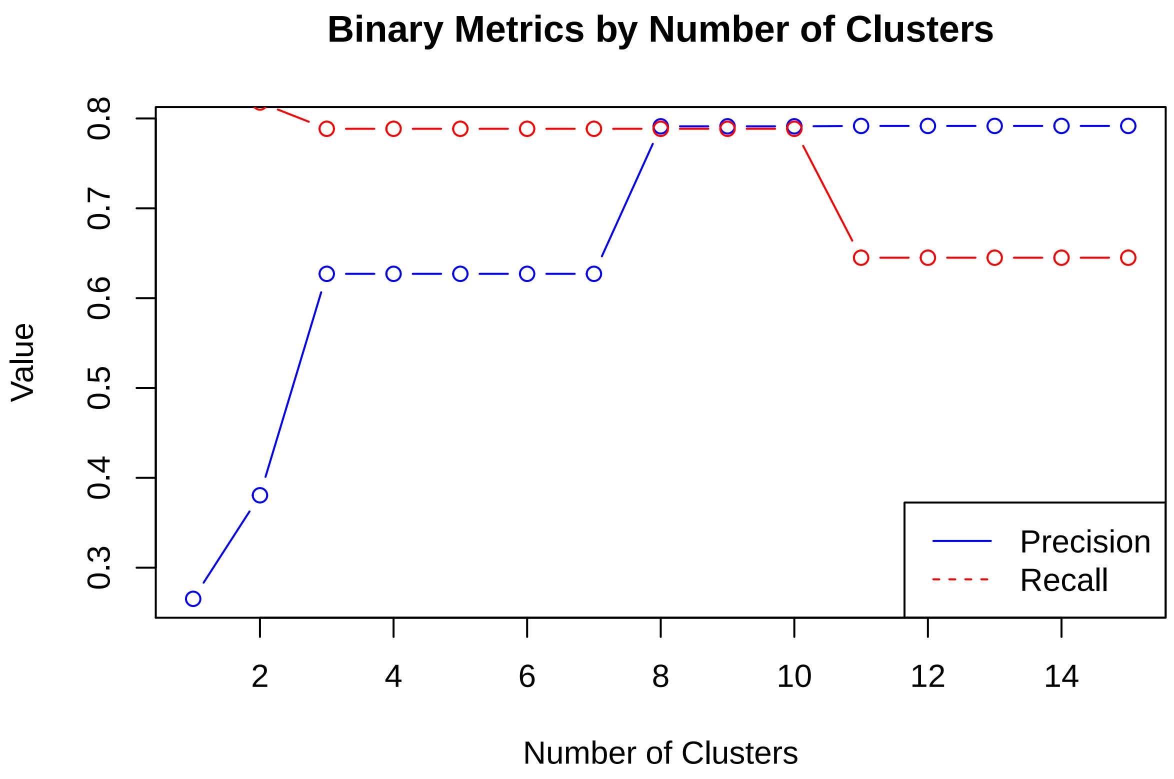
Finally we can evaluate accuracy measures.

Tuning Parameters - Number of Cluster

We want to evaluate how precision and recall change as a function of the number of clusters. Let's investigate:

```r
num_clusters <- 1:15
precision <- c()
recall <- c()
for(cluster in num_clusters) {
  clustering <- cluster_documents(cdist, num_clusters=cluster)
  results <- evaluate_cluster(clustering)
  precision <- c(precision, results$precision)
  recall <- c(recall, results$recall)
}
```

```r
plot(num_clusters, precision, type='b', col='blue', main='Binary Metrics by Number of Clusters',
     ylab='Value', xlab='Number of Clusters')
lines(num_clusters, recall, type='b', col='red')
legend('bottomright', legend=c('Precision', 'Recall'), col=c('blue', 'red'),
       lty=1:2)
```


Binary Metrics by Number of Clusters
Value
Number of Clusters
Precision
Recall