MET CS688 C1

WEB ANALYTICS AND MINING

ZLATKO VASILKOSKI

TEXT MINING 1

Discussion - Motivation

Motivation: Electronic discovery or e-discovery refers to discovery in litigation or government investigations which deals with the exchange of information in electronic format. In April 2012, a state judge in Virginia issued the first state court ruling allowing the use of predictive coding in e-discovery in the case Global Aerospace, Inc., et al. v. Landow Aviation, LP, et al. The Global Aerospace case pertained to an accident that occurred during a winter storm in 2010, in which several hangars collapsed at the Dulles Jet Center.

Tasks:

- Describe the most popular legal analysis (e-discovery) platforms available with focus on their types of legal data (digital text, OCR, audio, video) that they can deal with text mining and text analytics features of these platforms.

In particular focus your attention on these key features:

- visualization of their data analytics
- learning technology (predictive coding) to cluster conceptually related documents
- how much data they can review
- how fast is their timeline effectiveness (dealing with large data in very short timeline)
- how complex customer reviews these platforms allow for
- compare professional platform features to open source solutions

Based on all of these features what e-discovery platform would you recommend to a

- smaller size law firm
- large law firm

Provide formal arguments and citation for your response if necessary. Please have a look at other people's comments too.

Text Mining

Content:

- Preprocessing and content extraction
- Searching and fuzzy string matching
- Clustering text
- Classification, categorization, and tagging
- Question answering systems

The general approach is to represent the text analytics tasks into a mathematical ones and apply standard (100 years old) math techniques implemented as a computer algorithm.

Note: that the math notes in this class are for your reference only. They refer to the use in various algorithms and techniques used throughout the modules. It is not expected that you need to understand all of the math mentioned. It is intended to give more of an idea what is behind some of the more complicated concepts such as document matching in context space for example, if some of you have additional interest.

An illustrative example

- Consider an application that allows a user to enter query to search all the files on your computer that may contain the keyword in the text.
- The first thing you would need to do is to transform all these files into a common format so that you only have to worry about one format internally.
- This process of standardizing the many different file types to a common text-based representation is called preprocessing. Preprocessing includes any steps that must be undertaken before inputting data into the library or application for its intended use.
- As a primary example of preprocessing, we'll look at extracting content from common file formats. We'll discuss the importance of preprocessing and introduce an open source framework for extracting content and metadata from common file formats such as MS Word and Adobe PDF.

Text Mining (Text Analytics)

Most of today's data is unstructured (in a form of mixed media)

- text, images, audio data, etc.

The information in the data eventually is retrieved in a textual form – the way we communicate and express ourselves.

- The most optimal, most compact way of keeping the content of the information we use on a daily basis.

- The goal of text mining (text analytics) is to obtain (retrieve) the essential information contained in the text data by using statistical pattern learning.
  - Goal - content management and information organization
  - Tools - applying the knowledge discovery to tasks such as
    - topic detection
    - phrase extraction
    - document grouping, etc.

Text Mining (Text Analytics)

- Typical steps involved in information retrieval
  - Structuring the input text
    - Preprocessing, parsing, searching for some linguistic features and the removal of others.
  - Obtaining and matching patterns within the structured data (text).
    - Linear algebra, Machine Learning (regression, classification, clustering).
  - Evaluation and interpretation of the results.
    - Statistics (Precision, Recall, F score etc.).

Introduction

Example:

- We have a large set of digital text documents (books, newspapers, emails, etc.).
- We want to extract useful information from this massive amounts of text quickly.
- To summarize the main themes and identify those of most interest to us or to particular people (clients).
- Using automated algorithms, we can achieve this much quicker than a human could.
- To this we refer to as knowledge distillation:
  - Text classification/categorization
  - Document clustering—finding groups of similar documents
  - Information extraction
  - Summarization

Text processing

- Why Is Processing Text Important?
  - All of our digital communication, language, documents, storage of knowledge etc. is in a textual form.
  - Understanding the information content of this data part of our daily job.
  - Increase productivity, find relevant information quicker etc.

- Levels of text processing, by increasing complexity:
  - Characters
  - Words
  - Multiword text and sentences
  - Multi sentence text and paragraphs
  - Documents
  - Multi document text and corpora

What Makes Text Processing Difficult?

- Challenges at many levels, from handling character encodings to inferring meaning.
- On a small scale, much easier to deal with:
  - Search the text data based on user input and return the relevant passage (a paragraph, for example).
  - Split the passage into sentences.
  - Extract “interesting” things from the text, for example the names of people.

- On a large scale, much harder to deal with:
  - Linguistics as syntax (grammar), understanding the rules about categories of words and word groupings.
  - Language translation. Have you tried Google translate?
  - “Understanding” the text content like people do.

- We are far from the famous Turing Test—a test to determine whether a machine's intelligent behavior is distinguishable from that of a human.

Searching Through Text Data

- Simplest possible text analytics task
  - Search digital text documents for a content of a particular word.

- Typical approach
  - A linear scan (Grepping - half a century old way since 1970, UNIX, “sed”, “awk”):
    - Going through all of the text and checking for the appearance of that word by matching the exact pattern of letters contained in the query word with every single word in the text we search.

- No better way for a small set of documents.

- Nowadays impractical since the data accumulated online is so large.

Commonly Used Terms in Text Mining

- **Index** – This refers to cataloging (indexing) the documents in advance, to avoid linearly scanning the texts for each query.
- **Terms** – Terms are the indexed units (words) that we are searching for.
- **Term-Document Incidence Matrix** – This is a (binary) table that relates the terms and the documents. The transposed version is called **Document-Term Incidence Matrix.**
- **The Boolean Retrieval Model** – This is one of the basic models used to retrieve information from the term-document incidence matrix by
  - Uses term-document incidence matrix so it avoids linear search of all documents.
  - Uses binary operations which are the fastest possible.

Commonly Used Terms in Text Mining

- **Indexed Notation of a Matrix** – TDM is sparse, more memory economical (thus faster) to keep just the nonzero entries.

- **Dictionary** (**Vocabulary** or **Lexicon**) – This keeps all the unique terms.

- **Postings** (**Posting Lists**) – Relates dictionary (unique) terms with document ID (the way we numbered documents).

- **Ranked Retrieval Model** – Contrasting the Boolean model are ranked retrieval models such as the vector space model. These models use free text queries (more complicated math SVD), and the system decides which documents best satisfy the queries.

Term-Document Incidence Matrix

Documents / Terms	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	…
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
…							

- The result is a binary term-document incidence matrix, as shown above. The rows of the term-document incidence matrix are made up of terms (words), and the columns of the term-document incidence matrix are made up of documents (plays in this case).

- In a term-document incidence matrix an element (*t*, *d*) is 1 if the play (the document) in column *d* contains the word (term) in the row *t*, and is 0 otherwise.
- We can see that the document "Julius Caesar" does not contain the term "Cleopatra" thus the entry "0" in the term-document incidence matrix.

Commonly Used Terms in Text Mining

- **The Boolean Retrieval Model** – This is one of the basic models used to retrieve information from the term-document incidence matrix.
  - Uses term-document incidence matrix so it avoids linear search of all documents.
  - Uses bitwise (binary) logical operations which are the fastest possible.

Illustration:

To answer the query Brutus AND Caesar AND NOT Calpurnia, we take the vectors for Brutus, Caesar and Calpurnia, complement (negate) the binary vector for Calpurnia, and then do a bitwise AND:

(110100) AND (110111) AND (101111) = (100100)

- So this query is contained in the first and the fourth document (see previous slides).

Commonly Used Terms in Text Mining

- **The Boolean Retrieval Model** – This is one of the basic models used to retrieve information from the term-document incidence matrix.
  - Uses term-document incidence matrix so it avoids linear search of all documents.
  - Uses bitwise (binary) logical operations which are the fastest possible.

Illustration:

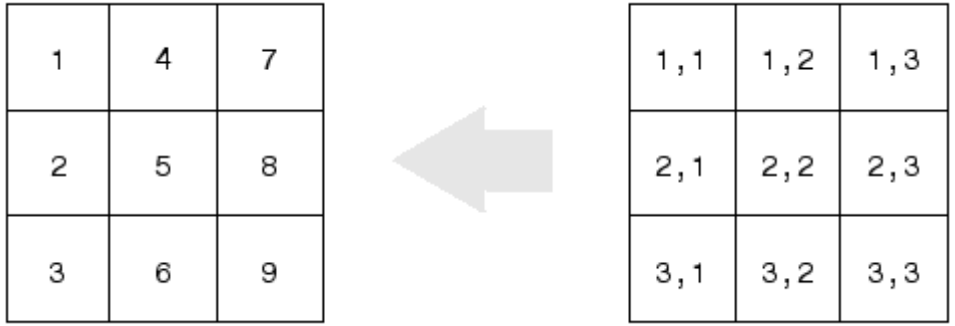
- Suppose we have N = 1 million documents (or a collection - units we use to build an information retrieval system from).
- Suppose each document is about W=1000 words long (2–3 book pages). Assuming an average of 6 bytes per word including spaces and punctuation, then this is a document collection of about 6 GB in size. Assume we have M = 500,000 distinct terms.
- This gives a matrix of zeros and ones of size M*N=5 10^{11}, which is too big to keep and analyze.
- What can we do about this?
- Note that the DTM-matrix is extremely sparse (only few nonzero entries). The maximum number of ones is W*N=10^9, which is much smaller than M*N5 10^{11} (total number of 1's and 0's). Or at least 99.8% in M*N will be 0's.
- What is the best way to deal with sparse matrices?
  - Indexing!

Commonly Used Terms in Text Mining

- **Indexed Notation of a Matrix** – TDM is sparse, more memory economical (thus faster) to keep just the nonzero entries.

Illustration:

- Better representation is to record only the position of 1's. Example of a mapping from subscript (right) to linear indexes notation (left) for a 3-by-3 matrix.



1	4	7
2	5	8
3	6	9

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

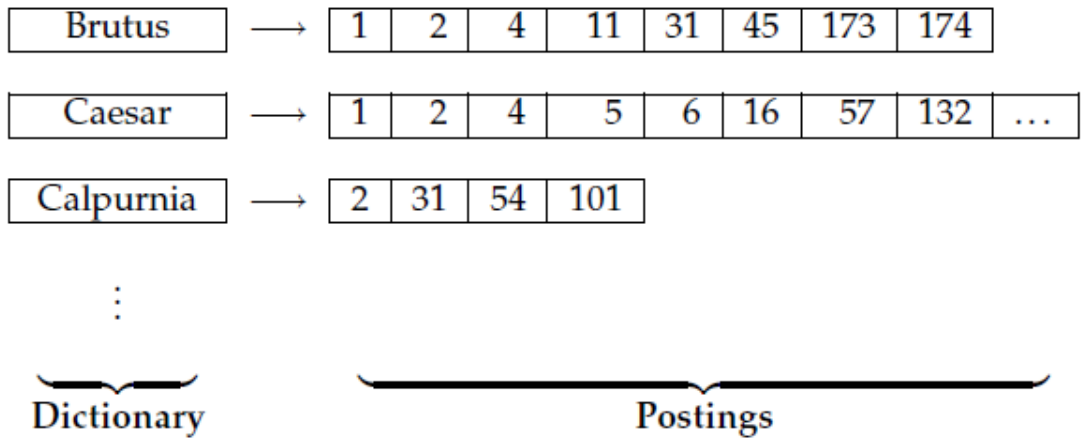
- In the case of having a large sparse matrix the advantage of the index notation is in the fact that only the indices of the nonzero elements need to be kept.
- How many indices do you need to keep track of the nonzero entries in the matrix A?

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Commonly Used Terms in Text Mining

- **Dictionary** (**Vocabulary** or **Lexicon**) – This keeps all the unique terms. For each term we keep a list (vector) in which document this term occurs, such as Brutus appears in documents 1, 2, 4....

- **Postings** (**Posting Lists**) – Relates dictionary (unique) terms with document ID (the way we numbered documents). Lists are typically sorted alphabetically and each POSTINGS LIST (matrix) is sorted by document ID. This is very useful but there are also alternatives to doing this. An example of this is illustrated below. The dictionary has been sorted alphabetically and each postings list is sorted by document ID.



Commonly Used Terms in Text Mining

Building an INDEX by sorting and grouping. There are four major steps

1. Collect the documents to be indexed:
   - Friends, Romans, countrymen. So let it be with Caesar . . .
2. Tokenize the text, turning each document into a list of tokens:
   - Friends Romans countrymen So . . .
3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms: friend roman countryman so . . .
4. Index the documents that each term occurs in by creating an index, consisting of a dictionary and postings as the index built on next slide.

Then queries are processed by

- Locating terms in the dictionary
- Retrieve its postings (vectors of document indices)
- Intersect the two postings lists

Query optimization is the process of selecting how to organize the work of answering a query so that the least total amount of work needs to be done by the system.

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

$\Longrightarrow$

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

$\Longrightarrow$

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

(Credits: "An Introduction to Information Retrieval" by Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze)

Commonly Used Terms in Text Mining

- **Ranked Retrieval Model** – Contrasting the Boolean model are ranked retrieval models such as the vector space model. These models use free text queries (more complicated math SVD), and the system decides which documents best satisfy the queries.

Commonly Used Terms in Text Mining

Typical goals in processing text are:

- Searching and matching
- Extracting information
- Grouping information
- Building QA system

Common tools for processing digital text:

- String manipulation tools.
  - Most programming languages contain libraries for doing basic operations like concatenation, splitting, substring search, and a variety of methods for comparing two strings.
- Tokens and tokenization
  - The first step after extracting content from a file is almost always to break the content up into small, usable chunks of text, called tokens. In English this is best done by occurrence of whitespace such as spaces and line breaks.
- Part of speech assignment
  - Identifying whether a word is a noun, verb, or adjective. Using part of speech can help determine what the important keywords are in a document. Commonly used to enhance the quality of results in digital text processing. There are many readily available, trainable part of speech taggers available in the open source community.
- Stemming
  - Stemming is the process of reducing a word to a root, or simpler form, which isn't necessarily a word on its own. An example is searching for the word bank to retrieve an documents on banking.
- Sentence detection
  - Computing sentence boundaries can help reduce erroneous phrase matches as well as provide a means to identify structural relationships between words and phrases and sentences to other sentences.

Commonly Used Terms in Text Mining

Text processing is also a hard problem on a small scale. Few tasks that come up time and time again in text applications:

- Search the text based on user input and return the relevant passage (a paragraph for example).
- Split the passage into sentences.
- Extract “interesting” things from the text, like the names of people and/or their email addresses for example.

Regular string matching

These operations (regular and fuzzy string matching) most commonly refer to strings from the indexed text documents.

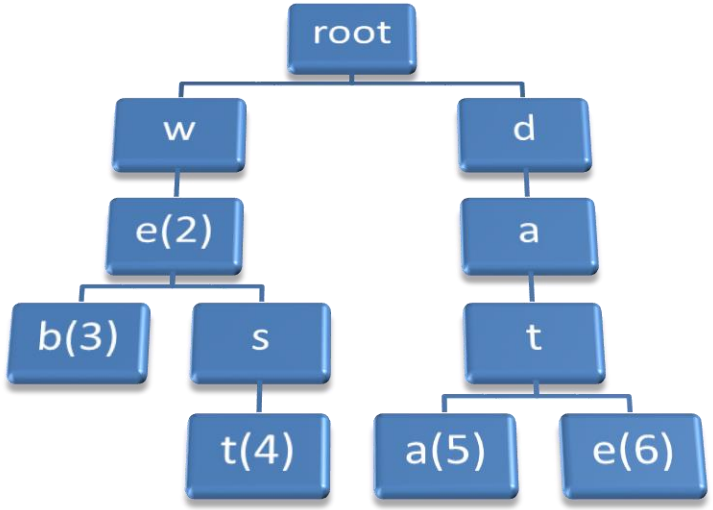
- Regular String Matching
  - Please refer to lecture notes for the common packages for regular string manipulations in R.
  - Some low level regular string manipulations include:
    - Splitting a String
    - Counting the Number of Characters in a String
    - Detecting a Pattern in a String; Detecting the Presence of a Substring

Fuzzy string matching

- Fuzzy String Matching
  - Strings comparison process – similarity between strings, such as in a spell checker.
  - Algorithms reduce to linear algebra concepts such as similarity between vectors (dot product and cosine similarity).
  - Three measures:
    - Character-overlap measures (Jaccard measure, Jaro-Winkler etc.)
    - Edit-distance measures (the minimum operations needed to transform one string into another)
    - N-gram edit distance (similar to the previous, transforms q-characters instead of letters).
- All of these are already implemented in R, you just need to be familiar with them.

Using Trie to Find Fuzzy Prefix Matches

- Used in finding prefix string matches.
  - Predictive text or auto complete dictionary, such as is found in a spell check or on a mobile telephone.
- The trie, or prefix tree, stores strings by decomposing them into characters.
- Retrieving words is done by traversing the tree structure to the node that represents the prefix being queried.
- A search ends if a node has a non null value.



Basic R Overview – Editing Strings

By default cat() concatenates vectors when writing to the text file. You can change it by adding options sep="\n" or fill=TRUE. The default encoding depends on your computer.

```
cat(text,file="file.txt",sep="\n")
writeLines(text, con = "file.txt", sep = "\n", useBytes = FALSE)
```

paste() - Strings can be also concatenated using the paste function. The function takes a variable number of arguments of any type and returns a string using space as the default separator. The separator may also be explicitly specified using the sep argument.

paste0() - Concatenates the argument strings without any separator.

substr() - The substr function is used for extracting portions of the specified string. The start and stop arguments are required. The first character is at position number 1.

sub() - The sub function is used for substituting the first matching string or regular expression (the first argument) with the specified string (the second argument). The new string is returned. The original string is not modified.

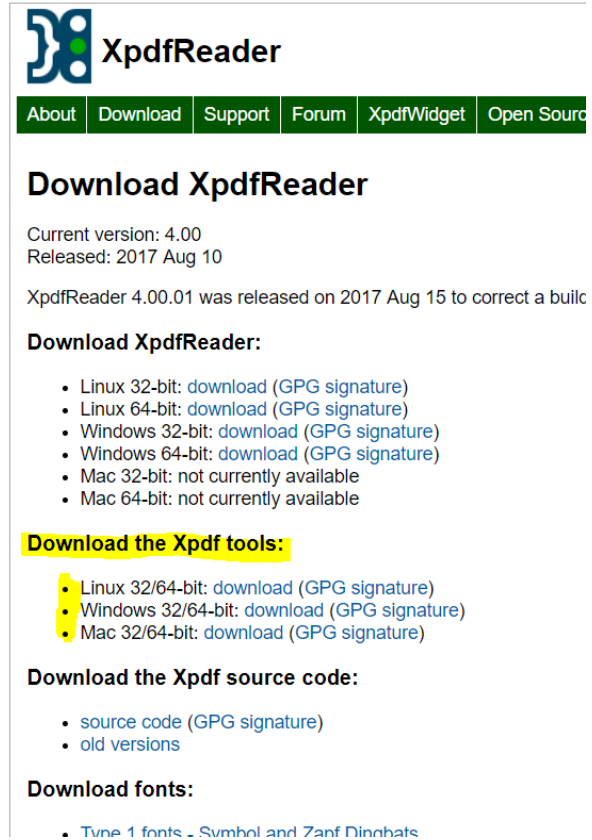
Exercise: Extracting the text content from a pdf file with Xpdf

Implement the following steps:

1. Install Xpdf Tools.
2. Test the installation on your OS by running "pdftotext.exe" in a terminal mode.
3. Run the terminal mode command "pdftotext.exe" from R using R's *system()* function.
4. Use R to extract content from several pdf files in a folder by running the terminal mode command "pdftotext.exe" in a loop using *lapply()*.

1. Extracting the content from a pdf file

- Installing Xpdf - Extracts the content from a pdf file
- You might find it helpful to use it in Text mining projects on your own.
- Install pdftotext.exe (open-source PDF viewer) part of the Xpdf software suite.
- It can be downloaded from: http://www.xpdfreader.com/download.html
- Choose the download for your operating system.
- The Windows installation is more involved so it is illustrated on the next few slides.



Additional Xpdf Utilities

These are all of the Xpdf utilities:

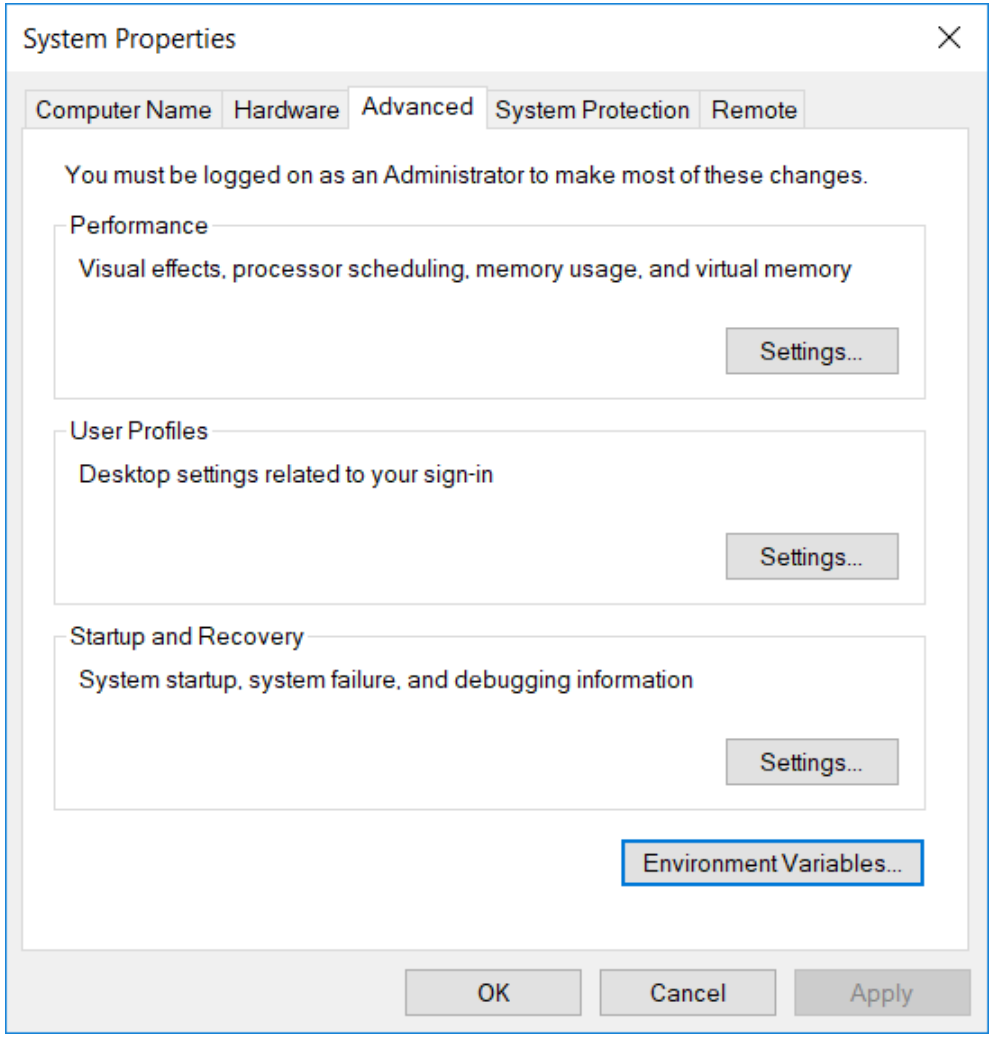
1. pdfttext -- generates a text file from a pdf file
2. pdftops -- generates a PostScript file from a pdf file
3. pdfinfo -- dumps a PDF file's Info dictionary (plus some other useful information)
4. pdffonts -- lists the fonts used in a PDF file along with various information for each font
5. pdfdetach -- lists or extracts embedded files (attachments) from a PDF (archived) file
6. pdftoppm -- converts a PDF file to a series of PPM/PGM/PBM-format bitmaps
7. pdfimages -- extracts the images from a PDF file

Installing Xpdf

- For mac users follow: http://macappstore.org/pdftotext/

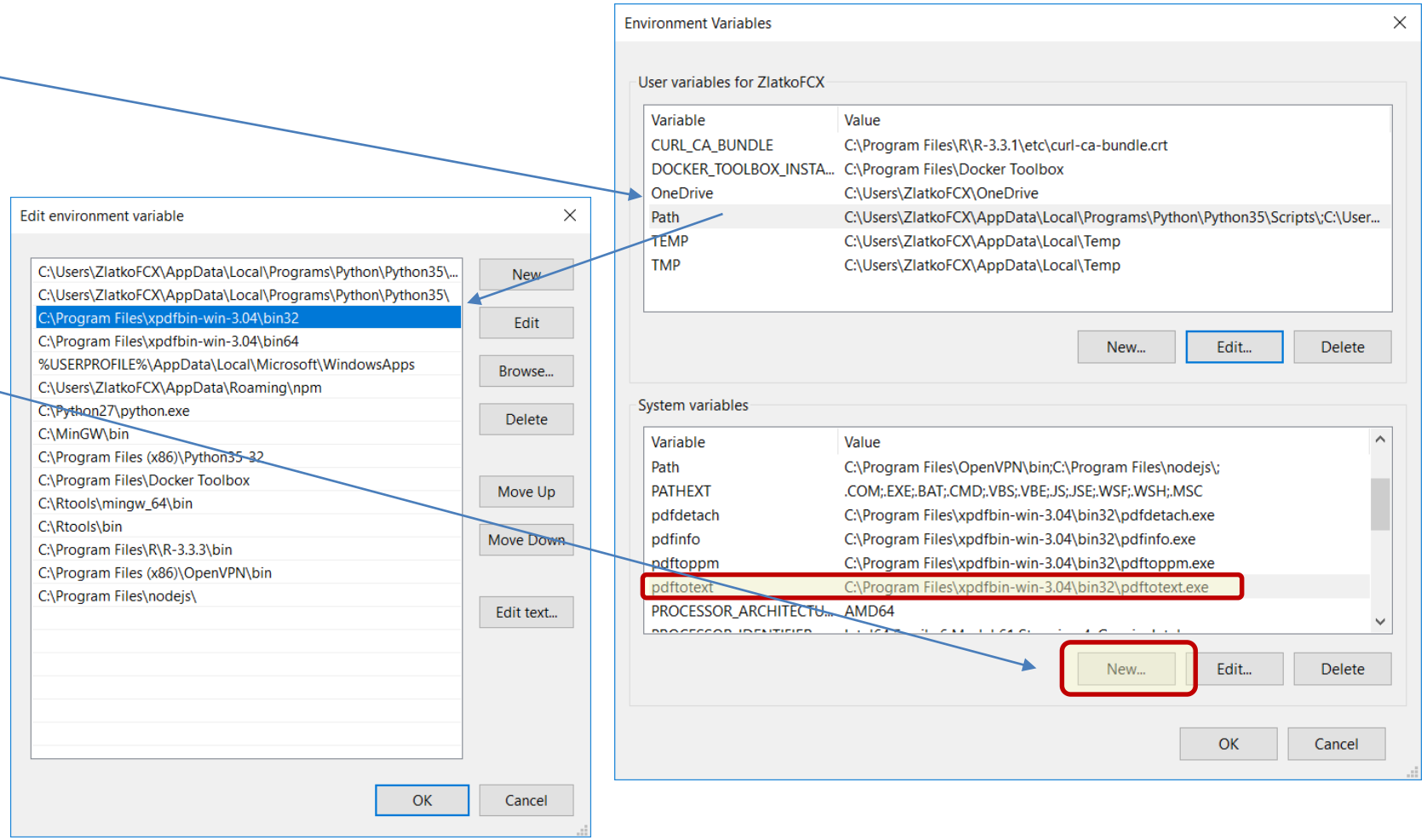
- Un-compress the downloaded files in a folder and set the environmental variables and the path to point to the folder where downloaded files are so R can find them.

- On the Windows operating system you would do that by going to **"System"**, choosing **"Advanced system settings"**, and **"Environment Variables"**.



Installing Xpdf

- Add to the “Path” user-variable the **folder** where the downloaded Xpdf files are.
- By clicking “New” add to the System the 2 new environmental variables with “Variable name” "pdfinfo" and "pdftotext".
- For the “Variable value” enter the **path** to the executables "pdfinfo.exe" and "pdftotext.exe".



2. Test the "pdftotext.exe" installation

- To test the installation, convert a pdf file in a specific folder to a text file.

- The "pdftotext.exe" conversion from pdf to text in a terminal mode is implemented at as:

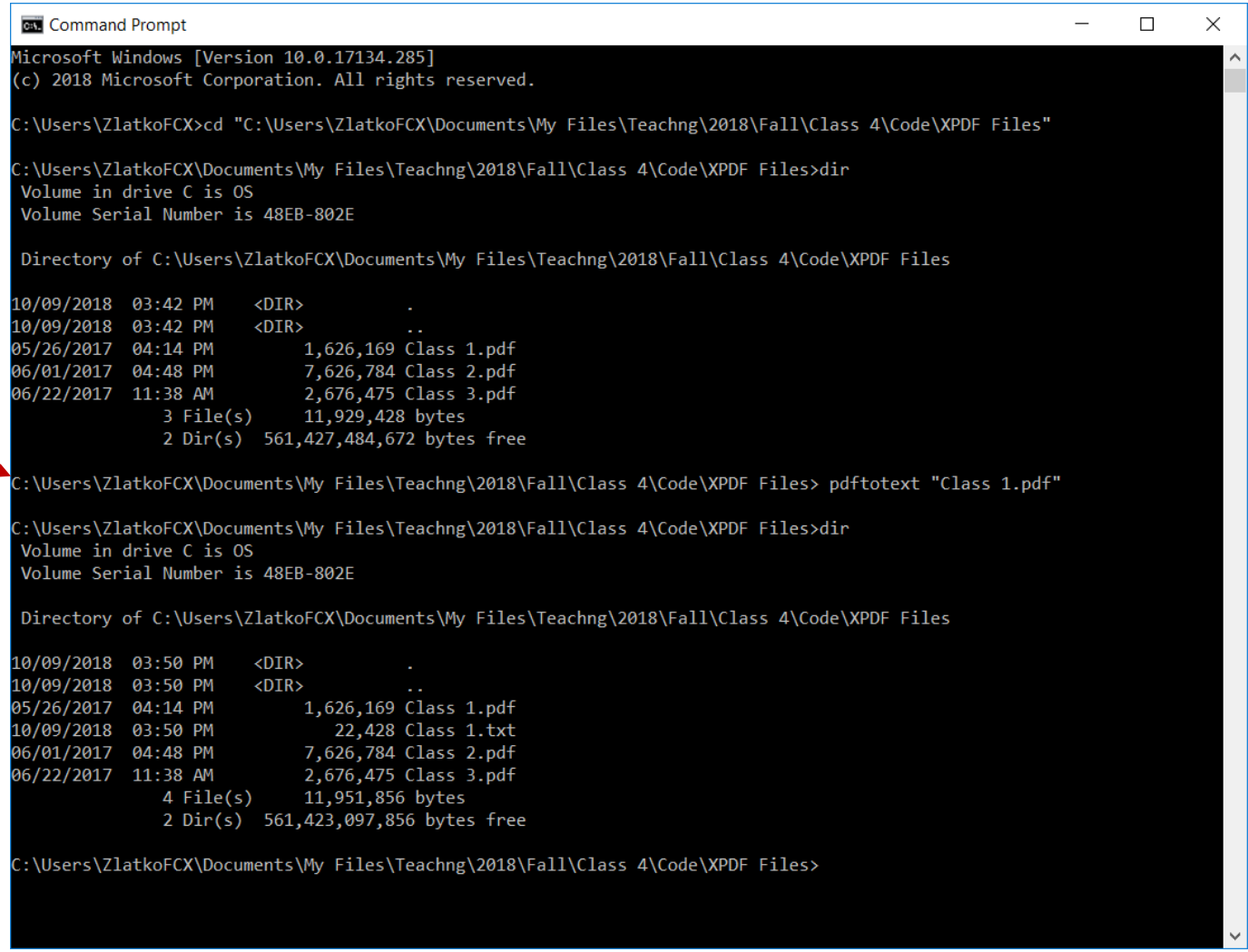
  > pdftotext "file.pdf"

- Note:
  - This usage produces a text file with the same name as the input file
  - The text file is created in the same directory as the PDFs.

Test the "pdftotext.exe" installation

- Open a terminal window (cmd on Windows).
- Either navigate to the directory where the PDF file is or include the path to it in the filename.
- List all the files in the folder using “dir”
- Type: pdftotext "Class 1.pdf"
- This produces a text file, with the same name as the pdf file, created in the same directory as the PDFs.
- List all the files in the folder again using “dir” to see it.

- Note that the file name (and the path) needs to be enclosed in quotation.
- In R we can use the function *system()* to implement a cmd command as you would implement in a terminal window.



3. Try implementing these R code examples

Download several pdf class notes from Blackboard and place them in a folder “PDF Files”, then

Task1: Use R to get the text content by implementing terminal mode command such as

```
> pdftotext "file.pdf"
```

- To accomplish this from an R script you can use R’s function *system()*
- To insert the "" around the pdf filename you need to escape them with '' . To merge, use *paste0()*. Here is the R code example:

```
system(paste(Sys.which("pdftotext"), paste0('"', myPDFfiles[1], '"')), wait=FALSE)
```

Task2: How to extend this to multiple files in a folder?

- A wildcards (*), for example *pdftotext "*pdf"*, for converting multiple files, cannot be used because pdftotext expects only one file name.

- Using R’s *lapply()* several pdf files that are contained in a single folder (specified by the R object “myPDFfiles”) can be converted with an in line function such as this,

```
> lapply(myPDFfiles, function(i) system(paste(Sys.which("pdftotext") , paste0('"', i, '"')), wait = FALSE))
```

- Note how each PDF file converted into a text file is indexed by *"i"*
- Note: Quotes (“”) in R are tricky. Make sure you properly “escape” them with single quotes such as in paste0('"', i, '"')) . **Copy/pasting the above code in R may not work. You need to type it!**

4. Getting several pdf files from a folder

Task: How to get path to **several** pdf files that are contained in a **single** folder.

```
# Example 1: Convert to text single pdf files that is contained in a single folder.
exe.loc <- Sys.which("pdftotext")  # location of "pdftotext.exe"
pdf.loc=file.path(getwd(),"PDF Files")  # folder "PDF Files" with PDFs

# Get the path (character vector) of PDF file names
myPDFfiles <- normalizePath(list.files(path = pdf.loc, pattern = "pdf",  full.names = TRUE))

# Convert single pdf file to text by placing "" around the character vector of PDF file name
system(paste(exe.loc, paste0('"', myPDFfiles[1], '")), wait=FALSE)
```

Note:

- *Sys.which* gives the path to *pdftotext.exe* (the one you set it up during installation).
- *file.path(getwd(),"PDF Files")* Gets the current folder (*getwd()*) and forms a path.
- *normalizePath()* Converts the file paths to a canonical form for the operating system.
- *list.files()* Lists the files in a Directory/Folder.
- *system()* Invoke a system command.
- myPDFfiles[1] Access the first element of vector “myPDFfiles” (contains path to the PDF file).

Using the “pdftools” package

- Another way is to use the package “pdftools”.
- To get the text content from a PDF file use

  > my.text <- pdf_text(myPDFfiles[1])

- To save text content as txt file use

  > write.table(my.text, file=paste0(pdf.loc,"/text.txt"), quote = FALSE, row.names = FALSE, col.names = FALSE, eol = " ")

- To convert and save several pdf files you can create a function as illustrated below.

```r
# Extracting the content from a pdf file
rm(list=ls()); cat("\014") # Clear Workspace and Console
library(pdftools)

pdf.loc <- file.path(getwd(),"PDF Files") # folder "PDF Files" with PDFs
myPDFfiles <- normalizePath(list.files(path = pdf.loc, pattern = "pdf",  full.names = TRUE)) # Get the path (chr-vector) of PDF file names

my.text <- pdf_text(myPDFfiles[1]) # Get the text content from the PDF file
write.table(my.text, file=paste0(pdf.loc,"/text.txt"), quote = FALSE, row.names = FALSE, col.names = FALSE, eol = " " ) # Save as txt file


# Convert to text several pdf files that are contained in a single folder.
convert.PDF <- function(myPDFfiles) {
  for (ff in 1:length(myPDFfiles)) {
    pdf.file <- myPDFfiles[ff]
    my.text <- pdf_text(pdf.file) # Get the text content from the PDF file
    File.Name <- sub(".pdf",".txt",pdf.file)
    write.table(my.text, file=File.Name, quote = FALSE, row.names = FALSE, col.names = FALSE, eol = " " ) # Save as txt file
  }
}

convert.PDF(myPDFfiles)

# Use lapply with in line function to convert each PDF file indexed by "i" into a text file
lapply(1:length(myPDFfiles),
       function(ff, myPDFfiles)
         {my.text = pdf_text(myPDFfiles[ff]); write.table(my.text, file=sub(".pdf",".txt",myPDFfiles[ff]),
                                                          quote = FALSE, row.names = FALSE, col.names = FALSE, eol = " " )},
       myPDFfiles)

```

Implementing Text Mining in R

1. **Loading/Accessing Files** (pdf, csv, txt, html, xml etc.).
2. **Extracting textual content** into electronic form (Create Corpus) – using **tm** package, specify tm **readers** and **sources**.
3. **Preprocessing** with **tm_map** - remove numbers, capitalization, common words, punctuation, and prepare your texts for analysis.
4. **Staging the Data** - create a document term matrix (**dtm**).
5. **Explore your data** - Organize terms by their frequency, export dtm to excel, clipboard etc.
6. **Analysis**
   - Analyze most frequent terms - Word Frequency
   - Plot Word Frequencies
   - Relationships Between Terms
   - Term Correlations
   - Word Clouds!
   - ML Analysis (Clustering by Term Similarity, Hierarchal Clustering, K-means clustering)

Implementing Text Mining In R

It is very unlikely that you would need to create from scratch any of the text mining tools and terms such as TDM-term document matrix.

All of the R text mining packages (such as ***tm***) are already capable of doing the necessary math and creating for example TDM.

In order to use the *tm* package in your script, you need to load it by typing:

```
> library(tm) # Load the Text Mining package
```

Follow the text mining steps in the following exercises.

Using the tm Package

The main structure for managing documents in *tm* is called Corpus (Latin for body).

- It is electronically stored texts from which we would like to perform our analysis.
- Corpora are R objects held fully in memory.
- It can represent a single document or a collection of text documents.

tm **package readers** (types of digital files):

```
> getReaders()
[1] "readDOC"    "readPDF"    "readPlain"    "readRCV1"
[5] "readRCV1asPlain"  "readReut21578XML"    "readReut21578XMLasPlain"
[8] "readTabular"   "readXML
```

tm **sources** (to get the files from):

```
>  getSources()
[1] "DataframeSource" "DirSource"  "URISource"  "VectorSource"  "XMLSource"
```

To avoid errors, make sure you use the appropriate source for your data.

- "DirSource" to get all the files in a folder, "URISource" for a specific file path.

Exercise: Content extraction from a Text File

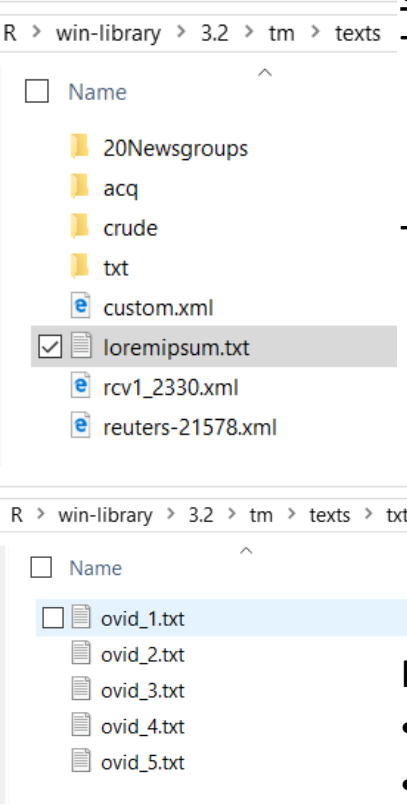
Use R and tm package to extract the content from the 2 poems that come with the “tm” package.

1. Access the 2 text files
   - Use proper tm readers and sources.

2. Create Corpus – an electronic form of the textual context from the 2 files.

3. Examine the first 5 lines of the Corpus

Exercise: Content extraction from a Text File



Example of loading a sample of text documents and creating a corpus.

The *tm* package comes with few text files (poems in Latin). Use the following:

- "loremipsum.txt"
- "txt/ovid_1.txt"

The following R script (assumes *tm* is loaded) extracts the content from these files into a corpus.

```
# Example R Script: Extracting text content from text files and load them as Corpus
loremipsum <- system.file("texts", "loremipsum.txt", package = "tm") # Path to "loremipsum.txt"
ovid <- system.file("texts", "txt", "ovid_1.txt", package = "tm") # Path to "ovid.txt"
Docs.pth <- URISource(sprintf("file://%s", c(loremipsum, ovid))) # Specify Source
corpus.txt<-VCorpus(Docs.pth) # load them as Corpus
inspect(corpus.txt)
```

Note

- *system.file* creates the path in appropriate form for R (otherwise harder to specify on Windows)
- The way *URISource* was used to create the path (syntax) to the 2 document files understandable to *tm*.

You can examine the first 5 lines of the "ovid.txt" corpus (second entry) with

```
> corpus.txt[[2]]$content[1:5]
```

Exercise: **Try to follow the notes and extract the content from a PDF file.**

Common tools for digital text processing

- String manipulation tools.
  - Most programming languages contain libraries for doing basic operations like concatenation, splitting, substring search, and a variety of methods for comparing two strings.

- Tokens and tokenization
  - The first step after extracting content from a file is almost always to break the content up into small, usable chunks of text, called tokens. In English this is best done by occurrence of whitespace such as spaces and line breaks.
- Stemming
  - Stemming is the process of reducing a word to a root, or simpler form, which isn't necessarily a word on its own. An example is searching for the word bank to retrieve an documents on banking.

- Sentence detection
  - Computing sentence boundaries can help reduce erroneous phrase matches as well as provide a means to identify structural relationships between words and phrases and sentences to other sentences.

Step 3. Preprocessing – *tm* implementation

Once we have the corpora we need to perform some pre-processing transformations such as:

- converting the text to lower case
- removing numbers and punctuation
- removing stop words
- stemming and identifying synonyms

- The basic transforms available within tm package can be seen with:

```
> getTransformations()
[1] "removeNumbers"   "removePunctuation" "removeWords"  "stemDocument"
[5] "stripWhitespace"
```

- These transformations are applied with the function "*tm_map()*".
- **Custom transformations** can be implemented by creating an *R* function wrapped within *content_transformer()* and then applying it to the corpus by passing it to *tm_map()*.

Exercise

- Perform custom transformation on a Corpus using *content_transformer()* and tm_*map()*.
- Consider the following email text:

  "Jon_Smith@bu.edu",
  "Subject: Speaker's Info",
  "I hope you enjoyed the lectures.",
  "Here is address of the lecturer: ",
  "123 Main St. #25","Boston MA 02155"

- Transforms the "@" into " “, then implement multi character transformation such as "@" and "#" into a into " ".
- Use *tm_map()* to remove numbers, punctuation, stop words, stemming …
- Replace words "Jon_Smith“ with "Jane_Doe“.

Custom Transformation - Illustration

- Consider the following R script (*tm* loaded) that transforms the "@" into " ".

```
# Example: content_transformer()
email.texts <- c("Jon_Smith@bu.edu",
                 "Subject: Speaker's Info",
                 "I hope you enjoyed the lectures.",
                 "Here is address of the lecturer: ", "123 Main St. #25","Boston MA 02155")
# Step 1: Create corpus
corpus.txt <- Corpus(DataframeSource(data.frame(email.texts)))
# Step 2: "content transformer()" crate a function to achieve a custom transformation
transform.chr <- content_transformer(function(x, pattern) gsub(pattern, " ", x))
docs.tranf <- tm_map(corpus.txt, transform.chr, "@")
```

- Note
  - The data is in a DataFrame format, so *tm*'s DataframeSource is used to create the corpus.
  - How the function "*transform.chr*" is created. It transforms the character that is passed to it.
  - > ?gsub will give you more info on R's standard pattern matching and replacement capabilities.
  - How the *content_transformer()* is implemented on the Corpus using *tm_map()* .
- As a practice try more examples on the next slieds.

1. Transforms using *content_transformer()*

The first line of the corpus text contains an email address.

```
> corpus.txt[[1]]
> Jon_Smith@bu.edu
```

The character "@" got transformed the into " "

```
> docs.tranf[[1]]
> Jon_Smith bu.edu
```

by the use of the function "transform.chr" that transforms the character that is passed to it into a space (" ").

To implement multi character transformation such as "@" and "#" into a " ", we can pass them to "transform.chr" by separating them with logical OR "|", such as:

```
> docs.tranf <- tm_map(corpus.txt, transform.chr, "@|#")
```

We can see that this transformed lines 1 and 4 of *email.texts* object where the characters "@" and "#" appear.

2. Transforms using *tm_map()*

Numbers may or may not be relevant to given analyses. This transform can remove numbers simply by

```
> docs.tranf <- tm_map(corpus.txt, removeNumbers)
```

In a similar fashion the punctuation can be removed by

```
> docs.tranf <- tm_map(corpus.txt, removePunctuation))
```

Stop words are common words found in a language. Words like "for"," very", "and", "of", "are", etc, are common stop words in the English language. We can list them (the ones provided by tm) with:

```
> stopwords("english")
```

They can be removed with

```
> docs.tranf <- tm_map(corpus.txt, removeWords, stopwords("english")))
```

In addition we can remove user defined stop words (such as " address " and "MA") with:

```
> docs <- tm_map(docs, removeWords, c("address", "MA"))
```

3. Replace words

Specific Transformations

Sometimes we would like to perform a more specific transformations such as replacing words. This is illustrated in the code below.

```
# Example 4 : Replacing a word with another one
transform.wors <- content_transformer(function(x, from, to) gsub(from, to, x))
docs.tranf <- tm_map(corpus.txt, transform.wordsd, "Jon_Smith", "Jane_Doe")
```

Stemming

- Stemming refers to an algorithm that removes common word endings for English words, such as \es", \ed" and \'s".

- The functionality for stemming is provided by wordStem() from the library SnowballC that you would need to install if you don't have it.

```
> library(SnowballC)
```

- By implementing stemming on the corpus of our previous example

```
> docs.tranf <- tm_map(corpus.txt, stemDocument)
```

we can see that the common word endings of certain words such as "Speaker's" and "enjoyed" were removed.