MET CS688

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 (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, the meaning (the semantics) 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, classification, 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.).

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.

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.

Word embeddings

Word embedding - Representing text as numbers

- Algorithms (or Machine learning models) cannot operate with text, instead they must use vectors (arrays of numbers) as input.
- When working with text, the first thing that needs to be done is to come up with a strategy to convert strings to numbers (or to "vectorize" the text) before feeding it to an algorithm or a ML model.
- This vectorization is called word/sentence/text embedding.

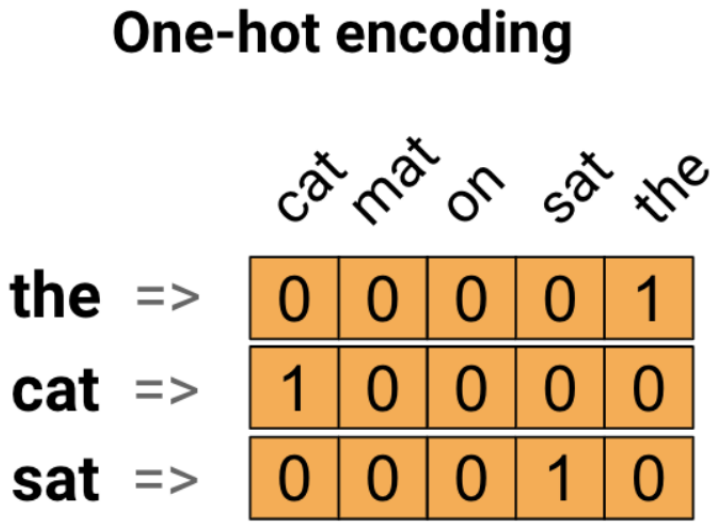
These three embedding strategies are most common.

1. One-hot encodings
2. Encode each word with a unique number
3. Word embeddings

1) One-hot encodings

The simplest approach is to "one-hot" encode each word in our vocabulary.

- Consider the sentence
  - "*The cat sat on the mat*".



- The vocabulary (or unique words) in this sentence is
  - cat,
  - mat,
  - on,
  - sat,
  - the
- To represent each word, we can create a zero vector with length equal to the vocabulary, then place a one in the index that corresponds to the word, as shown in the figure above.
- To create a vector that contains the encoding of the sentence, you could then concatenate the one-hot vectors for each word.
- This approach is inefficient. A one-hot encoded vector is sparse (meaning, most indices are zero).
  - To one-hot encode each word in the vocabulary, the created vector would have 99.99% of the elements as zero.

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).

How efficient is this sparse encoding?

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!

2) Dense Encoding

Another possible approach is to encode each word with a unique number.

- Using the previous sentence example,
  - "*The cat sat on the mat*".
- We could assign 1 to "cat", 2 to "mat", and so on. We could then encode the sentence as a dense vector (with no zero elements)
  - "*The cat sat on the mat*".
  - [5, 1, 4, 3, 5, 2].
- This approach is efficient and instead of a **sparse** vector (**with** zero elements), we now have a **dense** vector (**with no** zero elements).

There are two downsides to this approach, however:

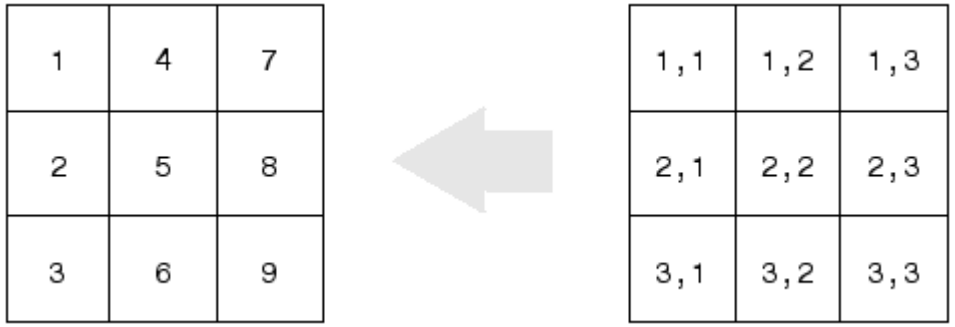
- The integer-encoding is arbitrary (it does not capture any relationship (semantics) between words).
- An integer-encoding can be challenging for a model to interpret.
- A linear classifier, for example, learns a single weight for each feature. Because there is no relationship between the similarity of any two words and the similarity of their encodings, this feature-weight combination is not meaningful.

2) Dense Encoding

- **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.



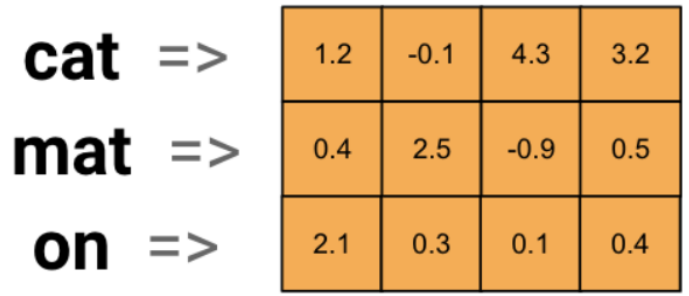
- In the case of having a large sparse matrix the advantage of the index notation is in the fact that only the indices of the nonzero elements need to be kept.
- How many indices do you need to keep track of the nonzero entries in the matrix A?

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

3) Word Embeddings

The goal of word embeddings is to give us a way to use an efficient, dense representation in which **similar words have a similar encoding**. Importantly, you do not have to specify this encoding by hand. An embedding is a dense vector of floating-point values (the length of the vector is a parameter you specify).

A 4-dimensional embedding



Instead of specifying the values for the embedding manually, they are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer).

It is common to see word embeddings that are 8-dimensional (for small datasets), up to 1024-dimensions when working with large datasets. A higher dimensional embedding can capture fine-grained relationships between words but it takes more data to learn.

Another way to think of an embedding is as "lookup table". After these weights have been learned, you can encode each word by looking up the dense vector it corresponds to in the table.

Commonly Used Steps 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.

Regular string matching

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

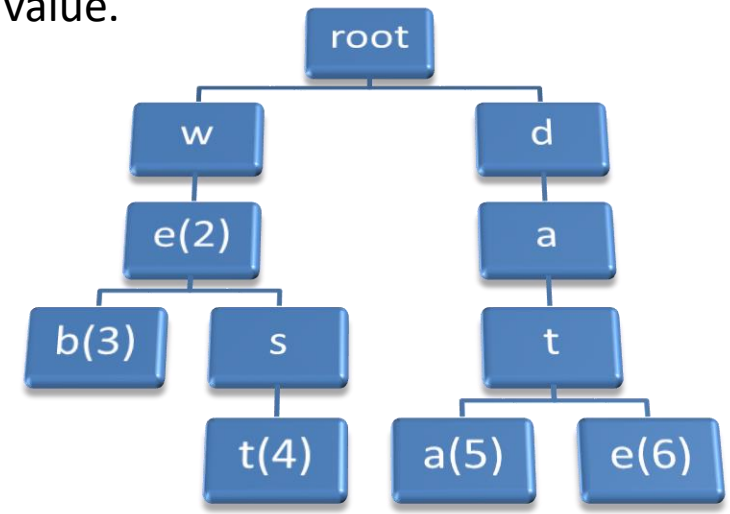
- Regular String Matching
  - Please refer to lecture notes for the common packages for regular string manipulations in R.
  - Some low-level regular string manipulations include:
    - Splitting a String
    - Counting the Number of Characters in a String
    - Detecting or extracting a Pattern in a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

Fuzzy string matching

- Fuzzy String Matching
  - String 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**).
    - **cosine similarity** is used a lot these days in text mining and NLP, so refresh your knowledge of it.
  - 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 data structure (from the word re***trie***val), or prefix tree, stores strings by decomposing them into characters.
- The complexity $O(n^{\alpha})$ (polynomial) of an algorithm is a function of the size of the input n (in bits).
- Consider: M maximum string length, N is number of encoded words (keys).
  - For tree, the search time is proportional to $M*\log_2(N)$ - logarithmic in time .
  - For trie, the search time is proportional to O(M) - linear in time. However, the trie has larger storage (memory) complexity (requirements).
- 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.



See Text Mining Exercises

- For more insight into extraction of text from different file formats.

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.

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)

Text Mining Packages in R

We will use package "*tm*" but there are several other text mining packages such as:

Package "*textclean*"

- Tools to clean and process text
- Can easily handle emoticons in text which is not an easy task for the other text mining packages
- replace_emoticon() function replaces emoticons with word equivalents

```
> x
 [1] "text from: http://www.webopedia.com/quick_ref/textmessageabbreviations_02.asp"
 [2] "... understanding what different characters used in smiley faces mean:"
 [3] "The close bracket represents a sideways smile )"
 [4] "Add in the colon and you have sideways eyes :"
 [5] "Put them together to make a smiley face :)"
 [6] "Use the dash - to add a nose :-)"
 [7] "Change the colon to a semi-colon ; and you have a winking face ;) with a nose ;-)"
 [8] "Put a zero 0 (halo) on top and now you have a winking, smiling angel 0;) with a nose 0;-)"
 [9] "Use the letter 8 in place of the colon for sunglasses 8-)"
[10] "Use the open bracket ( to turn the smile into a frown :-("
>
>
> replace_emoticon(x)
 [1] "text from: http skeptical /www.webopedia.com/quick_ref/textmessageabbreviations_02.asp"
 [2] "... understanding what different characters used in smiley faces mean:"
 [3] "The close bracket represents a sideways smile )"
 [4] "Add in the colon and you have sideways eyes :"
 [5] "Put them together to make a smiley face smiley "
 [6] "Use the dash - to add a nose smiley "
 [7] "Change the colon to a semi-colon ; and you have a winking face wink with a nose wink "
 [8] "Put a zero 0 (halo) on top and now you have a winking, smiling angel 0 wink with a nose 0 wink "
 [9] "Use the letter 8 in place of the colon for sunglasses smiley "
[10] "Use the open bracket ( to turn the smile into a frown frown "
>
```

Other Text Mining Packages

Package "*textclean*"

- replace_grade() function Replace Letter Grade with Words
- replace_html() function Recognizes html tags (such as for € euro) and replaces them with words

```
> text
[1] "Moe got an A+ grade"                          "Joe deserves an F grade for his quiz"
[3] "It's C+ work"                                 "A poor perdormance on the exam is graded as C!"
> replace_grade(text)
[1] "Moe got an very excellent excellent grade"          "Joe deserves an very very bad grade for his quiz"
[3] "It's slightly above average work"                   "A poor perdormance on the exam is graded as average!"
>
```

```
> x
[1] "<bold>Random</bold> text with symbols:   &lt; &gt; &amp; &quot; &apos;"
[2] "<p>More text</p> &cent; &pound; &yen; &euro; &copy; &reg;"
> replace_html(x)
[1] " Random  text with symbols:   < > & \" '"  " More text  cents pounds yen euro (c) (r)"
>
```

html	symbol
©	(c)
®	(r)
™	tm
“	"
”	"
‘	'
’	'
•	-
·	-
⋅	[]
–	-
—	-
¢	cents
£	pounds
€	euro
≠	!=
½	half
¼	quarter
¾	three fourths
°	degrees
←	<-
→	->
…	...
<	<
>	>
&	&
"	"
'	'
¥	yen

Other Text Mining Packages

Package “*quanteda*” (https://tutorials.quanteda.io/introduction/install/)

- quantitative text analysis
- runs solely on base R
- recommend to install the latest version of RStudio

Package “*readtext*”

- to read in different types of text data

Package "*spacyr*" () Access the powerful functionality of spaCy

- Allowing R to harness the power of Python
- spacyr opens a connection by being initialized within your R session
- Necessary for preparing text for deep learning

Package “*text*” (https://www.r-text.org/)

- Embed text using Bert
- Multilingual language models

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

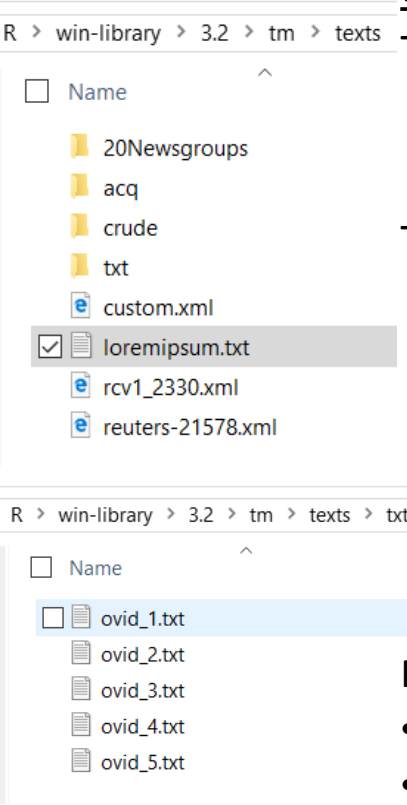
Use R and tm package to extract the content from the 2 poems that come with the “tm” package.

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 <- VCorpus(VectorSource(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.