
# MET CS688

# WEB ANALYTICS AND MINING

ZLATKO VASILKOSKI

## TEXT MINING

# 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, speech in audio and video 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 NLP tasks such as
    - Topic detection
    - Phrase extraction
    - Document grouping, classification
    - Document summarization , etc.

# 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

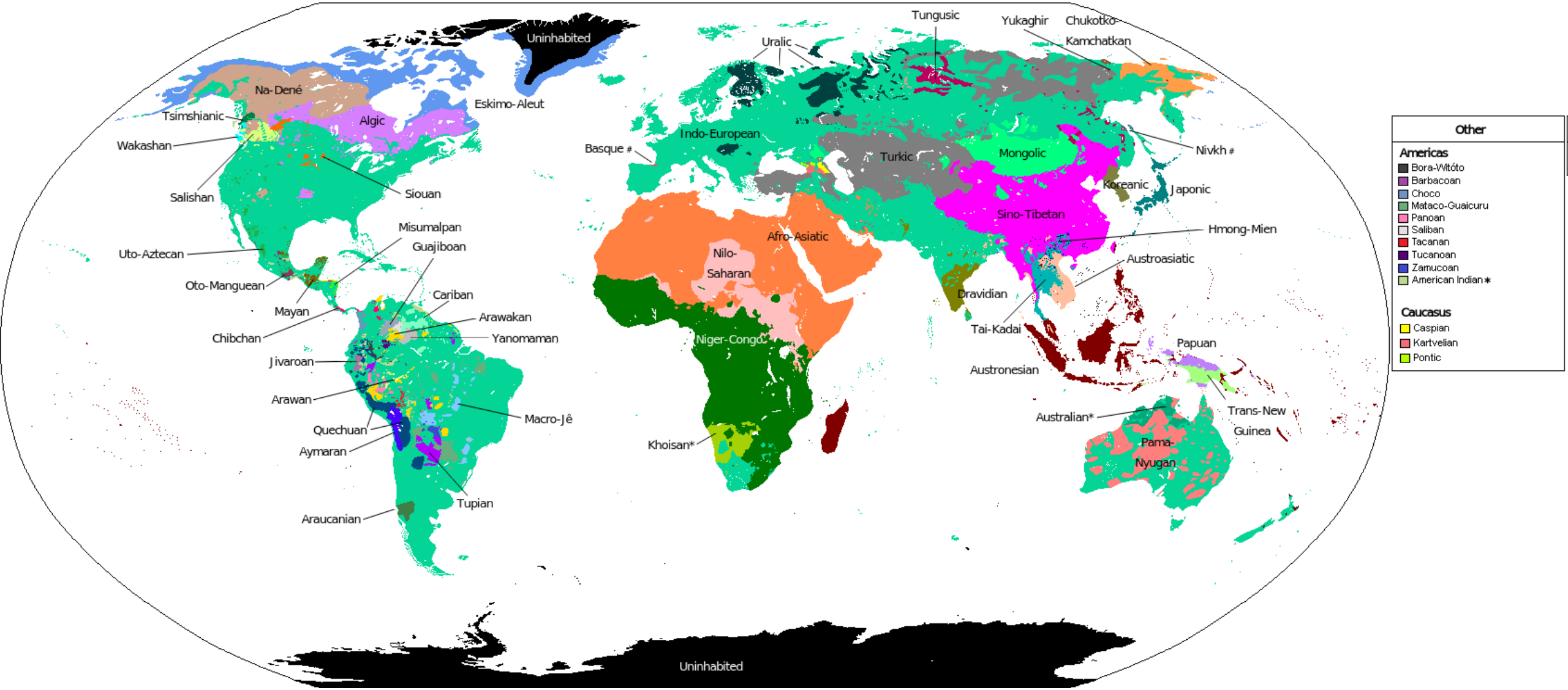
# What Makes Text Processing Difficult?

- 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.

# Language Families Today

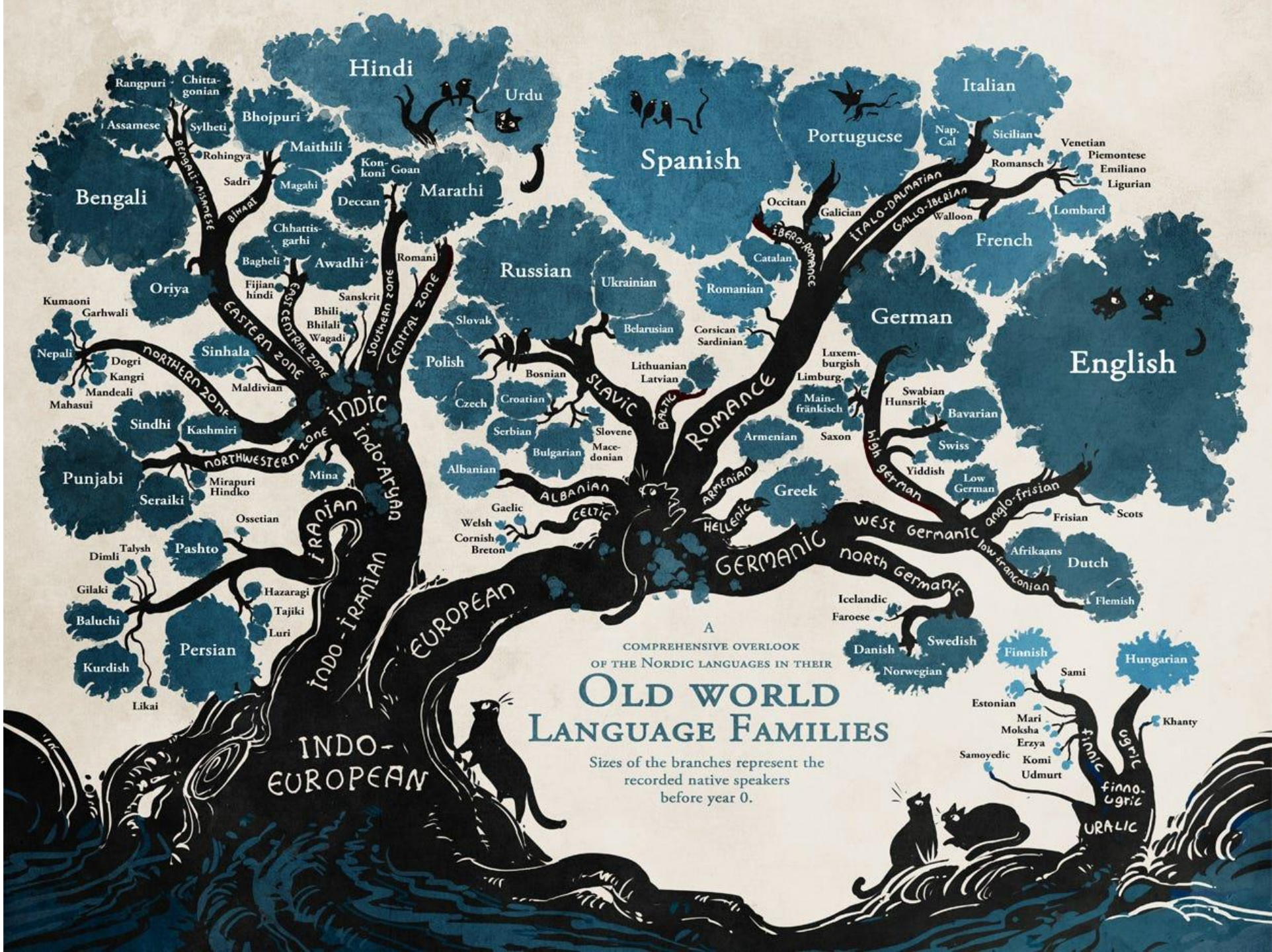


# Indo-European languages

In total, **46%** of the world's population (**3.2 billion people**) speaks an Indo-European language as a first language—by far the highest of any language family.

All of the languages illustrated here stem from subcategories of either **Indo-European** or **Uralic** origin, and upon closer inspection many fascinating links are revealed.

There are about **445** living Indo-European languages with over **two-thirds** (**313**) of them belonging to the **Indo-Iranian** branch.



# Indo-European languages

A language family native to the overwhelming majority of **Europe**, the **Iranian** plateau, and the northern **Indian** subcontinent.

**Origin**: 6,400–3,500 BC, around the Black and Caspian seas.

The **Indo-European** family is divided into several branches or sub-families, of which there are **8 groups** with languages still alive today:

- Albanian, Armenian, Balto-Slavic, Celtic, Germanic, Hellenic, Indo-Iranian, and Italic/Roman
- Another 9 subdivisions that are now extinct.

These European Languages have expanded across several continents through colonialism:

- English, French, Portuguese, Dutch, and Spanish.

| | Sanskrit | Ancient Greek | Latin | German | English | French | Russian |
|---|---|---|---|---|---|---|---|
| 1 | eka | en | unus | eins | one | un | odyn |
| 2 | dva | duo | duo | zwei | two | deux | dva |
| 3 | tri | tri | tres | drei | three | trois | tri |
| 4 | catur | tetra | quatuor | vier | four | quatre | chetyre |
| 5 | panca | pente | quinque | funf | five | cinq | piat |
| 6 | sas | hex | sex | sechs | six | six | shest |
| 7 | sapta | hepta | septum | sieben | seven | sept | sem |
| 8 | asta | octo | octo | acht | eight | huit | vosem |
| 9 | nava | ennea | novem | neum | nine | neuf | deviat |
| 10 | daca | deca | decem | zehn | ten | dix | desiat |

Table 1. Similarity of the words for number in the Indo-European languages. (Tobias Dantzig, 2007)

Present-day distribution of Indo-European languages

# Human Languages

**Source:** *The Washington Post, April 23, 2015.*

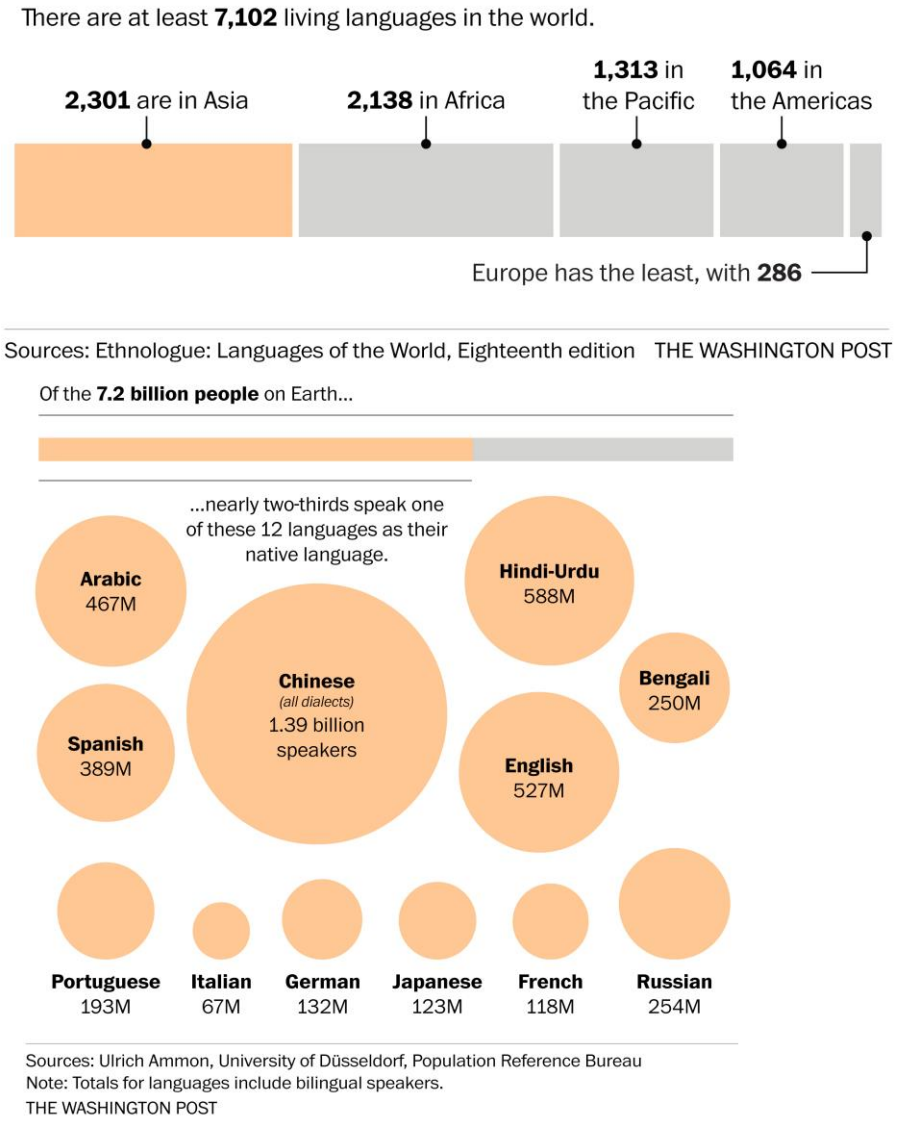
Link: https://www.washingtonpost.com/news/worldviews/wp/2015/04/23/the-worlds-languages-in-7-maps-and-charts/

- About **3%** of the world's population accounts for **96%** of all languages spoken today!
- Out of **all** languages in the world, **2,000** have fewer than **1,000** native speakers.

Some continents have more languages than others.

- **Asia** leads the statistics with **2,301** languages, **Africa** follows closely with **2,138**.
- There are about **1,300** languages in the **Pacific**, and **1,064** in **South and North America**. **Europe**, despite its many nation-states, is at the bottom of the pack with just **286**.
    1. **Chinese** has more native speakers than any other language,
    2. It is followed by north Indian **Hindi** and **Urdu**, which have the same linguistic origins.
    3. **English** comes next with 527 million native speakers.
    4. **Arabic** is used by nearly 100 million more native speakers than **Spanish**.

These numbers reflect the fact that **two-thirds** of the world's population share only **12 native languages**.



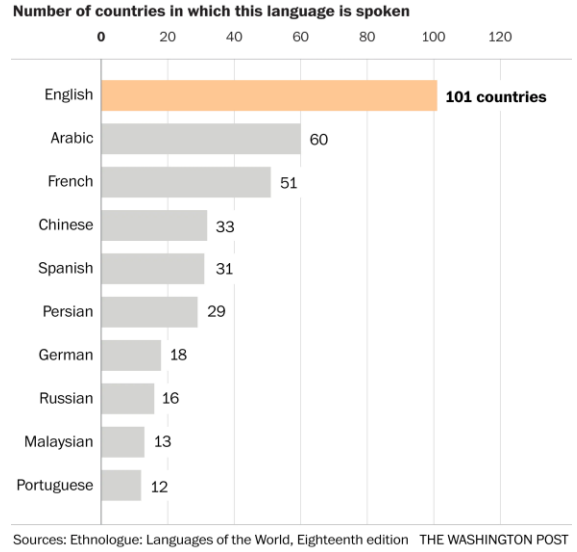
# Languages Spoken in more than just one country

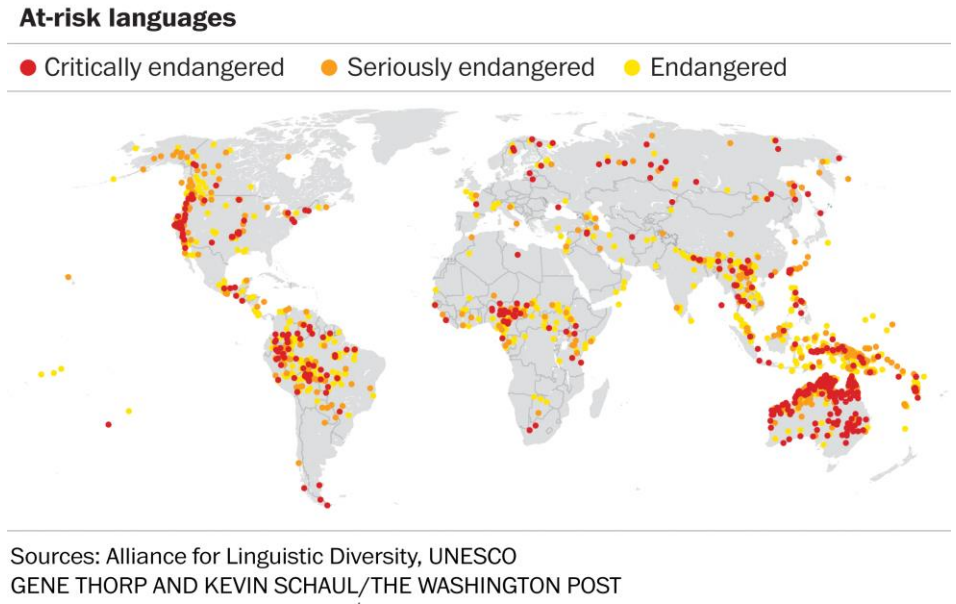
The reason why English, French and Spanish are among the world's most widespread languages has its roots in the imperial past of the nations where they originate.

Today, English is widely used as an official language.

- However, whether a country has English as its official language says little about how its citizens really communicate with one another. In some of the nations highlighted above, only a tiny minority learned English as a native language.

Most languages are spoken only by a handful of people. That's why about half of the world's languages will disappear by the end of the century



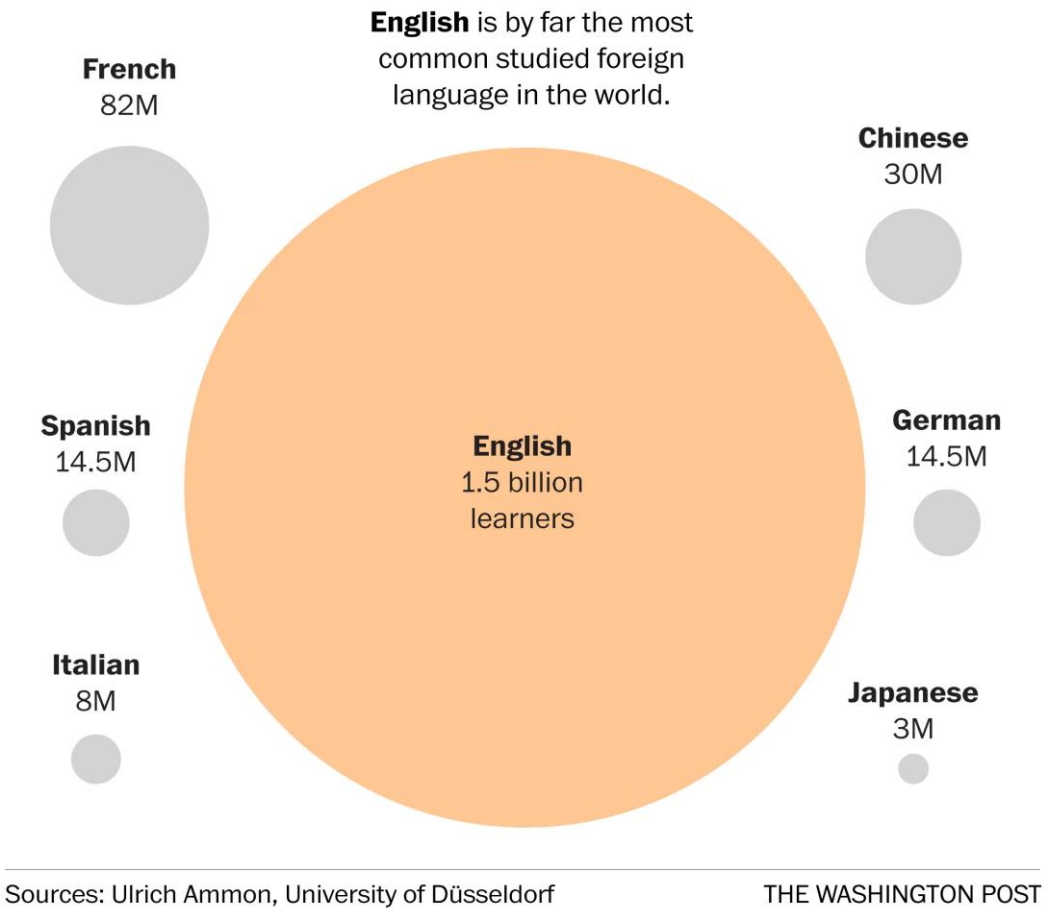



# How Many People Learn a New Language

Overall, more people learn English than French, Spanish, Italian, Japanese, German and Chinese combined.

Some languages have only recently gained attention:

- The number of U.S. colleges that teach **Chinese** has risen by 110 percent between 1990 and 2013, making the language more accessible.
- During the same time, the number of offered Russian college courses decreased by 30 percent.



# The Written Symbols


Symbols on bones and cave paintings go date back to the **Neanderthals** for at least 50,000 years.

**Writing symbols**:

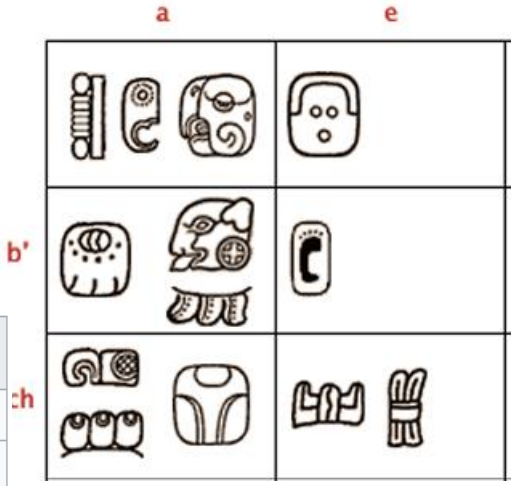
- **Syllabic** - based on sound - syllables that represent vowel sound (Korean), with or without surrounding consonants.
- **Logographic**, meaning that it uses symbols (such as Chinese characters) to represent meanings rather than sounds.

- **Sumerian** cuneiform and **Egyptian** hieroglyphs are generally considered to be the earliest examples of true writing systems. These ancient symbols, (including Mayan) combined logographic, syllabic and alphabetic elements.
  - These pictographic symbols known as hieroglyphics, or cuneiform wedges, both gradually evolved from proto-writing between **3400** and **3100** BCE.

**The Alphabet**: A standardized set of written symbols. Specifically, letters correspond to a categories of sounds that can distinguish one word from another in a given language.

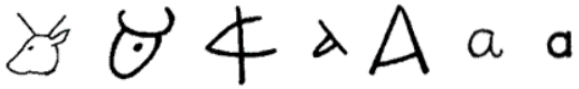
First Alphabet symbols date back to **1700 BC**.

- The Phoenician system is considered the first true alphabet and is the ultimate ancestor of many modern scripts, including Arabic, Cyrillic, Greek, Hebrew, Latin, and possibly Brahmic.




Mayan Glyphs

From Amun Ra to the letter A



| nr. | Reconstruction | IPA | value | Proto-Sinaitic | Ugaritic | Phoenician | Hebrew | Arabic | Greek | Latin | Cyrillic | Runic |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | *ʾalp* "ox" [30][31] | /ʔ/ | 1 | | | 𐤀 *ʾālep* | א *ʾālef* | ا *ʾalif* | Α alpha | A | А azŭ | ᚨ *ansuz |
| 2 | *bayt* "house" [30] | /b/ | 2 | | | 𐤁 *bēt* | ב *bēṯ* | ب *bāʾ* | Β bēta | B | В vědě, Б buky | ᛒ *berkanan |
| 3 | *gaml* "throwstick" | /g/ | 3 | | | 𐤂 *gīmel* | ג *gīmel* | ج *jīm* | Γ gamma | C, G | Г glagoli | ᚲ *kaunan |
| 4 | *dalt* "door" / *digg* "fish" | /d/ | 4 | | | 𐤃 *dālet* | ד *dāleṯ* | د *dāl*, ذ *ḏāl* | Δ delta | D | Д dobro | |

# A

A

# 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.

# String Matching - First Steps of Text Mining

- Regular String Matching
  - 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

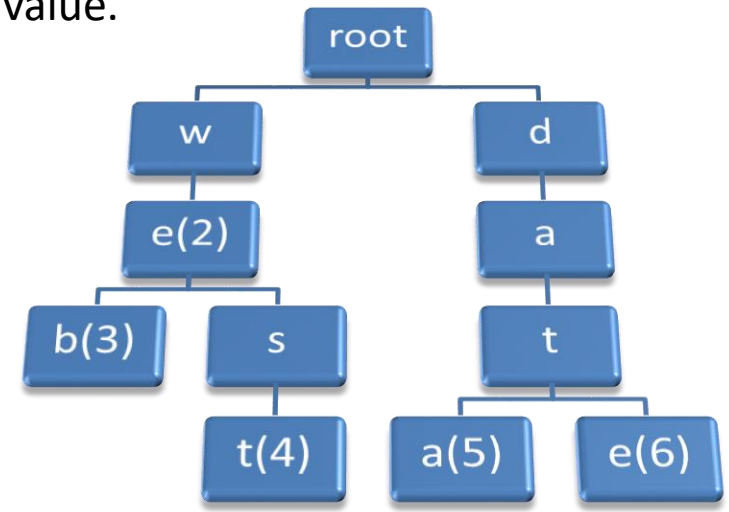
For regular string manipulations in R, please refer to lecture notes for the common packages.

# 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.



# Fuzzy String Matching

- Queries may have typos and that is a challenge with exact matches of strings. Especially for phrases that use foreign expressions. Fuzzy string matching is similar to regular string matching. It is the process of finding strings that are similar, but not necessarily exactly alike. Spell-checking is just one example of fuzzy string matching.

- Fuzzy string matching immediately opens up a number of questions for which the answers aren't so clear. For instance:
  - How many characters need to match?
  - What if the letters are the same but not in the same order?
  - What if there are extra letters?
  - Are some letters more important than others?

- Different approaches to these questions. These approaches can be broken down into three measures:
  1. Character overlap measures
  2. Edit distance measures
  3. N-gram edit distance

# Fuzzy string matching

- For the tasks of fuzzy string matching the R package ***stringdist*** is a useful tool. Among other functions it contains the function amatch() where the matching algorithm to use can be specified as an argument. Currently, the following distance metrics are supported by stringdist.

| Method name | Description |
|---|---|
| osa | Optimal string aligment, (restricted Damerau-Levenshtein distance). |
| lv | Levenshtein distance (as in R's native adist). |
| dl | Full Damerau-Levenshtein distance. |
| hamming | Hamming distance (a and b must have same nr of characters). |
| lcs | Longest common substring distance. |
| qgram | q-gram distance. |
| cosine | cosine distance between q-gram profiles |
| jaccard | Jaccard distance between q-gram profiles |
| jw | Jaro, or Jaro-Winker distance. |
| soundex | Distance based on soundex encoding. |

- For example, finding the Jaccard measure between 2 strings dict1 and dict2 can be done using

```
> amatch(query, c(dict1, dict2), method ="jaccard", maxDist=1)
[1] 2
```

returning the second object (dict2) form the lookup list ( c(dict1, dict2) ) as a best match.

# 1. Character overlap measures

There are several similarity measures to this approach.

- Jaccard similarity $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$
- Sorensen–Dice similarity $D(A,B) = \frac{2\,|A \cap B|}{|A|+|B|}$
- Ochiai similarity $O(A,B) = \frac{|A \cap B|}{\sqrt{|A||B|}}$

where $|A|$ and $|B|$ are the number of elements in each set

The Jaccard measure, or similarity coefficient measure is used in the context of string comparisons.

- It is computed as the percentage of unique characters that two strings share when compared to the total number of unique characters in both strings. Let A be a set of characters in the first string, and B is the set of characters in the second string. Then the Jaccard measure is always a number between 0 and 1 and it is calculated as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

where the $|A \cap B|$ is the intersect and $|A \cup B|$ is the union of the unique characters in the two strings A and B.

Sometimes the generalized Jaccard similarity is used which is the Jaccard similarity + a threshold.

# Exercise

Example: Let the query be the misspelled query word "analisis" and let's have the 2 candidate words A="analytics" and B="analysis" obtained from the dictionary (or a lookup table) as a possible match. This is illustrated by the following R code:

```
# Example 6: The Jaccard measure
library(stringdist)
dict1 <- "analytics"
dict2 <- "analysis"
query <- "analisis"
A <- unlist(strsplit(dict1,"")) # Get the set of characters in dict1
B <- unlist(strsplit(dict2,"")) # Get the set of characters in dict2
Q <-  unlist(strsplit(query,"")) # Get the set of characters in query
UA <- union(A,Q)
IA <- intersect(A,Q)
JA <- length(IA)/length(UA) # The Jaccard measure
sprintf("%s %f","The Jaccard Q-A measure is", JA) # Display calculated measure
UB <- union(B,Q)
IB <- intersect(B,Q)
JB <- length(IB)/length(UB) # The Jaccard measure
sprintf("%s %f","The Jaccard Q-B measure is", JB) # Display calculated measure
```

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Note that the
Sorensen–Dice similarity (0.588235) and
Ochiai similarity (0.5892557)
are the same for the AQ and BQ

# The Jaccard distance

- The resulting Jaccard measure indicates that a better match to the misspelled query word "analisis" is the word B="analysis".

[1] "The Jaccard Q-A measure is 0.625"
[1] "The Jaccard Q-B measure is 0.833"

$$d_J(A,B) = 1 - J(A,B)$$

- The Jaccard distance can also be defined. It measures dissimilarity between set of characters A and B.
- The Jaccard measure treats all letters equally, and a common extension is to assign a weight to each character based on its frequency.

- Another measure is the Jaro-Winkler distance which matches window of characters from the first string to the second string.

- The character overlap technique does not model character order very well, which is an important feature of wording.

# 2. Edit distance measures

- This is another approach to determining how similar one string is to another. It uses the number of edit operations required to turn one string into the other string. Computing the minimum sequence of operations needed to transform one string into another can be done by performing nxm comparisons where n is the length of one of the strings and m is the length of the other. The distance matrix is formed for each pair of strings. The columns (rows) of the matrix are set by all the letters contained in the first string (second string).

- For example, measuring the distance between the two strings: "Web Analytics" and "Data Analytics" is done by forming a matrix containing columns for each letter of the word "Web Analytics" and containing rows for each letter of the word "Data Analytics".

- This approach can be improved by setting a threshold for edit distances and introducing weights on the edit distance.

# 3. Q-gram edit distance

- This approach is similar to the previous one but instead of looking at columns and rows made of letters q-characters from the string are considered.

- For example, measuring the distance between the two strings: "Web Analytics" and "Data Analytics" by using 3-gram edit distance is done by forming a matrix containing columns for each 3 letters of the word "Web Analytics" (i.e. "Web", "eb ", "b A" etc.) and containing rows for each 3 letters of the word "Data Analytics" (i.e. "Dat", "ata ", "at ", "t A" etc.).

# See Embeddings Slides

- For more on representing text as numeric vectors.

# 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

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.

# Step 1. Loading/Accessing Files with 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.

# Step 2. Content extraction from a Text File - Exercise

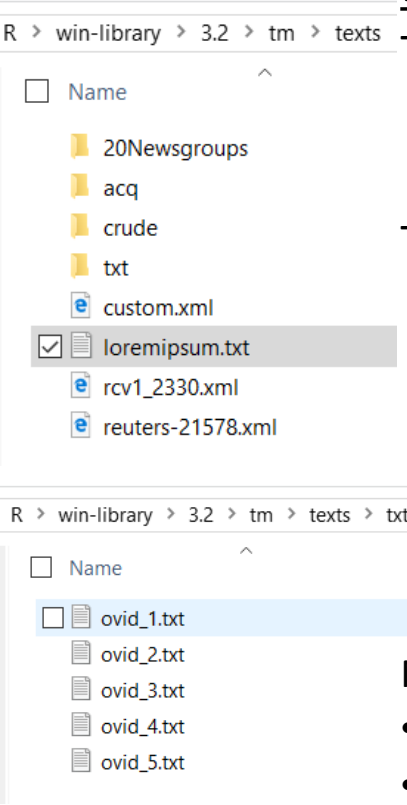
Use R and tm package to extract the content from the 2 poems that come with the “tm” package.

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

# Step 2. Content extraction from a Text File - Exercise




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.**

# See OCR Text Mining Exercises

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

# 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.

# 3a. 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.

# 3b. 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"))
```

# 3c. 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.

# A

A

# Step 4. Staging - Creating a Document Term Matrix

- A document term matrix is a matrix with
  - documents as the **rows**
  - terms as the **columns**
  - a **count** of the frequency of words as the cells of the matrix.

- To create the DTM and TF-IDF matrix this code is used (in the "tm" package)

```
> dtm <- DocumentTermMatrix(Docs.corpus)
> tfidf <- DocumentTermMatrix(Docs.corpus, control = list(weighting = weightTfIdf))
```

- To inspect the document term matrix use

```
> inspect()
```

- The transpose of the DocumentTermMatrix() is created with

```
> TermDocumentMatrix(Docs.corpus)
```

# Exercise: Explore the DTM of "tm.pdf"

Implement all the text mining steps to explore your DTM data

1. Access the "tm.pdf" file from your tm package
   - Use proper tm readers and sources
2. Create Corpus of the "tm.pdf" text content.
3. Skip preprocessing
4. Create the dtm (Document Term matrix).
5. Analyze term frequencies in "tm.pdf".
   - What is max appearance frequency of a term?
   - What are the most/least frequent terms?

# Exercise Hints

- To find the term frequency you can use

```
freq <- colSums(as.matrix(dtm)) # Term frequencies
ord <- order(freq) # Ordering the frequencies
freq[tail(ord)] # Most frequent terms
freq[head(ord)] # Least frequent terms
findFreqTerms(dtm, lowfreq=10) # List terms (alphabetically) with frequency higher than 10

# Plot Histogram of Word Frequencies
wf <- data.frame(word=names(freq), freq=freq)
head(wf)
library(ggplot2)
p <- ggplot(subset(wf, freq>7), aes(word, freq))
p <- p + geom_bar(stat="identity")
p <- p + theme(axis.text.x=element_text(angle=45, hjust=1))
p
```

# Step 5. Explore your DTM data

Exploring the Document Term Matrix (*Text Mining Exercises 1 & 2.R*)

- The term frequencies can be obtained as a vector by converting the document term matrix into a matrix and summing the column counts. Obtaining the most frequent terms in the "tm.pdf" is illustrated with the following script.

```
# Example 5: Creating a Document Term Matrix
Docs.pth <- system.file(file.path("doc", "tm.pdf"), package = "tm") # Path to tm.pdf
Docs.corpus <- Corpus(URISource(Docs.pth), readerControl = list(reader = readPDF(engine = "xpdf")))
dtm <- DocumentTermMatrix(Docs.corpus) # Document Term Matrix
freq <- colSums(as.matrix(dtm)) # Term frequencies
ord <- order(freq) # Ordering the frequencies
freq[tail(ord)] # Most frequent terms
```

- The output of this script lists the most frequent terms

```
> freq[tail(ord)]
 which corpus    for   text    and    the
    18     26     27     27     31     92
```

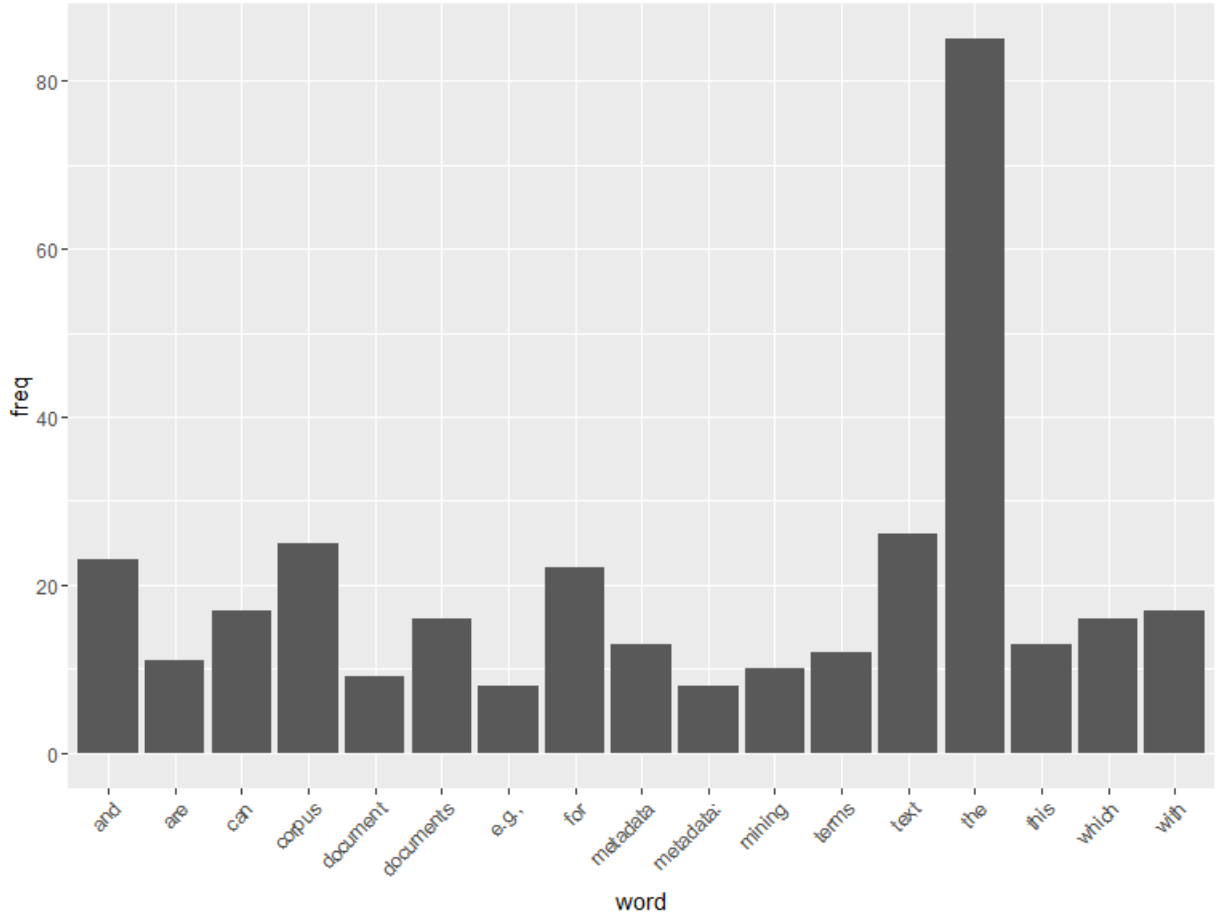
- For example, the most frequent terms in the corpus that appear at least 10 times can be seen with

```
> findFreqTerms(dtm, lowfreq=10)
```

# Histogram of Word Frequencies

- Words with freq > 7 are alphabetically ordered.

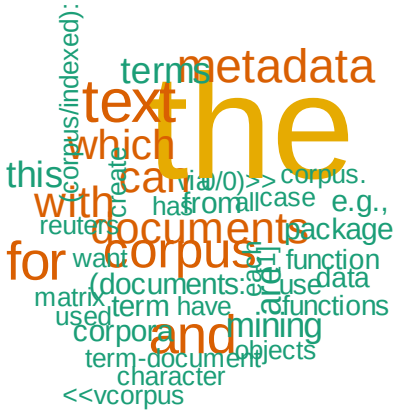
```
# Create a Histogram
library(ggplot2)
# Organize Histogram Data
WordFreq <- data.frame(word=names(freq), freq=freq)
p <- ggplot(subset(WordFreq, freq>7), aes(word, freq));
p <- p + geom_bar(stat="identity")
p <- p + theme(axis.text.x=element_text(angle=45, hjust=1))
p # Display Histogram
```



# Distribution of Term Frequencies

- For a quick visual overview of the frequency of words in a corpus we can generate a word cloud with:

```
> library(wordcloud)
> set.seed(123)
> wordcloud(names(freq), freq, min.freq=5, colors=brewer.pal(6, "Dark2"))
```




- If needed the document term matrix can be converted to a simple matrix and saved.

```
> matdtm <- as.matrix(dtm)
> write.csv(matdtm, file=" dtm.csv")
```

# Step 6. Analysis

# Text Data Categorization

- The goal is to learn from text data, the process is subdivided as follows:
  - Supervised learning (know categories)
    - Classification (ranking sports teams, categories team names and scores being known)
  - Unsupervised learning (no knowledge of what categories are contained in data)
    - Clustering (grouping a set of emails without knowing what they contain)

- There are many different clustering algorithms.
- They all use some kind of similarity measure (“distance”) to determine the clusters.
- Typically, the "distance" between two documents is measured as the distance between two vectors.
- There are many different "distance" measures available such as:
  - Euclidean distance (based on the Pythagorean theorem)
  - Manhattan distance (laid out on a square grid, like the streets of Manhattan in New York City)
  - Cosine distance (cosine similarity, as defined earlier)

# Exercise - Text Clustering

Task: group (cluster) by similarity documents with the following 12 most frequent words

```
# Example 9: Word Clustering
doc1 <- c("Web Analytics", "Text Analysis", "Web Mining", "Text Mining")
doc2 <- c("Data Processing", "Machine Learning", "learn from data", "Big Data")
doc3 <- c("bedroom furniture", "dining room furniture", "diner chair", "new chairs")
```

- Implement preprocessing - stemming. This will affect the clustering, so you need some intuition to decide what particular pre-processing you need to implement for particular data to achieve better results. Keep 2 version of your corpus
  - Stemmed
  - Not stemmed
- Create the Document term matrix
- Create similarity dendrogram using similarity measures package ("*arules*") and *hclust()* (performs hierarchical cluster analysis).
- Create similarity dendrogram using optimal string alignment (restricted Damerau-Levenshtein distance).

# Exercise - Text Clustering

Task: Assume that you have several documents with 12 most frequent strings in them and you would like to group (cluster) them together by similarity.

```
# Example 9: Word Clustering
doc1 <- c("Web Analytics", "Text Analysis", "Web Mining", "Text Mining")
doc2 <- c("Data Processing", "Machine Learning", "learn from data", "Big Data")
doc3 <- c("bedroom furniture", "dining room furniture", "diner chair", "new chairs")
doc <- c(doc1,doc2,doc3) # Merge all strings
dtm <- as.matrix(DocumentTermMatrix(Corpus(VectorSource(doc)))) # Document term matrix
```

Note

- To implement the clustering, we need to create the Document term matrix
- Implement any preprocessing we want to apply.
- This will affect the clustering, so you need some intuition to decide what particular pre-processing you need to implement for particular data to achieve better results.

# Text Clustering Examples

- If necessary, stemming can be performed on the strings to improve the clustering performance.

```
corpus.temp <- tm_map(Corpus(VectorSource(doc)), stemDocument, language = "english")
dtm <- as.matrix(DocumentTermMatrix(corpus.temp))
```

- Now all of the terms contained in the document term matrix look like this:

```
colnames(dtm)
 [1] "analysi"  "analyt"   "bedroom"  "big"     "chair"    "data"    "dine"    "diner"    "from"
[10] "furnitur" "learn"    "machin"   "mine"    "new"      "process" "room"    "text"     "web"
```

- This will affect the clustering since if you perform stemming, the term "chair" and "chairs" will be stemmed to a single term "chair" for example.

# Text Clustering Examples

- The non stemmed document term matrix for the 12 strings and the first 7 alphabetically ordered terms of this example looks like this:

```
> dtm[ ,1:7]
    Terms
Docs analysis analytics bedroom big chairs data dining
  1        0         1       0   0      0    0      0
  2        1         0       0   0      0    0      0
  3        0         0       0   0      0    0      0
  4        0         0       0   0      0    0      0
  5        0         0       0   0      0    1      0
  6        0         0       0   0      0    0      0
  7        0         0       0   0      0    1      0
  8        0         0       0   1      0    1      0
  9        0         0       1   0      0    0      0
  10       0         0       0   0      0    0      1
  11       0         0       0   0      1    0      0
  12       0         0       0   0      1    0      0
```

# Text Clustering Examples

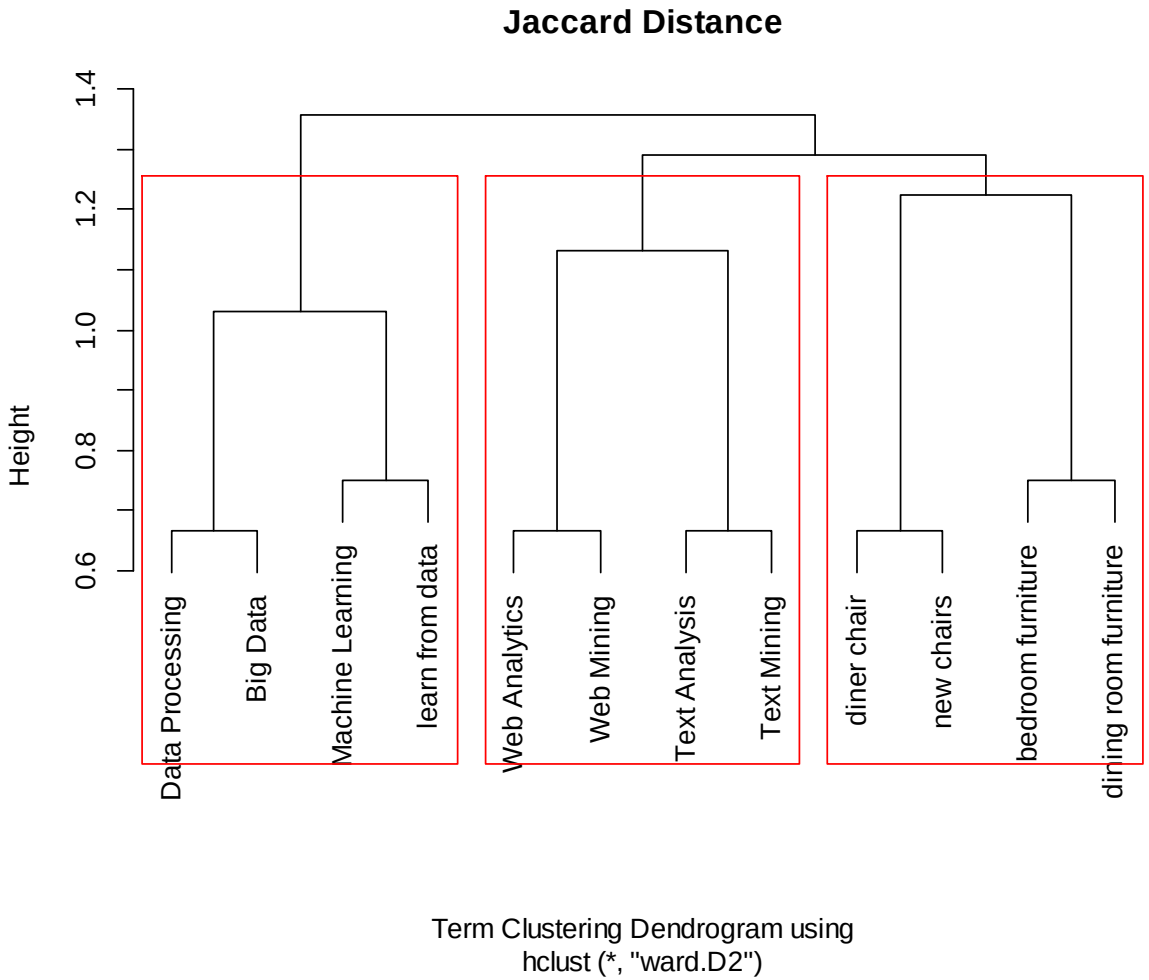
- The following code illustrates the text clustering.

```
# Using Jaccard Distance
library(cluster) # Load similarity measures package
d <- dist(dtm, method="binary") # Find distance between terms
cl <- hclust(as.dist(d)) # Perform clustering
cl$labels=doc # Assign labels to cluster leaves (documents we had)
plot(cl,main="Jaccard Distance") # Plot and set plot title
```

- Note:
  - The similarity measures package ("*cluster*") that was loaded for this purpose.
  - Obtaining the cluster leaves (*cl* objet) with *hclust()* where the argument was coerced as distance with *as.dist()*.
  - *hclust()* performs hierarchical cluster analysis.
  - The last two lines are related to plotting the dendogram (see lecture notes).
- Try different preprocessing to see the effect on clustering.
- Try different packages and different clustering functions and measures. Always consult help files/package for details.

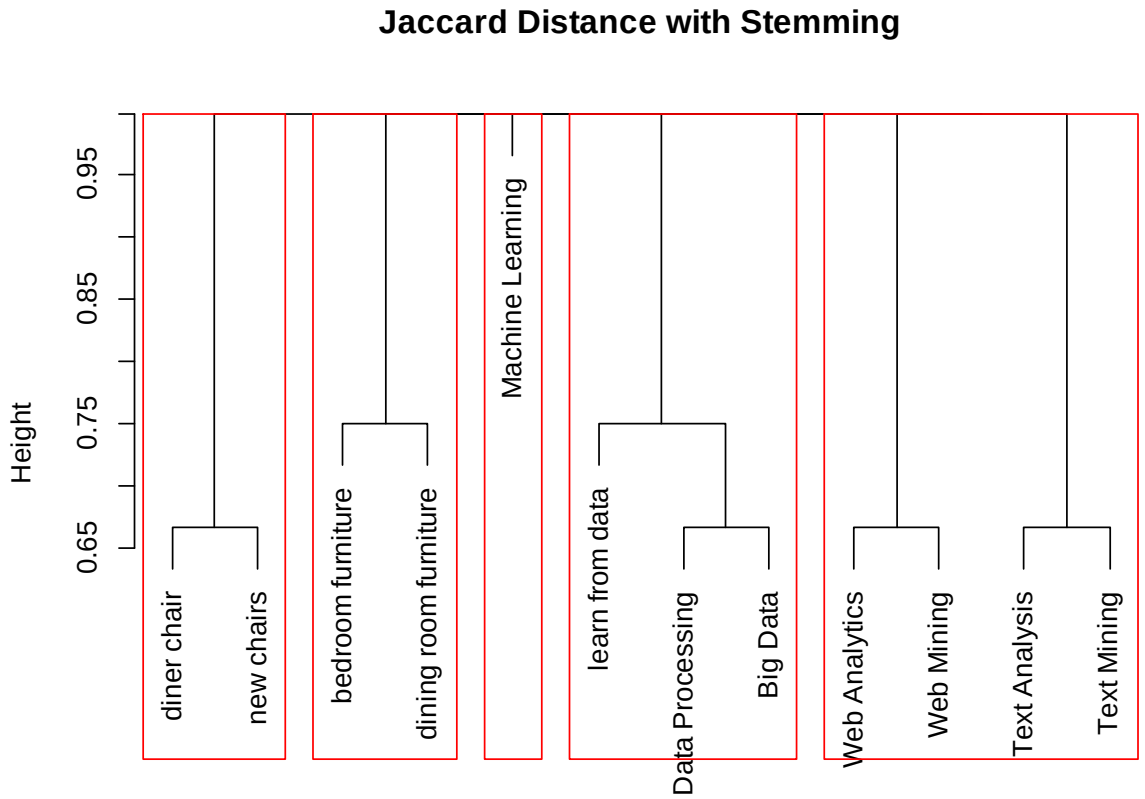
# Word clustering

- Words clustered by Jaccard distance similarity.



# Word clustering with Stemming

- Words are stemmed and then clustered by Jaccard distance similarity.




```
# With stemmed terms
corpus.temp <- tm_map(Corpus(VectorSource(doc)), stemDocument, language = "english")
dtm <- as.matrix(DocumentTermMatrix(corpus.temp))
d <- dist(dtm, method="binary") # Find distance between terms in dtm
cl <- hclust(as.dist(d)) # Perform clustering
cl$labels=doc # Assign labels (terms used) to cluster leaves
plot(cl,main="Jaccard Distance with Stemming",xlab="Term Clustering Dendrogram using") # Set plot title
rect.hclust(cl, k=5, border="red") # draw dendogram with red borders around the 5 clusters
```

Term Clustering Dendrogram using
hclust (*, "complete")

# Text Clustering Examples

- Yet another package (stringdist) can be used to perform clustering as illustrated with the following code:

```
# Hierarchical clustering
library(stringdist)
d <- stringdistmatrix(doc, doc) # Pairwise string distances (optimal string alignment)
cl <- hclust(as.dist(d)) # Perform hierarchical clustering
cl$labels=doc # Assign labels to cluster leaves
plot(cl) # Plot and set plot title
```

- Note that here within the function stringdistmatrix() we are using the default distance measure which is term optimal string alignment or the restricted Damerau-Levenshtein measure.

# A

A

# Classification Algorithms

- There are several classification algorithms.

- Consider one of them, the k-NN algorithm. In the k-NN algorithm,
  - Uses (TF - IDF) as the default weighting measure which combines the term frequency (TF), with inverse document frequency (IDF) which is logarithmically scaled. IDF tells whether the term is common or rare across all documents.
  - Cosine similarity is used as the default similarity metric.

- The arguments of the function *knn()* are

  knn(train, test , Tags, k)

  - **train** is a dataset for which classification is already known.
  - **test** is a dataset you are trying to classify.
  - **Tags** is a factor of correct answers for the training dataset
  - **k** # number of neighbors considered per row of data.

- **Note** the function *knn()* expects same number of **train** and **test** datasets.

# Exercise

Perform document classification using the *knn()* function (algorithm) from the *class* package.

- Classify the following documents (*Text Mining Exercises 1 & 2.R*):

```
# Example 10: Document kNN Text Classification
Doc1 <- "I spent 10K on my car. Compared to the prices of most cars in their class it was cheap. It is a red car so I like it and it has a lot of space."
Doc2 <- "I checked the car prices and I could not find a red car for under 10K. So the price was good even though it had a hole. I heard that it belonged to a movie star."
Doc3 <- "I like the red color, so I would buy a red car even if the car's price is over 10K."
Doc4 <- "I don't like red cars. The insurance for red cars is higher regardless of the price and I would not spend more than 10K. I like black cars."
Doc5 <- "A red giant star can curve the space to form a black hole. In absence of stars the space is flat."
Doc6 <- "With exception of the stars the space is filled with blackness making the black holes even harder to see."
Doc7 <- "Our sun is a small star and it will not end as a black hole. It does not have enough mass to curve the space."
Doc8 <- "Very few stars will end as black holes but still the space contains large number of black holes."
```

# Hints using the k-NN algorithm

- There are 8 simple documents containing few sentences.
  - The first 4 documents (document ID from 1 to 4) mostly refer to subject related to cars.
  - The other 4 documents (document ID from 5 to 8) relate to space.
- There are some overlapping words between these 2 categories (space, star, red, black etc.).
- Note:
  - All documents are combined into a Corpus using R's *c()* function.

    ```
    doc <- c(Doc1,Doc2,Doc3,Doc4,Doc5,Doc6,Doc7,Doc8) # Merge all strings
    ```

  - Preprocessing is implemented before *dtm* document term matrix is created.
  - The package "*class*" is used for implementing kNN.

    ```
    library(class) # Using kNN
    ```

  - The data from *dtm* is equally split into test and train datasets (*train.doc* and *test.doc* ).

    ```
    train.doc <- dtm[c(1,2,5,6),] # Dataset for which classification is already known
    test.doc <- dtm[c(3,4,7,8),] # Dataset you are trying to classify
    ```

  - The correct answers for the training dataset the tags (groups) are specified as factor data format required by the *knn()* classification function. Note the use of R's *c()* and *rep()* functions!

    ```
    Tags <- factor(c(rep("cars",2), rep("space",2))) # Tags - Correct answers for the training dataset
    ```

  - Finally, the classification is implemented, specifying that we have k=2 classes. The classification probability is returned for each test dataset.

    ```
    prob.test<- knn(train.doc, test.doc, Tags, k = 2, prob=TRUE) # k-number of neighbors considered
    ```

# Code Implementation

Pre-Processing & Creating the DTM

```r
rm(list=ls()); cat("\014") # Clear Workspace and Console
library(tm) # Load the Text Mining package
library(class) # Using kNN

### --- Example 10: Document kNN Text Classification   ----
Doc1 <- "I spent 10K on my car. Compared to the prices of most cars in their class it was cheap. It is a red car so I like it and it has a lot of space."
Doc2 <- "I checked the car prices and I could not find a red car for under 10K. So the price was good even though it had a hole. I heard that it belonged
Doc3 <- "I like the red color, so I would buy a red car even if the car's price is over 10K."
Doc4 <- "I don't like red cars. The insurance for red cars is higher regardless of the price and I would not spend more than 10K. I like black cars."
Doc5 <- "A red giant star can curve the space to form a black hole. In absence of stars the space is flat."
Doc6 <- "With exception of the stars the space is filled with blackness making the black holes even harder to see."
Doc7 <- "Our sun is a small star and it will not end as a black hole. It does not have enough mass to curve the space."
Doc8 <- "Very few stars will end as black holes but still the space contains large number of black holes."

True.Tags <- c("cars", "cars", "cars", "cars", "space", "space", "space", "space")

doc <- c(Doc1,Doc2,Doc3,Doc4,Doc5,Doc6,Doc7,Doc8) # Merge all strings
corpus <- Corpus(VectorSource(doc))

# Pre-processing
corpus.temp <- tm_map(corpus, removePunctuation) # Remove Punctuation
corpus.temp <- tm_map(corpus.temp, stemDocument, language = "english")# Perform Stemming
dtm <- as.matrix(DocumentTermMatrix(corpus.temp)) # Document term matrix
```

# Code Implementation

## Prepare Train and Test Data Before Classification

```
# Preparing the Data
Positive <- "cars"; Negative <- "space"; # Set "cars" as Positive and "space" as Negative
train.Range <- c(1,2,5,6) # Train Indices
test.Range <- c(3,4,7,8) # Test Indices
train.doc <- dtm[train.Range,] # Train Data Set
test.doc <- dtm[test.Range,] # Test Data Set
Train.Tags <- factor(c(rep(Positive,2), rep(Negative,2)), levels = c(Positive, Negative)) # Train Tags as Factors
Test.Tags <- factor(c(rep(Positive,2), rep(Negative,2)), levels = c(Positive, Negative)) # Test Tags as Factors
```

## Organize Train Data for Display

```
# Organize Train Data for Display
a <- train.Range  # Train Data Indices
b <- True.Tags[train.Range] # True Tags from Data
c <- Train.Tags # Train Tags created from the data range as Factor
TD <- data.frame('Train Doc'=a, 'True Tags'=b, 'Train Tags'=c)
```

## Implement knn() and Organize Predictions for Display

```
prob.test<- knn(train.doc, test.doc, Train.Tags, k = 2, prob=TRUE) # Implement knn Classifier

# Predictions on Test Data
a <- test.Range # Test Data Indices
b <- True.Tags[test.Range] # True Tags from Data
c <- Test.Tags # Test Tags
d <- levels(prob.test)[prob.test] # Predicted Tags by the Knn model
e <- attributes(prob.test)$prob # Probability of Predictions
f = prob.test==Test.Tags # Compare Predictions with Test Tags
Predictions <- data.frame("Test.Doc"=a, 'True.Tags'=b, 'Test.Tags'=c, 'Pred.Tags'=d, 'Prob'=e, 'T_F'=f)

result <- cbind(TD, '.'=rep(c('-'), dim(TD)[1]), Predictions)
knitr::kable(result) # Display result of Language Detection
```

# Analyzing the output of the knn()

The classification probability is returned for each test dataset.

Remember the test dataset contained:

1. doc3 related to cars.
2. doc4 related to cars.
3. doc7 related to space.
4. doc8 related to space.

The classification was correct.

| Train.Doc | True.Tags | Train.Tags | . | Test.Doc | True.Tags | Test.Tags | Pred.Tags | Prob | T_F |
|---|---|---|---|---|---|---|---|---|---|
| 1 | cars | cars | - | 3 | cars | cars | cars | 1 | TRUE |
| 2 | cars | cars | - | 4 | cars | cars | cars | 1 | TRUE |
| 5 | space | space | - | 7 | space | space | space | 1 | TRUE |
| 6 | space | space | - | 8 | space | space | space | 1 | TRUE |

Note that if you change Doc8 which is related to **space** to a **car** related sentence such as "*My car is priced well.*" and rerun the script, the Doc8 is still classified correctly with the tag "cars" but now with probability of just 50%.

| Train.Doc | True.Tags | Train.Tags | . | Test.Doc | True.Tags | Test.Tags | Pred.Tags | Prob | T_F |
|---|---|---|---|---|---|---|---|---|---|
| 1 | cars | cars | - | 3 | cars | cars | cars | 1.0 | TRUE |
| 2 | cars | cars | - | 4 | cars | cars | cars | 1.0 | TRUE |
| 5 | space | space | - | 7 | space | space | space | 1.0 | TRUE |
| 6 | space | space | - | 8 | space | space | space | 0.5 | TRUE |

Why?

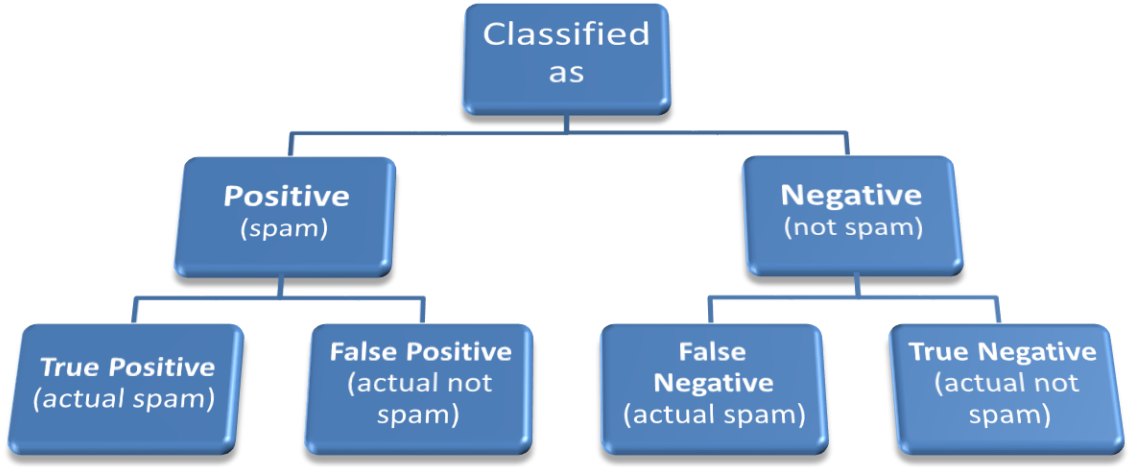
You would need to change the training set and the Tags accordingly.

# Effectiveness Analysis

- How do we measure effectiveness of an information retrieval system?
- Let’s illustrate the effectiveness analysis with the following example.
- Example:
  - Consider the classification performance of an email spam filter based on a user specified set of "spam" words. The filter analyzed 100 emails out of which 27 were spam. The filter classified 21 emails as spam, out of which 11 were actual spam.
- What is the effectiveness of this spam filter?

# Let’s Define Some Terms

- **Positive** Event:
  - It is typical to choose as "positive" the rarer event
  - So let's choose a spam email as a positive event.
- **Negative** Event:
  - Let’s choose regular email as negative event.
- Two types of events (emails)
  - **Actual**
  - **Predicted** (classified)



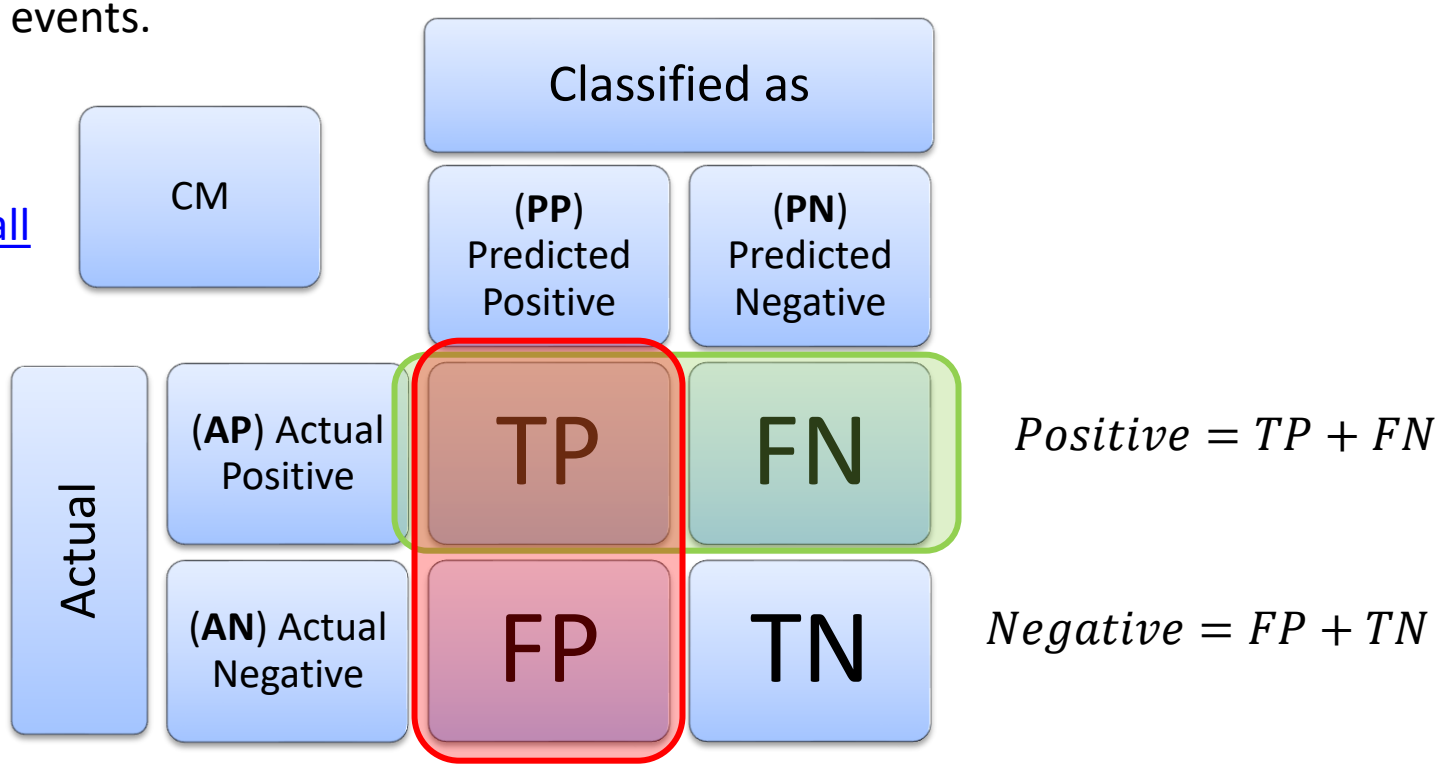
# Confusion Matrix Terms

- In particular the terms related to the spam email classification example are:
  - TP - True Positive, actual spam emails (true) classified as spam (positive).
  - FP (Type 1 error) - False Positive, actual not spam emails (false) classified as spam (positive).
  - FN (Type 2 error) - False Negative, actual spam emails (false) classified as not spam (negative).
  - TN - True Negative, actual not spam emails (true) classified as not spam (negative).
- The 1 and the 0 refer to positive and negative events.

Useful links:

https://en.wikipedia.org/wiki/Precision_and_recall

https://en.wikipedia.org/wiki/Confusion_matrix



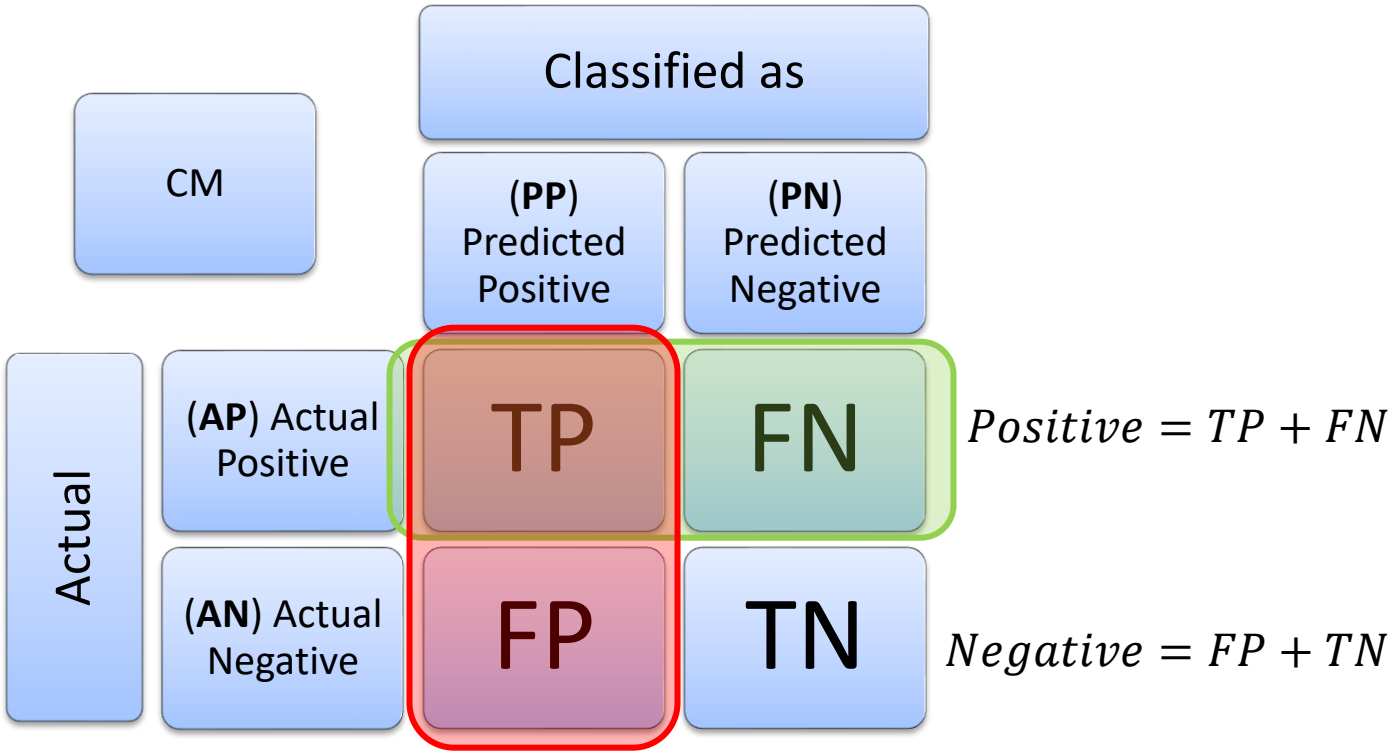
# Precision and Recall

- A metric to assess the effectiveness of any classification process
  - **Precision** - is what fraction of the returned results are relevant to the information need.
  - **Recall** - is what fraction of the relevant documents in the collection were returned by the system.
- Mathematically the CM terms are related to precision and recall by the following formulas:

- Question: Why do we not consider TN?

$$Precision = \frac{TP}{Predicted\ Positive} = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{Actual\ Positive} = \frac{TP}{TP + FN}$$



# Effectiveness Analysis of a Spam Filter

- Example: Effectiveness analysis of a spam filter
  - Consider the classification performance of an email spam filter based on a user specified set of "spam" words. The filter analyzed 100 emails out of which 27 were spam. The filter classified 21 emails as spam, out of which 11 were actual spam.
- This means (Take actual **spam** emails to be a **positive** event)
  - **TP** - True Positive, actual spam emails (**true**) classified as spam (**positive**).
  - **FP** - False Positive, actual not spam emails (**false**) classified as spam (**positive**).
  - **FN** - False Negative, actual spam emails (**false**) classified as not spam (**negative**).
  - **TN** - True Negative, actual not spam emails (**true**) classified as not spam (**negative**).

$$T = 100 \Longrightarrow AN = T - AP = 100 - 27 = 73$$
$$AP = 27 \Longrightarrow AP - TP = 27 - 11 = 16$$
$$PP = 21 \Longrightarrow PP = TP + FP \Longrightarrow \boldsymbol{FP} = PP - TP = 10$$
$$\boldsymbol{TP} = 11$$
$$\boldsymbol{FN} = AP - TP = 27 - 11 = 16$$
$$\boldsymbol{TN} = AN - FP = 73 - 10 = 63$$

- Thus the

$$Precision = \frac{TP}{TP+FP} = \frac{11}{11+10} \cdot 100 = 52.4\%$$

$$Recall = \frac{TP}{TP+FN} = \frac{11}{11+16} \cdot 100 = 40.7\%$$

| CM | PP (Predicted Positive) | PN (Predicted Negative) |
|---|---|---|
| AP (Actual Positive) | $TP = 11$ | $FN = 16$ |
| AN (Actual Negative) | $FP = 10$ | $TN = 63$ |

- High precision and recall gives us confidence in the classification effectiveness.
  - Note: before we assess the effectiveness of the classification, typically we divide the available data into 60% training, 20% cross validation and 20% test datasets.
  - Note that the precision and the recall need to be estimated on the **cross-validation** dataset.

# F score

- Note that these measures may not always be a good effectiveness indicator.
- Consider the following situation

$$TP = 2$$
$$FP = 34$$
$$FN = 1$$
$$TN = 63$$

| CM | PP (Predicted Positive) | PN (Predicted Negative) |
|---|---|---|
| AP (Actual Positive) | $TP = 2$ | $FN = 1$ |
| AN (Actual Negative) | $FP = 34$ | $TN = 63$ |

- The Precision=5.6%
- The Recall=66.7%
- Is this good or bad?
- It is hard to quantify this performance. Better to have single measure.
- F score is defined in terms of precision and recall.

$$F_{score} = 2\frac{P\ R}{P + R}$$

- The F score for the first performance is 45.8%, while for the second is only 10.3%, indicating that the first performance is much more effective.

# Classification & Ranking

- Binary **classification**, placing items into one of two types or classes.
- But what if the items in one class are not “created equally” and we want to rank the items within a class?
  - i.e. to say that one email is the most spammy, another is the second most spammy etc.
- Generating rules for **ranking** a list of items is a common task in machine learning.
- This kind of ranking is produced by a **recommendation system**, that we all have encountered.
  - Ecommerce websites benefit from leveraging data on their users to generate recommendations for other products their users might be interested in (i.e. books on a similar topic, accessories, movies etc.).
- Ranking falls into the supervised machine learning type.

# Ranking Example: Rank Newsgroups

- Define content features by extracting it from the messages.
- Specifically, if there are common terms in the subjects and bodies of emails received by a user, then future emails that contain these terms in the subject and body may be more important than those that do not.
- This is actually a common technique, and it is mentioned briefly in the description of Google's priority inbox.
- Some of the features:
  - Who is it from?
  - Subject line content.
  - Is it active thread, and how many replies are there?
  - Message content.

# Exercise - String Manipulation

- **Task1:** Create a corpus from the following text:

```
# Exercise: Corpus Manipulation.
library(tm)
Doc1 <- "From: ritterbus001@wcsub.ctstateu.edu"
Doc2 <- "Subject: Re: IR remote control receiver"
Doc3 <- "Nntp-Posting-Host: wcsub.ctstateu.edu"
Doc4 <- "Organization: Yale University, Department of Computer Science, New Haven, CT"
Doc5 <- "In article <wb9omc.735429954@dynamo.ecn.purdue.edu>, wb9omc@dynamo.ecn.purdue.edu (Duane P Mantick)
writes:"
```

- **Task2:** Use grepl() to create a meta field in your corpus called “Subject” and place the textual content of the subject line there. (i.e Doc.Corpus[[1]]$meta$Subject<-”…”)

# Exercise - String Manipulation

```r
### --- Exercise 1: Corpus Manipulation. ----
rm(list=ls()); cat("\014") # clear all
library("tm")
Doc1 <- "From: ritterbus001@wcsub.ctstateu.edu"
Doc2 <- "Subject: Re: IR remote control receiver"
Doc3 <- "Nntp-Posting-Host: wcsub.ctstateu.edu"
Doc4 <- "Organization: Yale University, Department of Computer Science, New Haven, CT"
Doc5 <- "In article <wb9omc.735429954@dynamo.ecn.purdue.edu>, wb9omc@dynamo.ecn.purdue.edu (Duane P Mantick) writes:"

doc <- c(Doc1,Doc2,Doc3,Doc4,Doc5) # Merge all text
Doc.Corpus <- VCorpus(VectorSource(doc))
Doc.Corpus[[1]]$meta

# Preprocessing -- Create corpus only from the "Subject:" line
Subject.List <- list(); cc <- 0
for (ff in 1:length(Doc.Corpus)) {
  TextLine <- unlist(Doc.Corpus[[ff]][1])
  if (grepl("Subject:",TextLine)) {
    cc <- cc +1
    Subject.List[cc] <- gsub("Subject: ","",TextLine)
    Doc.Corpus[[ff]]$meta$Subject <- gsub("Subject: ","",TextLine)
  }
}

Doc.Corpus[[2]]$meta$Subject # Subject line in Corpus
```

# A

A