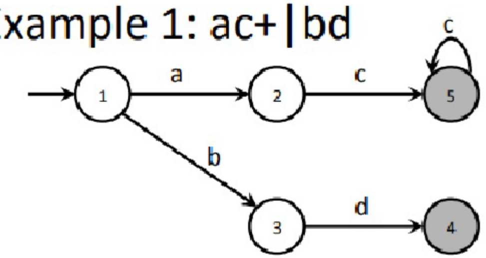
Text Analytics, Natural Language Processing & Text Similarity

David Li

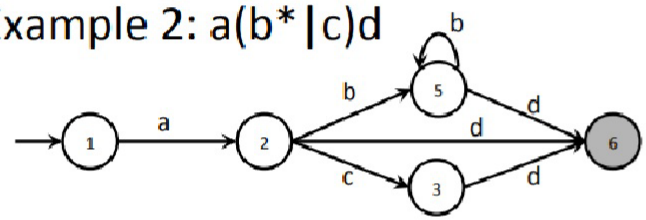
Regular expressions

State machine examples

- Example 1: ac+|bd

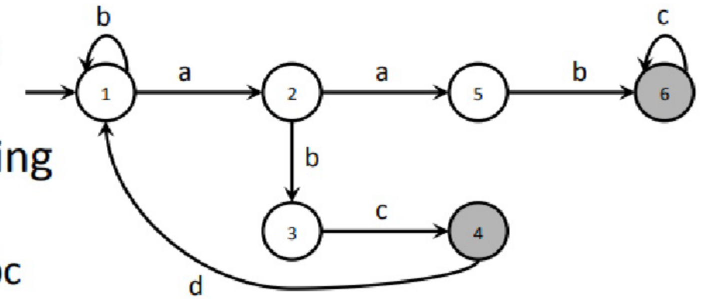


- Example 2: a(b*|c)d



Matching example

- Graph:



- Matching
  - abc
  - bbabc
  - baab
  - baabcc
  - abcdbbbabc
  - abcd
  - e

Regular expressions

- Regular expressions can be thought of as a combination of literals and *metacharacters*
- To draw an analogy with natural language, think of literal text forming the words of this language, and the metacharacters defining its grammar
- Regular expressions have a rich set of metacharacters

Literals

Simplest pattern consists only of literals. The literal "nuclear" would match to the following lines:

```
Ooh. I just learned that to keep myself alive after a
nuclear blast! All I have to do is milk some rats
then drink the milk. Aweosme. :}

Laozi says nuclear weapons are mas macho

Chaos in a country that has nuclear weapons -- not good.

my nephew is trying to teach me nuclear physics, or
possibly just trying to show me how smart he is
so I'll be proud of him [which I am].

lol if you ever say "nuclear" people immediately think
DEATH by radiation LOL
```

Literals

The literal "Obama" would match to the following lines

```
Politics r dum. Not 2 long ago Clinton was sayin Obama
was crap n now she sez vote 4 him n unite? WTF?
Screw em both + Mcain. Go Ron Paul!

Clinton conceeds to Obama but will her followers listen??

Are we sure Chelsea didn't vote for Obama?

thinking ... Michelle Obama is terrific!

jetlag..no sleep...early mornig to starbux..Ms. Obama
was moving
```

Regular Expressions

- Simplest pattern consists only of literals; a match occurs if the sequence of literals occurs anywhere in the text being tested
- What if we only want the word "Obama"? or sentences that end in the word "Clinton", or "clinton" or "clinto"?

Regular Expressions

We need a way to express

- whitespace word boundaries
- sets of literals
- the beginning and end of a line
- alternatives ("war" or "peace")

Metacharacters to the rescue!

Metacharacters

Some metacharacters represent the start of a line

```
^i think
```

will match the lines

```
i think we all rule for participating
i think i have been outed
i think this will be quite fun actually
i think i need to go to work
i think i first saw zombo in 1999.
```

Metacharacters

$ represents the end of a line

```
morning$
```

will match the lines

```
well they had something this morning
then had to catch a tram home in the morning
dog obedience school in the morning
and yes happy birthday i forgot to say it earlier this morning
I walked in the rain this morning
good morning
```

Character Classes with []

We can list a set of characters we will accept at a given point in the match

```
[Bb][Uu][Ss][Hh]
```

will match the lines

```
The democrats are playing, "Name the worst thing about Bush!"
I smelled the desert creosote bush, brownies, BBQ chicken
BBQ and bushwalking at Molonglo Gorge
Bush TOLD you that North Korea is part of the Axis of Evil
I'm listening to Bush - Hurricane (Album Version)
```

Character Classes with []

```
^[Ii] am
```

will match

```
i am so angry at my boyfriend i can't even bear to
look at him

i am boycotting the apple store

I am twittering from iPhone

I am a very vengeful person when you ruin my sweetheart.

I am so over this. I need food. Mmmm bacon...
```

Character Classes with []

Similarly, you can specify a range of letters [a-z] or [a-zA-Z]; notice that the order doesn't matter

```
^[0-9][a-zA-Z]
```

will match the lines

```
7th inning stretch
2nd half soon to begin. OSU did just win something
3am - cant sleep - too hot still.. :(
5ft 7 sent from heaven
1st sign of starvagtion
```

Character Classes with []

When used at the beginning of a character class, the "^" is also a metacharacter and indicates matching characters NOT in the indicated class

```
[^?.]$
```

will match the lines

```
i like basketballs
6 and 9
dont worry... we all die anyway!
Not in Baghdad
helicopter under water? hmmm
```

More Metacharacters

"." is used to refer to any character. So

```
9.11
```

will match the lines

```
its stupid the post 9-11 rules
if any 1 of us did 9/11 we would have been caught in days.
NetBios: scanning ip 203.169.114.66
Front Door 9:11:46 AM
Sings: 0118999881999119725...3 !
```

More Metacharacters: |

This does not mean “pipe” in the context of regular expressions; instead it translates to “or”; we can use it to combine two expressions, the subexpressions being called alternatives

```
flood|fire
```

will match the lines

```
is firewire like usb on none macs?
the global flood makes sense within the context of the bible
yeah ive had the fire on tonight
... and the floods, hurricanes, killer heatwaves, rednecks, gun nuts, etc.
```

More Metacharacters: |

We can include any number of alternatives...

```
flood|earthquake|hurricane|coldfire
```

will match the lines

```
Not a whole lot of hurricanes in the Arctic.
We do have earthquakes nearly every day somewhere in our State
hurricanes swirl in the other direction
coldfire is STRAIGHT!
'cause we keep getting earthquakes
```

More Metacharacters: |

The alternatives can be real expressions and not just literals

```
^[Gg]ood|[Bb]ad
```

will match the lines

```
good to hear some good knews from someone here
Good afternoon fellow american infidels!
good on you-what do you drive?
Katie... guess they had bad experiences...
my middle name is trouble, Miss Bad News
```

More Metacharacters: (and)

Subexpressions are often contained in parentheses to constrain the alternatives

```
^([Gg]ood|[Bb]ad)
```

will match the lines

```
bad habbit
bad coordination today
good, becuase there is nothing worse than a man in kinky underwear
Badcop, its because people want to use drugs
Good Monday Holiday
Good riddance to Limey
```

More Metacharacters: ?

The question mark indicates that the indicated expression is optional

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

will match the lines

```
i bet i can spell better than you and george bush combined
BBC reported that President George W. Bush claimed God told him to invade
a bird in the hand is worth two george bushes
```

One thing to note...

In the following

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

we wanted to match a "." as a literal period; to do that, we had to "escape" the metacharacter, preceding it with a backslash In general, we have to do this for any metacharacter we want to include in our match

More metacharacters: * and +

The * and + signs are metacharacters used to indicate repetition; * means "any number, including none, of the item" and + means "at least one of the item"

\(.*\)

will match the lines

```
anyone wanna chat? (24, m, germany)
hello, 20.m here... ( east area + drives + webcam )
(he means older men)
()
```

More metacharacters: * and +

The * and + signs are metacharacters used to indicate repetition; * means "any number, including none, of the item" and + means "at least one of the item"

```
[0-9]+ (.*)[0-9]+
```

will match the lines

```
working as MP here 720 MP battallion, 42nd birgade
so say 2 or 3 years at colleage and 4 at uni makes us 23 when and if we fi
it went down on several occasions for like, 3 or 4 *days*
Mmmm its time 4 me 2 go 2 bed
```

More metacharacters: { and }

{ and } are referred to as interval quantifiers; the let us specify the minimum and maximum number of matches of an expression

[Bb]ush(+[^]+){1,5} debate

will match the lines

```
Bush has historically won all major debates he's done.
in my view, Bush doesn't need these debates..
bush doesn't need the debates? maybe you are right
That's what Bush supporters are doing about the debate.
Felix, I don't disagree that Bush was poorly prepared for the debate.
indeed, but still, Bush should have taken the debate more seriously.
Keep repeating that Bush smirked and scowled during the debate
```

More metacharacters: m and n

{m, n}

- m,n means at least m but not more than n matches
- m means exactly m matches
- m, means at least m matches

More metacharacters: (and) revisited

- In most implementations of regular expressions, the parentheses not only limit the scope of alternatives divided by a "|", but also can be used to "remember" text matched by the subexpression enclosed
- We refer to the matched text with \1, \2, etc.

More metacharacters: (and) revisited

So the expression

```
 +([a-zA-Z]+) +\1 +
```

will match the lines

```
time for bed, night night twitter!

blah blah blah blah

my tattoo is so so itchy today

i was standing all all alone against the world outside...

hi anybody anybody at home

estudiando css css css css.... que desastritooooo
```

More metacharacters: (and) revisited

The * is "greedy" so it always matches the *longest* possible string that satisfies the regular expression. So

`^s(.*)s`

matches

```
sitting at starbucks

setting up mysql and rails

studying stuff for the exams

spaghetti with marshmallows

stop fighting with crackers

sore shoulders, stupid ergonomics
```

More metacharacters: (and) revisited

The greediness of * can be turned off with the ?, as in

```
^s(.*?)s$
```

Summary

- Regular expressions are used in many different languages;
- Regular expressions are composed of literals and metacharacters that represent sets or classes of characters/words
- Text processing via regular expressions is a very powerful way to extract data from "unfriendly" sources (not all data comes as a CSV file)

(Thanks to Mark Hansen for some material in this lecture.)

Regular Expression Functions

The primary R functions for dealing with regular expressions are

- `grep`, `grepl`: Search for matches of a regular expression/pattern in a character vector; either return the indices into the character vector that match, the strings that happen to match, or a TRUE/FALSE vector indicating which elements match
- `regexpr`, `gregexpr`: Search a character vector for regular expression matches and return the indices of the string where the match begins and the length of the match
- `sub`, `gsub`: Search a character vector for regular expression matches and replace that match with another string
- `regexec`: Easier to explain through demonstration.

grep

Here is an excerpt of the Baltimore City homicides dataset:

```
> homicides <- readLines("homicides.txt")
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltimore, MD
21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"

> homicides[1000]
[1] "39.33626300000, -76.55553990000, icon_homicide_shooting, 'p1200',...
```

How can I find the records for all the victims of shootings (as opposed to other causes)?

grep

```
> length(grep("iconHomicideShooting", homicides))
[1] 228
> length(grep("iconHomicideShooting|icon_homicide_shooting", homicides))
[1] 1003
> length(grep("Cause: shooting", homicides))
[1] 228
> length(grep("Cause: [Ss]hooting", homicides))
[1] 1003
> length(grep("[Ss]hooting", homicides))
[1] 1005
```

grep

```
> i <- grep("[cC]ause: [Ss]hooting", homicides)
> j <- grep("[Ss]hooting", homicides)
> str(i)
 int [1:1003] 1 2 6 7 8 9 10 11 12 13 ...
> str(j)
 int [1:1005] 1 2 6 7 8 9 10 11 12 13 ...
> setdiff(i, j)
integer(0)
> setdiff(j, i)
[1] 318 859
```

grep

```
> homicides[859]
[1] "39.33743900000, -76.66316500000, icon_homicide_bluntforce,
'p914', '<dl><dt><a href=\"http://essentials.baltimoresun.com/
micro_sun/homicides/victim/914/steven-harris\">Steven Harris</a>
</dt><dd class=\"address\">4200 Pimlico Road<br />Baltimore, MD 21215
</dd><dd>Race: Black<br />Gender: male<br />Age: 38 years old</dd>
<dd>Found on July 29, 2010</dd><dd>Victim died at Scene</dd>
<dd>Cause: Blunt Force</dd><dd class=\"popup-note\"><p>Harris was
found dead July 22 and ruled a shooting victim; an autopsy
subsequently showed that he had not been shot,...</dd></dl>'"
```

grep

By default, grep returns the indices into the character vector where the regex pattern matches.

```
> grep("^New", state.name)
[1] 29 30 31 32
```

Setting value = TRUE returns the actual elements of the character vector that match.

```
> grep("^New", state.name, value = TRUE)
[1] "New Hampshire" "New Jersey"    "New Mexico"    "New York"
```

grepl returns a logical vector indicating which element matches.

```
> grepl("^New", state.name)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALS
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALS
[25] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALS
[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALS
[49] FALSE FALSE
```

regexpr

Some limitations of `grep`

- The `grep` function tells you which strings in a character vector match a certain pattern but it doesn't tell you exactly where the match occurs or what the match is (for a more complicated regex.
- The `regexpr` function gives you the index into each string where the match begins and the length of the match for that string.
- `regexpr` only gives you the first match of the string (reading left to right). `gregexpr` will give you all of the matches in a given string.

"g" means global

regexpr

How can we find the date of the homicide?

```
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltimore,
MD 21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"
```

Can we just 'grep' on "Found"?

The word 'found' may be found elsewhere in the entry.

```
> homicides[954]
[1] "39.30677400000, -76.59891100000, icon_homicide_shooting, 'p816',
'<dl><dd class=\"address\">1400 N Caroline St<br />Baltimore, MD 21213</dd>
<dd>Race: Black<br />Gender: male<br />Age: 29 years old</dd>
<dd>Found on March  3, 2010</dd><dd>Victim died at Scene</dd>
<dd>Cause: Shooting</dd><dd class=\"popup-note\"><p>Wheeler\\'s body
was found on the grounds of Dr. Bernard Harris Sr. Elementary
School</p></dd></dl>'"
```

regexpr

Let's use the pattern

```
<dd>[F|f]ound(.*)</dd>
```

What does this look for?

```
> regexpr("<dd>[F|f]ound(.*)</dd>", homicides[1:10])
 [1] 177 178 188 189 178 182 178 187 182 183
attr(,"match.length")
 [1] 93 86 89 90 89 84 85 84 88 84
attr(,"useBytes")
[1] TRUE
> substr(homicides[1], 177, 177 + 93 - 1)
[1] "<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
 Trauma</dd><dd>Cause: shooting</dd>"
```

* Is greedy, so it matches the furthest </dd>

regexpr

The previous pattern was too greedy and matched too much of the string. We need to use the ? metacharacter to make the regex "lazy".

```
> regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:10])
 [1] 177 178 188 189 178 182 178 187 182 183
attr(,"match.length")
 [1] 33 33 33 33 33 33 33 33 33 33
attr(,"useBytes")
[1] TRUE
```

The indices of beginnings stay the same

The lengths of matches are much shorter

```
> substr(homicides[1], 177, 177 + 33 - 1)
[1] "<dd>Found on January 1, 2007</dd>"
```

regmatches

One handy function is `regmatches` which extracts the matches in the strings for you without you having to use `substr`.

```
> r <- regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:5])
> regmatches(homicides[1:5], r)
[1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January 2, 2007</dd>
[3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January 3, 2007</dd>
[5] "<dd>Found on January 5, 2007</dd>"
```

- regexpr finds the locations of matches
- regmatches gets the content from the found locations

sub/gsub

Sometimes we need to clean things up or modify strings by matching a pattern and replacing it with something else. For example, how can we extract the data from this string?

```
> x <- substr(homicides[1], 177, 177 + 33 - 1)
> x
[1] "<dd>Found on January 1, 2007</dd>"
```

We want to strip out the stuff surrounding the "January 1, 2007" piece.

```
> sub("<dd>[F|f]ound on |</dd>", "", x)
[1] "January 1, 2007</dd>"
```

Tips:
sub -- substitute the first
gsub -- global substitute

```
> gsub("<dd>[F|f]ound on |</dd>", "", x)
[1] "January 1, 2007"
```

sub/gsub

sub/gsub can take vector arguments

```
> r <- regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:5])
> m <- regmatches(homicides[1:5], r)
> m
[1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January 2, 2007</dd>
[3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January 3, 2007</dd>
[5] "<dd>Found on January 5, 2007</dd>"
> gsub("<dd>[F|f]ound on |</dd>", "", m)
[1] "January 1, 2007" "January 2, 2007" "January 2, 2007" "January 3, 2007
[5] "January 5, 2007"
> as.Date(d, "%B %d, %Y")
[1] "2007-01-01" "2007-01-02" "2007-01-02" "2007-01-03" "2007-01-05"
```

regexec

The `regexec` function works like `regexpr` except it gives you the indices for parenthesized sub-expressions.

```
> regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1])
[[1]]
[1] 177 190
attr(,"match.length")
[1] 33 15
```

[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon Nelson</dt><dd class=\"address\">3400 Clifton Ave.
Baltimore, MD 21216</dd><dd>black male, 17 years old</dd><dd>Found on January 1, 2007</dd><dd>Victim died at Shock Trauma</dd><dd>Cause: shooting</dd></dl>'"

```
> regexec("<dd>[F|f]ound on .*?</dd>", homicides[1])
[[1]]
[1] 177
attr(,"match.length")
[1] 33
```

regexec

Now we can extract the string in the parenthesized sub-expression.

```
> regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1])
[[1]]
[1] 177 190
attr(,"match.length")
[1] 33 15

> substr(homicides[1], 177, 177 + 33 - 1)
[1] "<dd>Found on January 1, 2007</dd>"

> substr(homicides[1], 190, 190 + 15 - 1)
[1] "January 1, 2007"
```

regexec

Even easier with the regmatches function.

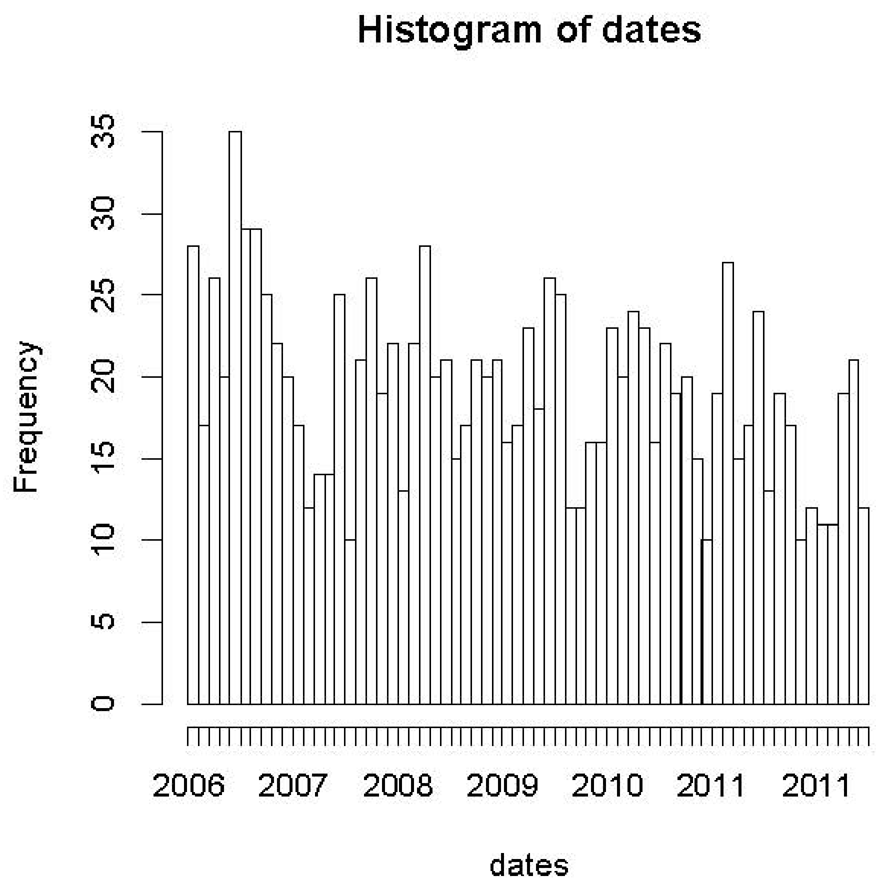
```
> r <- regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1:2])
> regmatches(homicides[1:2], r)
[[1]]
[1] "<dd>Found on January 1, 2007</dd>" "January 1, 2007"

[[2]]
[1] "<dd>Found on January 2, 2007</dd>" "January 2, 2007"
```

regexec

Let's make a plot of monthly homicide counts

```
> r <- regexec("<dd>[F|f]ound on (.*?)</dd>", homicides)
> m <- regmatches(homicides, r)
> dates <- sapply(m, function(x) x[2])
> dates <- as.Date(dates, "%B %d, %Y")
> hist(dates, "month", freq = TRUE)
```

regexec



Histogram of dates
Frequency
0
5
10
15
20
25
30
35
2006
2007
2008
2009
2010
2011
2011
dates

Summary

The primary R functions for dealing with regular expressions are

- `grep`, `grepl`: Search for matches of a regular expression/pattern in a character vector
- `regexpr`, `gregexpr`: Search a character vector for regular expression matches and return the indices where the match begins; useful in conjunction with `regmatches`
- `sub`, `gsub`: Search a character vector for regular expression matches and replace that match with another string
- `regexec`: Gives you indices of parethensized sub-expressions.

stringr package

- R provides a solid set of string operations, but because they have grown organically over time, they can be inconsistent and a little hard to learn.
- **stringr**: written by Hadley Wickham
  - Basic string operations
  - Pattern matching

Basic string operations

- str_c is equivalent to paste, but it uses the empty string (“”) as the default separator and silently removes zero length arguments.
- str_length is equivalent to nchar, but it preserves NA’ s (rather than giving them length 2) and converts factors to characters (not integers).
- str_sub is equivalent to substr but it returns a zero length vector if any of its inputs are zero length, and otherwise expands each argument to match the longest. It also accepts negative positions, which are calculated from the left of the last character. The end position defaults to -1, which corresponds to the last character.
- str_sub<- is equivalent to substr<-, but like str_sub it understands negative indices, and replacement strings not do need to be the same length as the string they are replacing.

Basic string operations

- str_dup to duplicate the characters within a string.
- str_trim to remove leading and trailing whitespace.
- str_pad to pad a string with extra whitespace on the left, right, or both sides.

Pattern matching

- str_detect detects the presence or absence of a pattern and returns a logical vector. Based on grepl.
- str_locate locates the first position of a pattern and returns a numeric matrix with columns start and end. str_locate_all locates all matches, returning a list of numeric matrices. Based on regexpr and gregexpr.
- str_extract extracts text corresponding to the first match, returning a character vector.
- str_extract_all extracts all matches and returns a list of character vectors.

Pattern matching

- str_match extracts capture groups formed by () from the first match. It returns a character matrix with one column for the complete match and one column for each group.
- str_match_all extracts capture groups from all matches and returns a list of character matrices.
- str_replace replaces the first matched pattern and returns a character vector.
- str_replace_all replaces all matches. Based on sub and gsub.
- str_split_fixed splits the string into a fixed number of pieces based on a pattern and returns a character matrix. str_split splits a string into a variable number of pieces and returnsa list of character vectors.

Pattern Match

- **Arguments**: Each pattern matching function has the same first two arguments, a character vector of strings to process and a single pattern (regular expression) to match.
  - The replace functions have an additional argument specifying the replacement string,
  - The split functions have an argument to specify the number of pieces.
- "When writing regular expressions, I strongly recommend generating a list of positive (pattern should match) and negative (pattern shouldn't match) test cases to ensure that you are matching the correct components." -- Hadley Wickham

Other resources

- A good reference sheet
  - http://www.regular-expressions.info/reference.html
- A tool that allows you to interactively test what a regular expression will match
  - https://regex101.com/
  - http://regexr.com/

Introduction to Natural Language Processing

- Bayes Classifier
- Text Similarity

Text/Document Representations

- Document set

$$D = \{d_1, d_2, ..., d_n\}$$

- These documents have a "bag-of-words" or the feature set

$$X = \{x_1, x_2, ..., x_m\}$$

- The class set is

$$C = \{c_1, c_2, c_k\}.$$

- Assumption: the features in a dataset are mutually independent

$$P(x_1, x_2, ..., x_k | C) = \prod_{i=1}^{k} P(x_i | C)$$

Text/Document Representations

```
vocab = ['blue', 'red', 'dog', 'cat', 'biscuit', 'apple']
doc = "the blue dog ate a blue biscuit"

# note that the words that didn't appear in the vocabulary will be discarded
bernoulli = [1 if v in doc else 0 for v in vocab]
multinomial = [doc.count(v) for v in vocab]
print('bernoulli', bernoulli)
print('multinomial', multinomial)
```

```
bernoulli [1, 0, 1, 0, 1, 0]
multinomial [2, 0, 1, 0, 1, 0]
```

Naïve Bayes Theorem

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- In NLP, Bayes theorem can be rewritten to

$$p(C = k|D) = \frac{p(C = k)\, p(D|C = k)}{p(D)} \propto p(C = k)\, p(D|C = k)$$

- $\propto$ means is proportional to
- p(C=k) represents the class k's **prior** probabilities.
- p(D|C=k) is the **likelihoods** of the document given the class k.
- p(D) is the **normalizing factor** which we don't have to compute since it does not depend on the class C.

Bernoulli Model

- To calculate the probability of observing features X1 through Xd, given some class C

$$p(x_1, x_2, \ldots, x_d \mid C) = \prod_{i=1}^{d} p(x_i \mid C)$$

- **Bernoulli Model**

$$p(D_i \mid C) = \prod_{t=1}^{d} b_{it} p(w_t \mid C) + (1 - b_{it})(1 - p(w_t \mid C))$$

Where:

- $p(w_t \mid C)$ is the probability of word w_t occurring in a document of class C.
- $1 - p(w_t \mid C)$ is the probability of w_t not occurring in a document of class C.
- b_{it} is either 0 or 1 representing the absence or presence of word w_t in the i_{th} document.

Bernoulli Model

- Estimate $p(w_t \mid C)$ and $p(C)$

$$p(w_t \mid C = k) = \frac{n_k(w_t)}{N_k}$$

Where:

- $n_k(w_t)$ is the number of class $C = k$'s document in which w_t is observed.
- N_k is the number of documents that belongs to class k.

$$p(C = k) = \frac{N_k}{N}$$

Where N is the total number of documents in the training set.

Multinomial Model

- In the multinomial case, calculating $p(D|C=k)$ for the i_{th} document becomes

$$p(D_i|C=k) = \frac{x_i!}{\prod_{t=1}^{d} x_{it}!} \prod_{t=1}^{d} p(w_t|C)^{x_{it}} \propto \prod_{t=1}^{d} p(w_t|C)^{x_{it}}$$

Where:

- x_{it}, is the count of the number of times word w_t occurs in document D_i.
- $x_i = \sum_t x_{it}$ is the total number of words in document D_i.
- Often times, we don't need the normalization term $\frac{x_i!}{\prod_{t=1}^{d} x_{it}!}$, because it does not depend on the class, C.
- $p(w_t \mid C)$ is the probability of word w_t occurring in a document of class C. This time estimated using the word frequency information from the document's feature vectors. More specifically, this is: $\text{Number of word } w_t \text{ in class } C / \text{Total number of words in class } C$.
- $\prod_{t=1}^{d} p(w_t|C)^{x_{it}}$ can be interpreted as the product of word likelihoods for each word in the document.

Laplace Smoothing

- What if $p(w_t \mid C)$ is equal to 0? We add a count of one to each word type

$$p(w_t \mid C) = \frac{(\text{Number of word } w_t \text{ in class } C + 1)}{(\text{Total number of words in class } C) + |V|}$$

Log-Transformation

- Our original formula for classifying a document in to a class using Multinomial Naive Bayes was,

$$p(C|D) = p(C) \prod_{t=1}^{d} p(w_t|C)^{x_{it}}$$

- To prevent the small values from being rounded to zero, we can simply apply a log around everything,

$$p(C|D) = log\left(p(C) \prod_{t=1}^{d} p(w_t|C)^{x_{it}}\right)$$

- Which becomes,

$$p(C|D) = log\, p(C) + \sum_{t=1}^{d} x_{it} log\, p(w_t|C)$$

Example

$$P(c) = \frac{N_c}{N}$$

$$P(w \mid c) = \frac{count(w,c)+1}{count(c)+|V|}$$

	Doc	Words	Class
Training	1	Chinese Beijing Chinese	c
	2	Chinese Chinese Shanghai	c
	3	Chinese Macao	c
	4	Tokyo Japan Chinese	j
Test	5	Chinese Chinese Chinese Tokyo Japan	?

Priors:

$P(c) = \frac{3}{4}$

$P(j) = \frac{1}{4}$

Conditional Probabilities:

P(Chinese|*c*) = (5+1) / (8+6) = 6/14 = 3/7

P(Tokyo|*c*) = (0+1) / (8+6) = 1/14

P(Japan|*c*) = (0+1) / (8+6) = 1/14

P(Chinese|*j*) = (1+1) / (3+6) = 2/9

P(Tokyo|*j*) = (1+1) / (3+6) = 2/9

P(Japan|*j*) = (1+1) / (3+6) = 2/9

Choosing a class:

$P(c|d5) \propto 3/4 * (3/7)^3 * 1/14 * 1/14 \approx 0.0003$

$P(j|d5) \propto 1/4 * (2/9)^3 * 2/9 * 2/9 \approx 0.0001$

Applications and Use Cases

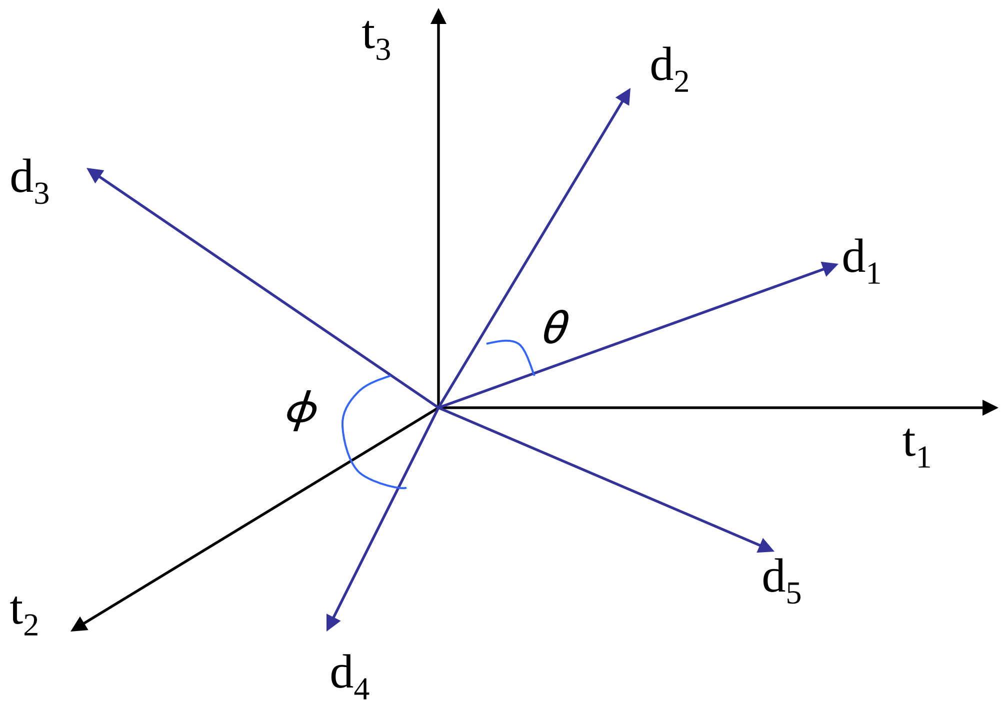
- News and sentiment analysis
- Social media network content analysis
- Marketing
- Customer Services
- Spam email detection
- Advertisement matching by Google AdSense
- Legal documents
- etc

Text Similarity

Question: measure how similar the documents are irrespective of their size

Use case: How to find all job posts that fit my resume?

Intuition



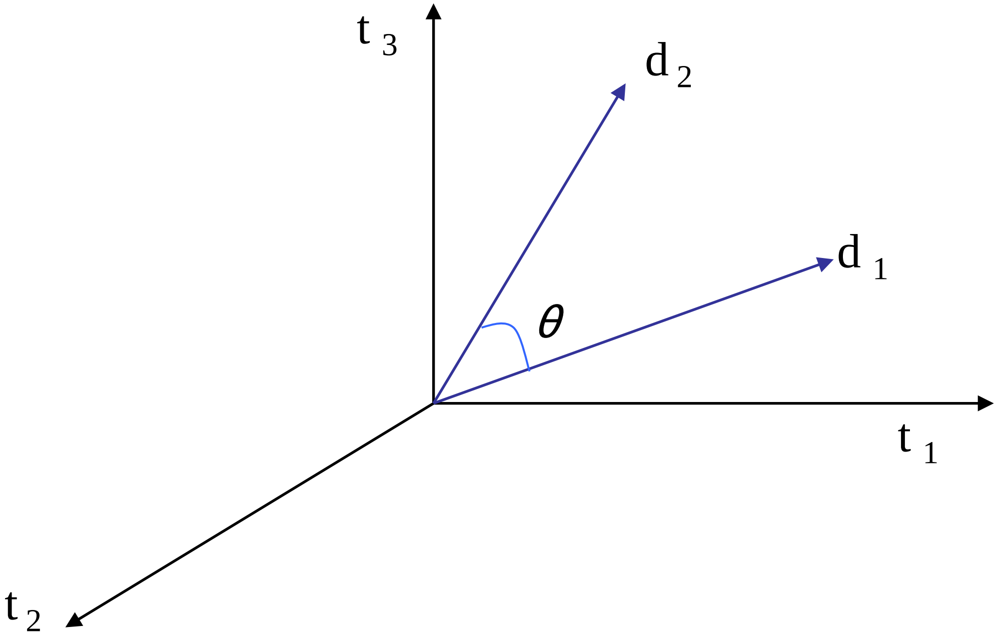
Postulate: Documents that are “close together” in the vector space talk about the same things.

First cut

- Distance between d_1 and d_2 is the length of the vector $|d_1 - d_2|$.
  - Euclidean distance
- Why is this not a great idea?
- We still haven't dealt with the issue of length normalization
  - Long documents would be more similar to each other by virtue of length, not topic
- However, we can implicitly normalize by looking at *angles* instead

Cosine similarity

- Distance between vectors d_1 and d_2 *captured* by the cosine of the angle x between them.
- Note – this is *similarity*, not distance
  - No triangle inequality for similarity.



Cosine Similarity

- With cosine similarity we can measure the similarity between two document vectors.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

- If the cosine similarity is 1, they are the same document.
- If it is 0, the documents share nothing.
- This is because term frequency cannot be negative so the angle between the two vectors cannot be greater than 90°
- It removes any bias we had towards longer documents.