Lec 6: Data Wrangling And Working With Strings

Key ideas: regular expressions, sed/awk/grep, working with text
Data Science Pipeline

Raw Data
- Text
- Images
- Sounds
- Sensor Readings
- CSV
- Log files
- Web Forms

Structure Extraction
- Image Segmentation
- Signal Processing
- Regularization
- Re-Sampling
- Cleaning
- Outlier Removal

Transformation/Integration

Stored Data

Tables

Querying/Processing

Visualization/Presentation

Modeling/Prediction
DATA WRANGLING

[Sean Kandel et al: Research directions in data wrangling: Visualizations and transformations for usable and credible data, Information Visualization, 2011]
THREE POWERFUL TOOLS

1) grep – find text matching a regular expression
   Basic syntax:
   
   grep 'regexp' filename
   
   or equivalently (using UNIX pipelining):
   
   cat filename | grep 'regexp'

2) sed – stream editor

3) awk – general purpose text processing language
A regular expression (regex) describes a set of possible input strings.

*Regular expressions* descend from a fundamental concept in Computer Science called *finite automata theory*.

*Regular expressions* are used in many *nix tools:
- vi, ed, sed, and emacs
- awk, tcl, perl and Python
- grep, egrep, fgrep
- compilers
REGULAR EXPRESSIONS

The simplest regular expressions are a string of literal characters to match.

The string *matches* the regular expression if it contains the substring.
Unix rocks.

UNIX sucks.

UNIX is okay.
A regular expression can match a string in more than one place.
The . regular expression can be used to match any character.

For me to open

match 1

match 2
REPETITION

The * is used to define zero or more occurrences of the single regular expression preceding it.

+ Matches one or more occurrences
I got mail, yaaaaaaay!

I sat on the stoop
REPETITION RANGES

Ranges can also be specified

- `{ }` notation can specify a range of repetitions for the immediately preceding regex
- `{n}` means exactly $n$ occurrences
- `{n,}` means at least $n$ occurrences
- `{n,m}` means at least $n$ occurrences but no more than $m$ occurrences

Example:

- `. {0,}` same as `.*`
- `a{2,}` same as `aaa*`
OR

\(a|b^*\) denotes \(\{\varepsilon, "a", "b", "bb", "bbb", \ldots\}\)

\((a|b)^*\) denotes the set of all strings with no symbols other than "a" and "b", including the empty string: \(\{\varepsilon, "a", "b", "aa", "ab", "ba", "bb", "aaa", \ldots\}\)

\(ab^*(c)\) denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": \("a", "ac", "ab", "abc", "abb", "abbc", \ldots\)
Character classes \([\ ]\) can be used to match any specific set of characters.
Character classes can be negated with the \[^\] syntax.

Example:

- Regular expression: `b[^eo]at`
- Input: `beat a brat on a boat`
- Output: Match
MORE ABOUT CHARACTER CLASSES

- \([aeiou]\) will match any of the characters \(a, e, i, o,\) or \(u\)
- \([kK]orn\) will match \(korn\) or \(Korn\)

Ranges can also be specified in character classes

- \([1–9]\) is the same as \([123456789]\)
- \([abcde]\) is equivalent to \([a–e]\)
- You can also combine multiple ranges
  - \([abcde123456789]\) is equivalent to \([a–e1–9]\)
- Note that the \(-\) character has a special meaning in a character class \textit{but only} if it is used within a range, \([-123]\) would match the characters \(-, 1, 2,\) or \(3\)
NAMED CHARACTER CLASSES

Commonly used character classes can be referred to by name \(\texttt{(alpha, lower, upper, alnum, digit, punct, cntrl)}\)

Syntax \[
\text{[:name:]}\]

- \([\texttt{a-zA-Z}]\)  \[[[:alpha:]]\]
- \([\texttt{a-zA-Z0-9}]\)  \[[[:alnum:]]\]
- \([\texttt{45a-z}]\)  \[[45[:lower:]]\]

Important for portability across languages
ANCHORS

Anchors are used to match at the beginning or end of a line (or both).

^ means beginning of the line

$ means end of the line
beat a brat on a boat

match

regular expression

^ b [eor] a t

beat a brat on a boat

match

regular expression

b [eor] a t $

beat a brat on a boat

match

^word$ ^$
MATCH LENGTH

By default, a match will be the longest string that satisfies the regular expression.
MATCH LENGTH

Append a ? to match the shortest string possible:

regular expression → a . * ? e

Scrap[ple from the apple. yes

no no
PRACTICAL REGEX EXAMPLES

Dollar amount with optional cents

• \$[0-9]+(\ .[0-9][0-9])?

Time of day

• (1[012]|1[0-9]):[0-5][0-9] (am|pm)

HTML headers <h1> <H1> <h2> ...

• <[hH] [1-4]>
GREP

- grep comes from the ed (Unix text editor) search command “global regular expression print” or g/re/p
- This was such a useful command that it was written as a standalone utility
- There are two other variants, egrep and fgrep that comprise the grep family
- *grep* is the answer to the moments where you know you want the file that contains a specific phrase but you can’t remember its name
"text": "RT @TwitterMktg: Starting today, businesses can request and share locations when engaging with people in Direct Messages. https://t.co/rpYn...",

"text": "Starting today, businesses can request and share locations when engaging with people in Direct Messages. https://t.co/rpYndqWfQw", 

```bash
grep "\"text\": ".*location.*\"" twitter.json
```
BACKREFERENCES

Used to refer to a match that made earlier in a regex

• \n is a backreference specifier, where \n is a number
Matches the \n\n\nth subexpression specified by (…)

E.g., to find if the first word of a line is the same as the last:

• ^([[:alpha:]]+) .* \1$

Here,

[[[:alpha:]]+] matches 1 or more letters
([[:alpha:]]+) is the first subexpression
\1 matches the first subexpression
FORMALLY

Regular expressions are “regular” because they can only express languages accepted by finite automata. Backreferences allow you to do much more.

\[
\{a^n b^n : n \geq 0\} \\
\{ww^R : w \in \{a,b\}^*\}
\]

Non-regular languages

Regular languages

\[
a^* b \\
b^* c + a \\
b + c(a+b)^* \\
etc...
\]

See: https://link.springer.com/article/10.1007%2Fs00224-012-9389-0
BACKREFERENCE TRICKS

Can you find a regex to match \( L=ww \; w \in \{a,b\}^* \)

e.g., aa, bb, abab, or abbabb

Cannot be expressed with a FA, because need to revisit the tokens in \( w \) exactly once, and \( w \) is an unknown length.

\( ([ab]^*)\1 \)
def f(n):
    s = "x" * n
    return re.match("^x?$|^(xx+?)\1+$", s)

Generates a string of length n, to test if n is prime

`^x?$`  
*base case:* 0 and 1 are not prime

`(? matches preceding character 0 or 1 times)`

`|`  
*or*

`^(xx+?)\1+$`  
*two or more xs*

*repeated on or more times, followed by $*

A prime is a number that cannot be factored. If we find a sequence of N xs that repeats two or more times without any xs left over, we know N is a factor, and the number is not prime.

**Example:**

- Doesn’t match, can’t consume all xs with repeated pattern, ==> Prime
- Matches, we consume all xs with 3x repeated pattern, ==> Not Prime
CLICKER QUESTION

Select the string for which the regular expression ‘..\.19..’ would find a match:

a) “12.1000”
b) “123.1900”
c) “12.2000”
d) the regular expression does not match any of the strings above
CLICKER QUESTION

Choose the pattern that finds all filenames in which
1. the first letters of the filename are chap,
2. followed by two digits,
3. followed by some additional text,
4. and ending with a file extension of .doc

For example: chap23Production.doc

a) chap[0-9]*.doc
b) chap*[0-9]doc
c) chap[0-9][0-9].*\doc
d) chap*doc
THREE POWERFUL TOOLS

1) grep
Basic syntax:
   grep 'regexp' filename
or equivalently (using UNIX pipelining):
   cat filename | grep 'regexp'

2) sed – stream editor
Basic syntax
   sed 's/regexp/replacement/g' filename
For each line in the input, the portion of the line that matches regexp (if any) is replaced with replacement.
Sed is quite powerful within the limits of operating on single line at a time.
You can use \( \) to refer to parts of the pattern match.
SED EXAMPLE

File = Trump is the president. His job is to tweet.

```
 sed 's/Trump/Biden/g' file
```

```
 sed 's/(His job is to\).*\l run the country./g' file
```

Biden is the president. His job is to tweet.
Trump is the president. His job is to run the country.
COMBINING TOOLS

Suppose we want to extract all the “screen_name” fields from twitter data

```json
[{
  "created_at": "Thu Apr 06 15:28:43 +0000 2017",
  "id": 850007368138018817,
  "id_str": "850007368138018817",
  "text": "RT @TwitterDev: 1/ Today we’re sharing our vision for the future of the Twitter API platform!nhttps://t.co/XweGngmx1P",
  "truncated": false,
  ...
  "user": {
    "id": 6253282,
    "id_str": "6253282",
    "name": "Twitter API",
    "screen_name": "twitterapi",
  }
}
```

grep \"screen_name\": twitter.json | sed 's/\[[ ]*\"screen_name\": \"(.\*\\)\",//1/g'
COMBINING TOOLS

Suppose we want to extract all the “screen_name” fields from twitter data

```json
[
  {
    "created_at": "Thu Apr 06 15:28:43 +0000 2017",
    "id": 850007368138018817,
    "id_str": "850007368138018817",
    "text": "RT @TwitterDev: 1/ Today we’re sharing our vision for the future of the Twitter API platform!nhttps://t.co/XweGngmx1P",
    "truncated": false,
  ...
  "user": {
    "id": 6253282,
    "id_str": "6253282",
    "name": "Twitter API",
    "screen_name": "twitterapi",
  }
]
```

grep \"screen_name\": twitter.json |
sed 's/[ ]*"screen_name": \"(.*)\"",/\1/g'
EXAMPLE 2: LOG PARSING

192.168.2.20 - - [28/Jul/2006:10:27:10 -0300] "GET /cgi-bin/try/ HTTP/1.0" 200 3395
127.0.0.1 - - [28/Jul/2006:10:22:04 -0300] "GET / HTTP/1.0" 200 2216

```
```

<table>
<thead>
<tr>
<th>IP Address</th>
<th>Stuff</th>
<th>URL up to quote</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.2.20</td>
<td>GET /cgi-bin/try/</td>
<td>HTTP/1.0</td>
</tr>
<tr>
<td>127.0.0.1</td>
<td>GET /</td>
<td>HTTP/1.0</td>
</tr>
</tbody>
</table>
Awk

Finally, awk is a powerful scripting language (not unlike perl). The basic syntax of awk is:

```
awk -F',' 'BEGIN{commands}
   /regexp1/ {command1} /regexp2/ {command2}
   END{commands}'
```

- For each line, the regular expressions are matched in order, and if there is a match, the corresponding command is executed (multiple commands may be executed for the same line).
- BEGIN and END are both optional.
- The -F',' specifies that the lines should be split into fields using the separator "","", and those fields are available to the regular expressions and the commands as $1, $2, etc.
- See the manual (man awk) or online resources for further details.
AWK COMMANDS

{ print $1 }  – Match any line, print the 1st field

$1=="Obama"{print $2}'

   If the first field is “Obama”, print the 2nd field

'$0 ~/Obama/ {t = gsub("Obama","Trump","g", $0); print t}'

   If the line contains Obama, globally replace “Trump” for ”Obama” and assign the result to the variable “txt”. Then print it.

Awk commands:

WRANGLING IN AWK

Input data

Reported crime in Alabama,
2004, 4029.3
2005, 3900
2006, 3937
2007, 3974.9
2008, 4081.9

Reported crime in Alaska,
2004, 3370.9
2005, 3615
2006, 3582
2007, 3373.9
2008, 2928.3

Reported crime in Arizona,
2004, 5073.3
2005, 4827

Desired Output:

2004, Alabama, 4029.3
2005, Alabama, 3900
2006, Alabama, 3937
2007, Alabama, 3974.9
2008, Alabama, 4081.9

2004, Alaska, 3370.9
2005, Alaska, 3615
2006, Alaska, 3582
2007, Alaska, 3373.9
2008, Alaska, 2928.3

2004, Arizona, 5073.3
2005, Arizona, 4827
2006, Arizona, 4741.6
2007, Arizona, 4502.6
2008, Arizona, 4087.3

2004, Arkansas, 4033.1
2005, Arkansas, 4068
BEGIN {FS="[, ,]

$1=="Reported" {
state = "$4" ""$5;
gsub(/[ 	]+$/,, "", state)
}

$1 ~ 20 {print $1","state","$2

Reported crime in Alabama,
, 2004,4029.3
2005,3900
2006,3937
2007,3974.9
2008,4081.9

line begins w/ reported
some states are two words
strip trailing spaces

line begins with 20
print year, state, and amount
DATA WRANGLER / TRIFACTA

http://vis.stanford.edu/wrangler/app/

<table>
<thead>
<tr>
<th>Event_ID</th>
<th>User_Email</th>
<th>Access_Date</th>
<th>Screen_Detail</th>
<th>Device_Manufacturer</th>
<th>Device_OS_Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC1000048004</td>
<td>lucius.vulpete.niss@felish</td>
<td>2012-09-13</td>
<td>00:00</td>
<td>samsung</td>
<td>Android 4.3</td>
</tr>
<tr>
<td>DC1000048005</td>
<td><a href="mailto:velita@Nunculpinvar.edu">velita@Nunculpinvar.edu</a></td>
<td>2012-10-17</td>
<td>23:00</td>
<td>samsung</td>
<td>Windows Phone 7.5</td>
</tr>
<tr>
<td>DC1000048006</td>
<td>nunc.risus.varius@nullvulpu</td>
<td>2012-11-28</td>
<td>04:17:49</td>
<td>Motorola</td>
<td>Android 4.0.2</td>
</tr>
<tr>
<td>DC1000048007</td>
<td>fermentum.vel@ruppiaeauct</td>
<td>2012-10-15</td>
<td>10:43:16</td>
<td>Samsung</td>
<td>Windows Mobile 6.1</td>
</tr>
<tr>
<td>DC1000048008</td>
<td>volup@pere@pure@eucteinc</td>
<td>2012-10-14</td>
<td>08:22:33</td>
<td>Nokia</td>
<td>iOS 6.1.3</td>
</tr>
<tr>
<td>DC1000048009</td>
<td>Duis.elementum@<a href="mailto:rius@e.net">rius@e.net</a></td>
<td>2012-11-03</td>
<td>14:56:07</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048010</td>
<td>non.ancii.Vivamus@<a href="mailto:proin@si.e">proin@si.e</a></td>
<td>2012-10-23</td>
<td>17:16:43</td>
<td>Nokia</td>
<td>iOS 6.1.3</td>
</tr>
<tr>
<td>DC1000048011</td>
<td><a href="mailto:nec@dictum.ca">nec@dictum.ca</a></td>
<td>2012-11-18</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.0.2</td>
</tr>
<tr>
<td>DC1000048012</td>
<td>Aenean@<a href="mailto:Pvivamus@i.com">Pvivamus@i.com</a></td>
<td>2012-09-27</td>
<td>23:00</td>
<td>Samsung</td>
<td>Windows Mobile 6.1</td>
</tr>
<tr>
<td>DC1000048013</td>
<td>in.hendrerit.consectetuer@eau</td>
<td>2012-10-17</td>
<td>23:00</td>
<td>Nokia</td>
<td>iOS 6.1.3</td>
</tr>
<tr>
<td>DC1000048014</td>
<td><a href="mailto:urma.Nunc@iac.com">urma.Nunc@iac.com</a></td>
<td>2012-10-22</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048015</td>
<td><a href="mailto:Fauci@bus.lectus">Fauci@bus.lectus</a>@por@titor@er@o</td>
<td>2012-11-12</td>
<td>23:00</td>
<td>Samsung</td>
<td>iOS 6.1.3</td>
</tr>
<tr>
<td>DC1000048016</td>
<td><a href="mailto:Done@net.org">Done@net.org</a></td>
<td>2012-12-19</td>
<td>23:00</td>
<td>Samsung</td>
<td>iOS 6.1.3</td>
</tr>
<tr>
<td>DC1000048017</td>
<td><a href="mailto:lobortis@sed.ca">lobortis@sed.ca</a></td>
<td>2012-10-12</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048018</td>
<td>amet.ritus.Done@integr@tine</td>
<td>2012-12-16</td>
<td>23:00</td>
<td>Samsung</td>
<td>iOS 6.1.3</td>
</tr>
<tr>
<td>DC1000048019</td>
<td>mollis@pur@ipulla</td>
<td>2012-10-26</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048020</td>
<td><a href="mailto:orci.adipiscing.nonmas@co.a">orci.adipiscing.nonmas@co.a</a></td>
<td>2012-11-03</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048021</td>
<td>blandit@PhaseIiusonore@Fusce</td>
<td>2012-09-14</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048022</td>
<td>tincidunt.adipiscing.Ma@uris</td>
<td>2012-10-13</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048023</td>
<td>vel@Robert@ius@<a href="mailto:spec@e.net">spec@e.net</a></td>
<td>2012-11-11</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048024</td>
<td><a href="mailto:Nulla.ae.nuque@mollis.ca">Nulla.ae.nuque@mollis.ca</a></td>
<td>2012-11-28</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048025</td>
<td>fringilla@<a href="mailto:eum@lact.org">eum@lact.org</a></td>
<td>2012-10-08</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048026</td>
<td><a href="mailto:fauci@bus.lectus">fauci@bus.lectus</a>@noc@torn@mocc@i</td>
<td>2012-11-14</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048027</td>
<td>nisi.Cum@Don@ces@&amp;@m@<a href="mailto:uri@is.com">uri@is.com</a></td>
<td>2012-10-16</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048028</td>
<td>parturient.montes.nascetur@p</td>
<td>2012-10-23</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
<tr>
<td>DC1000048029</td>
<td>nisi.quisque.fringilla@conse</td>
<td>2012-10-31</td>
<td>23:00</td>
<td>Samsung</td>
<td>Android 4.2</td>
</tr>
</tbody>
</table>

**SCRIPT**

highlight row: (date(2012, 11, 7) <= Access_Date) & (Access_Date < date(2012, 12, 27))

**SUGGESTED TRANSFORMS**

update row: (date(2012, 11, 7) <= Access_Date) & (Access_Date < date(2012, 12, 27))

delete row: (date(2012, 11, 7) <= Access_Date) & (Access_Date < date(2012, 12, 27))

keep row: (date(2012, 11, 7) <= Access_Date) & (Access_Date < date(2012, 12, 27))

rename col: column2 to 'Access_Date'
BREAK
WORKING WITH TEXT
TEXT AS DATA

What might we want to do?

Find similar documents
   E.g., for document clustering
Find similarity between a document and a string
   E.g., for document search

Answer questions from documents
Assess document sentiment
Extract information from documents

Focus today: Given two pieces of text, how do we measure similarity?
TOKENIZATION

**Input**: “Friends, Romans and Countrymen”

**Output**: Tokens

- Friends
- Romans
- and
- Countrymen

A **token** is an instance of a sequence of characters.

What are valid tokens?

Typically just words, but can be complicated.

*E.g.*, how many tokens is *Lebensversicherungsgesellschaftsangestellter*, meaning ‘life insurance company employee’ in German?
WHY TOKENIZE?

• Often useful to think of text as a bag of words, or as a table of words and their frequencies

• Need a standard way to define a word, and correct for differences in formatting, etc.

• Very common in information retrieval (IR) / keyword search
  • Typical goal: find similar documents based on their words or n-grams (length n word groups)
DOCUMENT SIMILARITY EXAMPLE

Suppose we have the following strings, and want to measure their similarity?

```python
sen = [
    "Tim loves the band Korn.",
    "Tim adores the rock group Korn.",
    "Tim loves eating corn.",
    "Tim used to love Korn, but now he hates them.",
    "Tim absolutely loves Korn.",
    "Tim completely detests the performers named Korn",
    "Tim has a deep passion for the outfit the goes by the name of Korn",
    "Tim loves listening to the band Korn while eating corn."
]
```
BAG-OF-WORDS MODEL

• Treat documents as sets
• Measure similarity of sets

Standard set similarity metric: Jaccard Similarity

\[ sim(s_1, s_2) = \frac{s_1 \cap s_2}{s_1 \cup s_2} \]

\[ sim(\{\text{tim}, \text{loves}, \text{korn}\}, \{\text{tim}, \text{loves}, \text{eating}, \text{corn}\}) = \frac{2}{5} \]
\[ sim(\{\text{tim}, \text{absolutely}, \text{adores}, \text{the}, \text{band}, \text{korn}\}, \{\text{tim}, \text{loves}, \text{korn}\}) = \frac{2}{7} \]

Problems:

All words weighted equally
Same word with different suffix treated differently (e.g., love & loves)
Semantic significance ignored (e.g., adores & loves are the same)
Duplicates are ignored (“Tim really, really loves Korn”)
STOP WORDS

With a stop list, you exclude from the dictionary entirely the commonest words. Intuition:

- They have little semantic content: *the, a, and, to, be*
- There are a lot of them: ~30% of postings for top 30 words

Sometimes you want to include them, as they affect meaning

- Phrase queries: “King of Denmark”
- Various song titles, etc.: “Let it be”, “To be or not to be”
- “Relational” queries: “flights to London”
STOP WORDS IN PYTHON

```python
from nltk.corpus import stopwords
print(stopwords.words('english'))
```

['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', 'she's', 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
STEMMING

- Reduce terms to their “roots” before indexing
- “Stemming” performs crude affix chopping
  - language dependent
  - e.g., automate(s), automatic, automation all reduced to automat.

for example compressed and compression are both accepted as equivalent to compress.

for exampl compress and compress ar both accept as equivil to compress
PORTER’S ALGORITHM

Most common algorithm for stemming English

• Other options exist, e.g., snowball
Conventions + 5 phases of reductions

• phases applied sequentially
• each phase consists of a set of commands
• sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*
TYPICAL RULES IN PORTER

sses → ss
ies → i
ational → ate
tional → tion

Weight of word sensitive rules

\(m > 1\) EMENT →

• replacement → replac
• cement → cement
import nltk.stem.porter

stemmer = nltk.stem.porter.PorterStemmer()
for w in sen[0].split(" "):  
    print(stemmer.stem(w))

tim
love
the
band
korn
STEP WORDS + STEMMING

sen = [
    "Tim loves the band Korn.",
    "Tim adores the rock group Korn.",
    "Tim loves eating corn.",
    "Tim used to love Korn, but now he hates them.",
    "Tim absolutely loves Korn.",
    "Tim completely detests the performers named Korn",
    "Tim has a deep passion for the outfit the goes by the name of Korn",
    "Tim loves listening to the band Korn while eating corn."
]

tim love band korn
tim ador rock group korn
tim love eat corn
tim use love korn hate
tim absolut love korn
tim complet detest perform name korn
tim deep passion outfit goe korn
tim love listen band korn eat corn
COSINE SIMILARITY

Given two vectors, a standard way to measure how similar they are

\[ \cos(v_1, v_2) = \text{closeness of two vectors (smaller is closer)} \]

\[ \cos(\theta) = \frac{V_1 \cdot V_2}{||V_1|| \times ||V_2||} \]

\[ \cos(\theta) = \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \end{bmatrix} / (\sqrt{5})^2 \]

\[ \arccos\left(\frac{4}{5}\right) = 36.8^\circ \]

\[ ||V_1|| = 2.01, \quad ||V_2|| = 2.02 \]

\[ \cos(\theta) = \begin{bmatrix} 0.2 & 2 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0.3 \end{bmatrix} / 2.015 \]

\[ = 1/2.015 \]

\[ \arccos\left(\frac{1}{2.015}\right) = 60.2^\circ \]
COSINE SIMILARITY OF WORD VECTORS

\[ \cos(\Theta) = \frac{V_1 \cdot V_2}{\|V_1\| \times \|V_2\|} \]

1 2 3
S1 = Tim loves Korn
4 5 6
S2 = Tim loves eating corn

V1 = 1 1 1 0 0 0
V2 = 1 0 0 1 1 1

V1 \cdot V2 = 1
\|V1\| = \sqrt{3}
\|V2\| = \sqrt{4}

1 / \sqrt{3} \times \sqrt{4} = .29

1 2 3
S1 = Tim loves Korn
4 5 6 7 8
S2 = Tim absolutely adores the band Korn

V1 = 1 1 1 0 0 0 0
V2 = 1 0 1 1 1 1 1

V1 \cdot V2 = 2
\|V1\| = \sqrt{3}
\|V2\| = \sqrt{7}

2 / \sqrt{3} \times \sqrt{7} = .43

Typically, when using cosine similarity, we don’t take the acos of the values (since acos is expensive)
JACCARD VS COSINE

\[ S_1 = \text{Tim loves Korn} \]
\[ S_2 = \text{Tim loves eating corn} \]

\[ \text{CosSim}(S_1, S_2) = .29 \]
\[ \text{Jaccard}(S_1, S_2) = .4 \]

\[ S_3 = \text{Tim absolutely adores the band Korn} \]
\[ \text{CosSim}(S_1, S_3) = .43 \]
\[ \text{Jaccard}(S_1, S_3) = .28 \]

*Jaccard more sensitive to different document lengths than CosSim*

*CosSim can incorporate repeated words (by using non-binary vectors)*
Consider two sentences:

Sam loves limp bizkit
Sam eats limp biscuits

What is their Jaccard similarity?
A. 4/6
B. 2/8
C. 2/6
D. Something else

What is their Cosine similarity?
A. 1/4
B. 2/4
C. 4/6
D. Something else
IMPLEMENTING COSINE SIMILARITY

#Count vectorizer translates each document into a vector of counts
f = sklearn.feature_extraction.text.CountVectorizer()
X = f.fit_transform(sen)

print(X.toarray())
print(f.get_feature_names())

<table>
<thead>
<tr>
<th>band</th>
<th>korn</th>
<th>love</th>
<th>tim</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 1 1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0 0 0 1 0 0 1 0 0 0 0 1 0 1 0 0 1 0 1 0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 1 0 0 1 0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0 0 1 0 1 0 0 1 0 0 1 1 1 0 0 0 0 0 1 0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

['absolut', 'ador', 'band', 'complet', 'corn', 'deep',
'detest', 'eat', 'goe', 'group', 'hate', 'korn',
'listen', 'love', 'name', 'outfit', 'passion', 'perform',
'rock', 'tim', 'use']
IMPLEMENTING COSINE SIMILARITY

#Count vectorizer translates each document into a vector of counts
f = sklearn.feature_extraction.text.CountVectorizer()
X = f.fit_transform(sen)

print(X.toarray())
print(f.get_feature_names())

#cosine_similarity computes the cosine similarity between
#a set of vectors
from sklearn.metrics.pairwise import cosine_similarity
cos_sim = cosine_similarity(X)
print(cos_sim)

Tim loves the band Korn
Tim adores the rock group Korn
Tim used to love Korn, but now he hates them
COSINE SIMILARITY PLOT

Includes stemming

- Tim loves the band Korn.
- Tim adores the rock group Korn.
- Tim loves eating corn.
- Tim used to love Korn, but now he hates them.
- Tim absolutely loves Korn.
- Tim completely detests the performers named Korn.
- Tim has a deep passion for the outfit the goes by the name of Korn.
- Tim loves listening to the band Korn while eating corn.
 WHICH WORDS MATTER: TF-IDF

Problem: neither Jaccard nor Cosine Similarity have a way to understand which words are important.

TF-IDF tries to estimate the importance of words based on
1) Their Term Frequency (TF) in a document
2) Their Inter-document Frequency (IDF), across all documents

Assumptions: If a term appears frequently in a document, it’s more important in that document.

If a term appears frequently in all documents, it’s less important.
**TF-IDF EQUATIONS**

\[
\text{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}
\]

Larger the more times document \(d\) uses term

\(t = t\)

\(d = \text{document}\)

\(f_{t,d} = \text{frequency of } t \text{ in } d\)

For each term \(t\) in \(d\), \(tf(t,d)\) is the fraction of words in \(d\) that are \(t\)

\[
\text{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}
\]

Approaches 0 as more documents use term

\(N = \text{number of documents}\)

\(D = \text{set of all documents}\)

\(|\{d \in D : t \in d\}| = \# \text{ documents which use term } t\)

For each term \(t\) in all \(D\), \(idf(t,D)\) is inversely proportional to the number of documents that use \(t\)
TF-IDF EQUATIONS

\[ tf(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \quad \text{idf}(t, D) = \log \frac{N}{|\{d \in D: t \in d\}|} \]

\[ tf - idf(t, d, F) = tf(t, d) \cdot idf(t, D) \]

\( t = t \)
\( d = \text{document} \)
\( f_{t,d} = \text{frequency of } t \text{ in } d \)

\( N = \text{number of documents} \)
\( D = \text{set of all documents} \)
\( |\{d \in D: t \in d\}| = \# \text{ documents which use term } t \)
**TF-IDF EXAMPLE**

**S1** = Tim loves Korn

**S2** = Tim loves eating corn

Terms = Tim, loves, Korn, eating Korn

\[
\begin{align*}
\text{tf-idf(Tim, s1)} &= \text{tf(Tim, s1)} \times \text{idf(Tim)} = \frac{1}{3} \times \log \left( \frac{2}{2} \right) = 0 \\
\text{tf-idf(loves, s1)} &= \text{tf(loves, s1)} \times \text{idf(loves)} = \frac{1}{3} \times \log \left( \frac{2}{2} \right) = 0 \\
\text{tf-idf(Korn, s1)} &= \text{tf(Korn, s1)} \times \text{idf(Korn)} = \frac{1}{3} \times \log \left( \frac{2}{1} \right) = \frac{1}{3} \times .69 = 0.23 \\
\text{tf-idf(eating, s2)} &= \text{tf(eating, s2)} \times \text{idf(eating)} = \frac{1}{4} \times \log(2/1) = 0.17 \\
\text{tf-idf(corn, s2)} &= \text{tf(corn, s2)} \times \text{idf(corn)} = \frac{1}{4} \times \log(2/1) = 0.17
\end{align*}
\]

Words in all documents aren’t helpful if we’re trying to rank documents according to their similarity or do keyword search.
TF-IDF IN PYTHON

These parameters make it match equations on previous slide

```
#TF-IDF using sklearn
f = sklearn.feature_extraction.text.TfidfVectorizer(smooth_idf=False, norm='l1')
X = f.fit_transform(sen)
print(X.toarray())
cos_sim = cosine_similarity(X)
print(cos_sim)
```

Tim loves the band Korn
Tim adores the rock group Korn
Tim loves eating corn
Tim used to love Korn, but now he hates them

```
[[1. 0.13 0.26 0.29 0.37 0.11 0.11 0.57]
 [0.13 1. 0.05 0.09 0.11 0.06 0.06 0.07]
 [0.26 0.05 1. 0.17 0.22 0.04 0.04 0.68]
 [0.29 0.09 0.17 1. 0.25 0.07 0.07 0.16]
 [0.37 0.11 0.22 0.25 1. 0.1 0.1 0.21]
 [0.11 0.06 0.04 0.07 0.1 1. 0.06 0.06]
 [0.11 0.06 0.04 0.07 0.1 0.06 1. 0.06]
 [0.57 0.07 0.68 0.16 0.21 0.06 0.06 1. ]]
```

TF-IDF not a great choice for these sentences, because it downweights frequent words (Tim, Korn and loves)
Modern deep learning has completely transformed text processing tasks like this NLP models, e.g., BERT and GPT-3/4 trained to understand documents.

Models are trained to predict missing words:
Tim loves the ____ Korn
Tim loves eating ____

Using billions of documents on the Web (training takes years of GPU time!!!)

We’re going to try BERT, which is a slightly older model than GPT-3/4.

Models take a window of text (e.g., 512 words) and produce an output vector (e.g., 768 floats) for each word.

Vector represents the “meaning” of that word in the context of the natural language in which it appears.

This vector can be used to predict the next word, or to measure the similarity of meaning of two words.
BERT

Randomly masked: A quick [MASK] fox jumps over the [MASK] dog

Predict: A quick brown fox jumps over the lazy dog

Trained via mask & predict

Transformer Architecture

Each word in input assigned a 768 element output vector, that depends on its context (before and after)

(GPT3 is 175B!)

110M params

https://towardsdatascience.com/bert-for-measuring-text-similarity-eec91c6bf9e1
USING BERT VECTORS

Each word is represented by a set of 768-element outputs
Convert to a single element 768-vector for each sentence by averaging words in document
Compute similarity between vectors (e.g., using Cosine Similarity)

Python `sentence-transformers` package makes this trivial

```python
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-mpnet-base-v2')
sen_embeddings = model.encode(sen)

import cosine_similarity

cos_sim = cosine_similarity(sen_embeddings)
print(cos_sim)
```

A popular BERT-like model known to perform well
Does averaging across documents
Contains a 768-element vector for each document
# USING BERT VECTORS

```python
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-mpnet-base-v2')
sen_embeddings = model.encode(sen)

cos_sim = cosine_similarity(sen_embeddings)

print(cos_sim)
```

<table>
<thead>
<tr>
<th>Tim loves the band Korn</th>
<th>[[1. 0.97 0.49 0.83 0.92 0.81 0.93 0.78]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tim adores the rock group Korn</td>
<td>[0.97 1. 0.46 0.82 0.91 0.81 0.93 0.77]</td>
</tr>
<tr>
<td>Tim loves eating corn</td>
<td>[0.49 0.46 1. 0.42 0.52 0.41 0.43 0.81]</td>
</tr>
<tr>
<td>Tim used to love Korn, but now he hates them</td>
<td>[0.83 0.82 0.42 1. 0.83 0.86 0.8 0.67]</td>
</tr>
</tbody>
</table>

*Captures meaning of sentences much better than other metrics*
<table>
<thead>
<tr>
<th>Statement</th>
<th>Heat Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tim loves the band Korn.</td>
<td>1.00</td>
</tr>
<tr>
<td>Tim adores the rock group Korn.</td>
<td>0.95</td>
</tr>
<tr>
<td>Tim loves eating corn.</td>
<td>0.90</td>
</tr>
<tr>
<td>Tim used to love Korn, but now he hates them.</td>
<td>0.95</td>
</tr>
<tr>
<td>Tim absolutely loves Korn.</td>
<td>0.90</td>
</tr>
<tr>
<td>Tim completely detests the performers named Korn.</td>
<td>0.95</td>
</tr>
<tr>
<td>Tim has a deep passion for the outfit the goes by the name of Korn.</td>
<td>0.90</td>
</tr>
<tr>
<td>Tim loves listening to the band Korn while eating corn.</td>
<td>0.95</td>
</tr>
</tbody>
</table>
Here are a bunch of sentences: "Tim loves the band Korn.", "Tim adores the rock group Korn.", "Tim loves eating corn.", "Tim used to love Korn, but now he hates them.", "Tim absolutely loves Korn.", "Tim completely detests the performers named Korn", "Tim has a deep passion for the outfit the goes by the name of Korn", "Tim loves listening to the band Korn while eating corn."

Which is most similar to the phrase "Sam loves the band Korn", and why?

ChatGPT

Based on the text provided in your message, the sentence most similar to "Sam loves the band Korn" is "Tim loves the band Korn."

This is because both sentences express a positive sentiment about the subject's feeling towards the band Korn, with the only difference being the subject of the sentence (Sam vs. Tim). The structure and the main verb "loves" are the same, making them closely similar in both meaning and sentiment.
SUMMARY

Saw three classes of tools - grep, sed, and awk, based on regular expressions to transform data

Saw how tools like Wrangler try to automate this

Looked at text processing techniques
   Jaccard and Cosine similarity
   Tokenization, stemming, stop lists
   TF-IDF
   Embeddings using BERT

We will return to embeddings and GPT models in a few weeks