Programming with Data Bootcamp: Lecture 2

http://dsg.csail.mit.edu/6.S079/

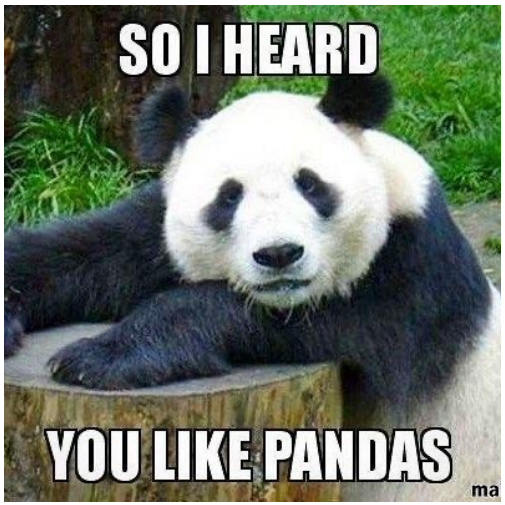
Slides courtesy of Sam Madden /

Tim Kraska (6.S079)

Key ideas:
Pandas & Parquet
Regexes
sed/awk/grep
Working with Text

Onto Pandas

- Pandas is a python library for working with tabular data
- Set-oriented thinking in Python
- Provides relation-algebra like ability to filter, join, and transform data



Loading a Data Set

```
import pandas as pd

df = pd.read_csv("bands.csv")
print(df)
```

Pandas tables are called “data frames”

All dataframes have an “index” – by default, a monotonically increasing number

As in SQL, columns are named and typed
Unlike SQL, they are also ordered (i.e., can access records by their position, and the notion of “next record” is well defined)

	bandid	bandname	genre
0	1	limp bizkit	rock
1	2	korn	rock
2	3	creed	rock
3	4	nickelback	rock

Accessing Columns

```
print(df.bandname)
```

```
0    limp bizkit
1           korn
2          creed
3       nickelback
```

Dots and brackets are equivalent
Can't use dots if field names are reserved keywords (e.g., "type", "class")

```
print(df["genre"])
```

```
Name: bandname, dtype: object
0    rock
1    rock
2    rock
3    rock
Name: genre, dtype: object
```

Accessing Rows

```
#limp bizkit rows
df_lb = df[df.bandname == 'limp bizkit']

print(df_lb)
```

```
   bandid     bandname genre
0       1  limp bizkit  rock
```

Array of Booleans with len(df) values in it

Indexing into a dataframe with a list of bools returns records where value in list is true

```
#get the record at position 1
print(df.iloc[1])
```

```
bandid          2
bandname     korn
genre        rock
Name: 1, dtype: object
```

	bandid	bandname	genre
0	1	limp bizkit	rock
1	2	korn	rock
2	3	creed	rock
3	4	nickelback	rock

iloc vs loc

```
#get the genre of record with index attribute = 1
print(df.loc[1,"genre"])
```

```
rock
```

Index column

	bandid	bandname	genre
0	1	limp bizkit	rock
1	2	korn	rock
2	3	creed	rock
3	4	nickelback	rock

df.loc[1,'bandid']
df.iloc[1,0]

- loc uses the dataframe index column to access rows and column names to access data
- iloc uses the position in the dataframe and index into list of columns to access data
- By default index column and position are the same

Changing the Index

```
df_new = df.set_index("bandname")
print(df_new)
```

```
             bandid genre
bandname
limp bizkit       1  rock
korn              2  rock
creed             3  rock
nickelback        4  rock
```

```
print(df_new.loc["creed"])
```

```
bandid       3
genre     rock
Name: creed, dtype: object
```

Clicker

bandname	bandid	genre
limp bizkit	1	rock
korn	2	rock
creed	3	rock
nickelback	4	rock

- Given dataframe with bandname as index

- What is does this statement output?

```
print(df.iloc[1,1],df.loc['korn','bandid'])
```

A. rock 2
B. 2 2
C. 2 rock
D. 1 2

https://clicker.mit.edu/6.S079/

Transforming Data

```
df["is_rock"] = df.genre == "rock"

print(df)
```

```
   bandid     bandname genre  is_rock
0       1  limp bizkit  rock     True
1       2         korn  rock     True
2       3       creed  rock     True
3       4   nickelback  rock     True
```

```
df.loc[df.bandname == 'limp bizkit','genre'] = 'terrible'

print(df)
```

```
   bandid     bandname     genre
0       1  limp bizkit  terrible
1       2         korn      rock
2       3       creed      rock
3       4   nickelback      rock
```

Must Use iloc/loc to Change Data

This works:

```
df.loc[df.bandname == 'limp bizkit','genre'] = 'terrible'
```

This does not (even though it is a legal way to read data):

```
df[df.bandname == 'limp bizkit']['genre'] = 'terrible'
```

```
/Users/madden/6.s079/lec4-code/code.py:14: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

Grouping

Apply "count" to all non-grouping columns

```
df_grouped = df.groupby("genre").count()
print(df_grouped)
```

Creates a "GroupByObject" which supports a variety of aggregation functions

genre	bandid	bandname
rock	3	3
terrible	1	1

Resulting data frame is indexed by the grouping column

Multiple Aggregates

```
df_grouped = df.groupby("genre").agg(max_band=("bandid","max"),
    num_bands=("bandname","count"))
print(df_grouped)
```

Name of column in output data frame
Note funky syntax

genre	max_band	num_bands
rock	4	3
terrible	1	1

Joining (Merge)

```
df_bandfans = pd.read_csv("bandfans.csv")

df_merged = df.merge(df_bandfans, left_on="bandid", right_on="bf_bandid")
print(df_merged)
```

Join attributes
"left" data frame is the one we are calling merge on
"right" data frame is the one we pass in

	bandid	bandname	genre	bf_bandid	bf_fanid
0	1	limp bizkit	terrible	1	1
1	1	limp bizkit	terrible	1	2
2	2	korn	rock	2	1
3	3	creed	rock	3	1

Bands that don't join are missing

Left/Right/Outer Join

```
df_merged = df.merge(df_bandfans, left_on="bandid", right_on="bf_bandid", how="left")
print(df_merged)
```

	bandid	bandname	genre	bf_bandid	bf_fanid
0	1	limp bizkit	terrible	1.0	1.0
1	1	limp bizkit	terrible	1.0	2.0
2	2	korn	rock	2.0	1.0
3	3	creed	rock	3.0	1.0
4	4	nickelback	rock	NaN	NaN

Chained Expressions

- All Pandas operations make a copy of their input and return it (unless you specify inplace=True)
- This makes long chained expressions common
  - Inefficient, but syntactically compact

```
df_merged = df.merge(df_bandfans, left_on="bandid", right_on="bf_bandid")\
              .groupby("bandname")\
              .agg(num_fans=("bf_fanid","count"))
print(df_merged)
```

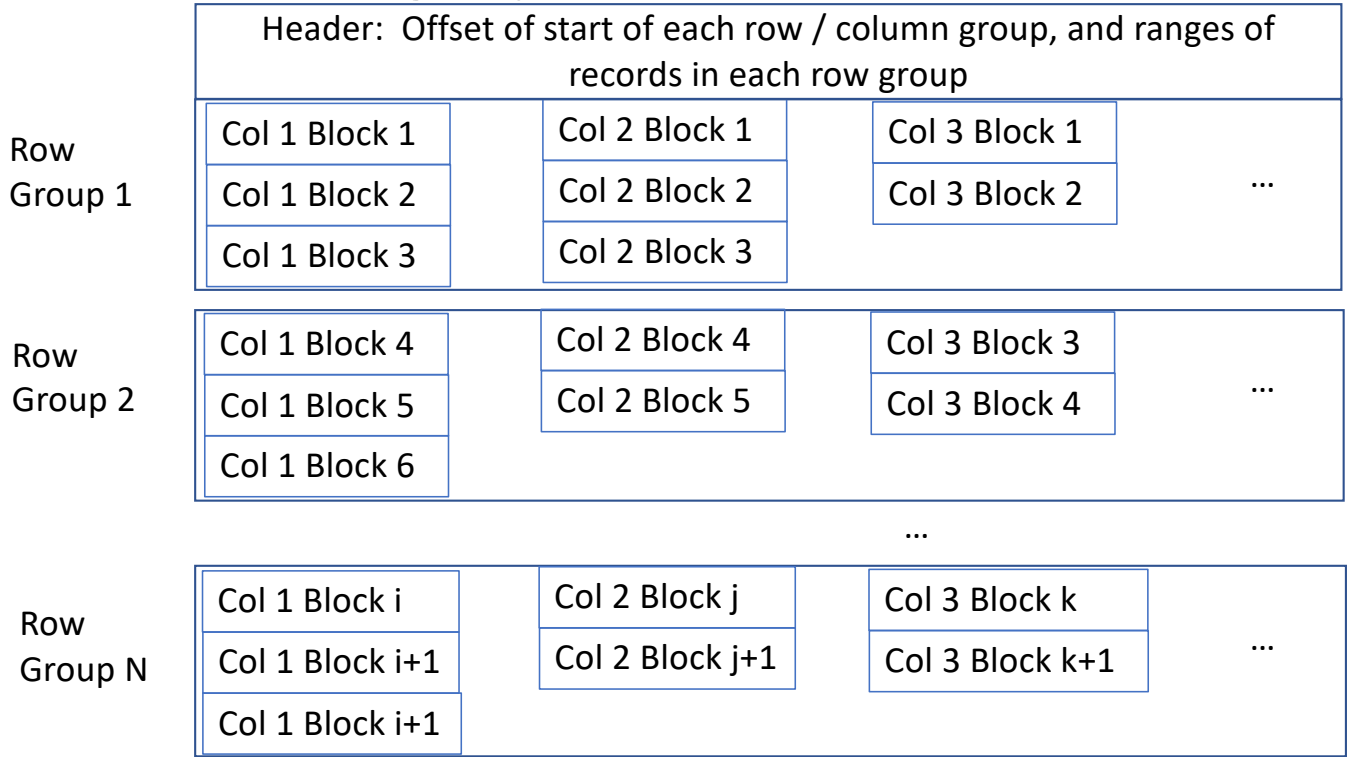
```
             num_fans
bandname
creed               1
korn                1
limp bizkit         2
```

Efficient Data Loading: Parquet

- Parquet is a file format that is MUCH more efficient than CSV for storing tabular data
- Data is stored in binary representation
  - Uses less space
  - Doesn't require conversion from strings to internal types
  - Doesn't require parsing or error detection
  - Column-oriented, making access to subsets of columns much faster

Parquet Format

- Data is partitioned sets of rows, called “row groups”
- Within each row group, data from different columns is stored separately



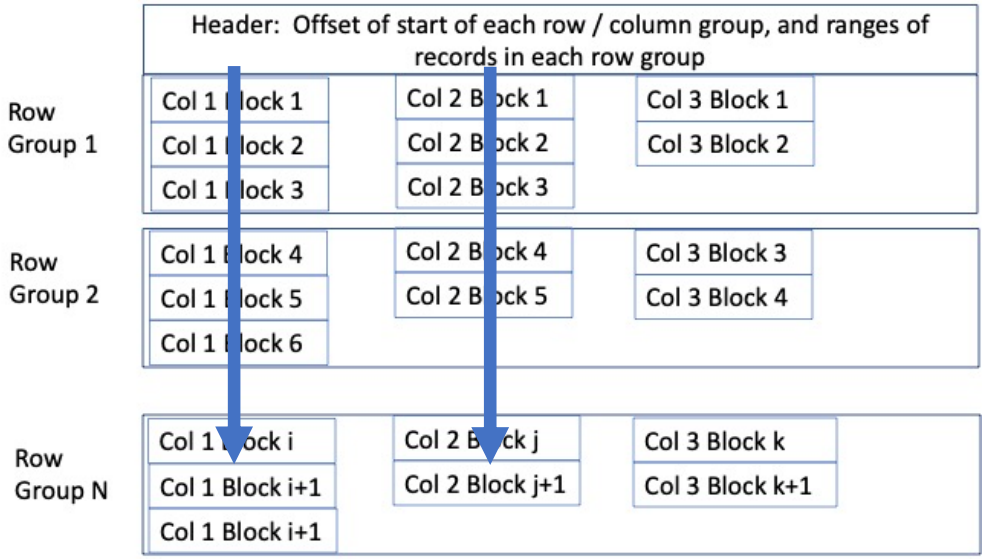
Using header, can efficiently read any subset of columns or rows without scanning whole file (unlike CSV)

Within a row group, data for each column is stored together

Predicate Pushdown w/ Parquet & Pandas

```
pd.read_parquet('file.pq', columns=['Col 1', 'Col 2'])
```

- Only reads col1 and col2 from disk
- For a wide dataset (e.g., our vehicle dataset w/ 93 columns), saves a ton of I/O



Performance Measurement

- Compare reading CSV to parquet to just columns we need

```
t = time.perf_counter()
df = pd.read_csv("FARS2019NationalCSV/Person.CSV", encoding = "ISO-8859-1")
print(f"csv elapsed = {time.perf_counter() - t:.3} seconds")

t = time.perf_counter()
df = pd.read_parquet("2019.pq")
print(f"parquet elapsed = {time.perf_counter() - t:.3} seconds")

t = time.perf_counter()
df = pd.read_parquet("2019.pq", columns = ['STATE','ST_CASE','DRINKING','PER_TYP'])
print(f"parquet subset elapsed = {time.perf_counter() - t:.3} seconds")
```

```
csv elapsed = 1.18 seconds
parquet elapsed = 0.338 seconds
parquet subset elapsed = 0.025 seconds
```

47x speedup

When to Use Parquet?

- Will always be more efficient than CSV
- Converting from Parquet to CSV takes time, so only makes sense to do so if working repeatedly with a file
- Parquet requires a library to access/read it, whereas many tools can work with CSV
- Because CSV is text, it can have mixed types in columns, or other inconsistencies
  - May be useful sometimes, but also very annoying!
  - Parquet does not support mixed types in a column

Pandas vs SQL

- Could we have done this analysis in SQL?
- Probably…
- But not the plotting, or data cleaning, or data downloads
  - So would need Python to clean up data, reload into SQL, run queries
  - Declaring schemas, importing data, etc all somewhat painful in SQL
- So usual workflow is to use SQL to get to the data in the database, and then python for merging, cleaning and plotting
- Generally, databases will be faster for things SQL does well, and they can handle data that is much larger than RAM, unlike Python

THREE EXTREMELY 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

WHAT IS A REGULAR EXPRESSION?

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

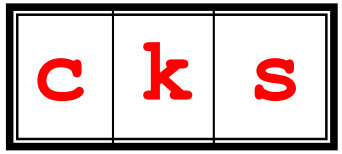
Regular expressions are endemic to Unix

- 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.

regular expression → 

Unix rocks.

match

UNIX sucks.

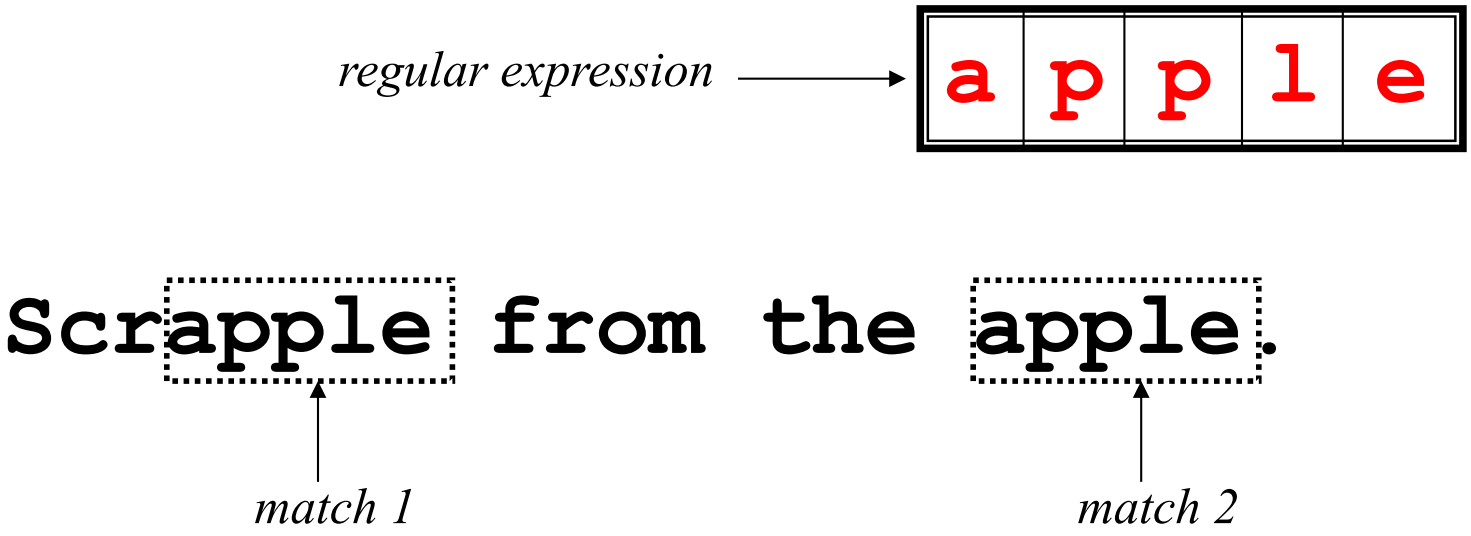
match

UNIX is okay.

no match

REGULAR EXPRESSIONS

A regular expression can match a string in more than one place.



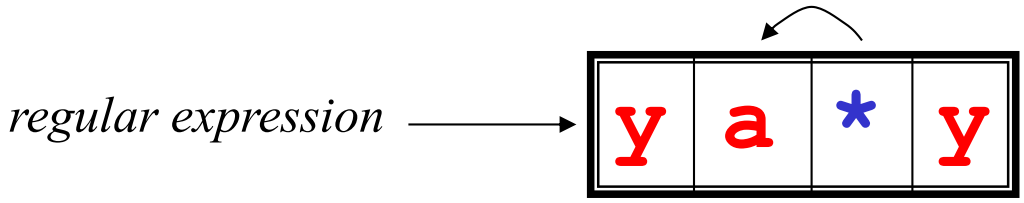
REGULAR EXPRESSIONS

The . regular expression can be used to match any character.

REPETITION

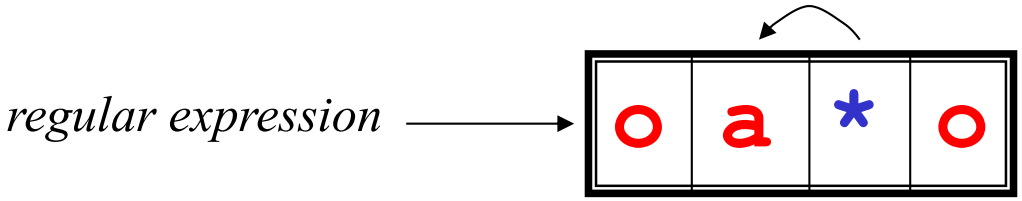
The * is used to define **zero or more** occurrences of the *single* regular expression preceding it.

+ Matches one or more occurrences




```
I got mail, yaaaaaaaaaay!
```

match




```
I sat on the stoop
```

match

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

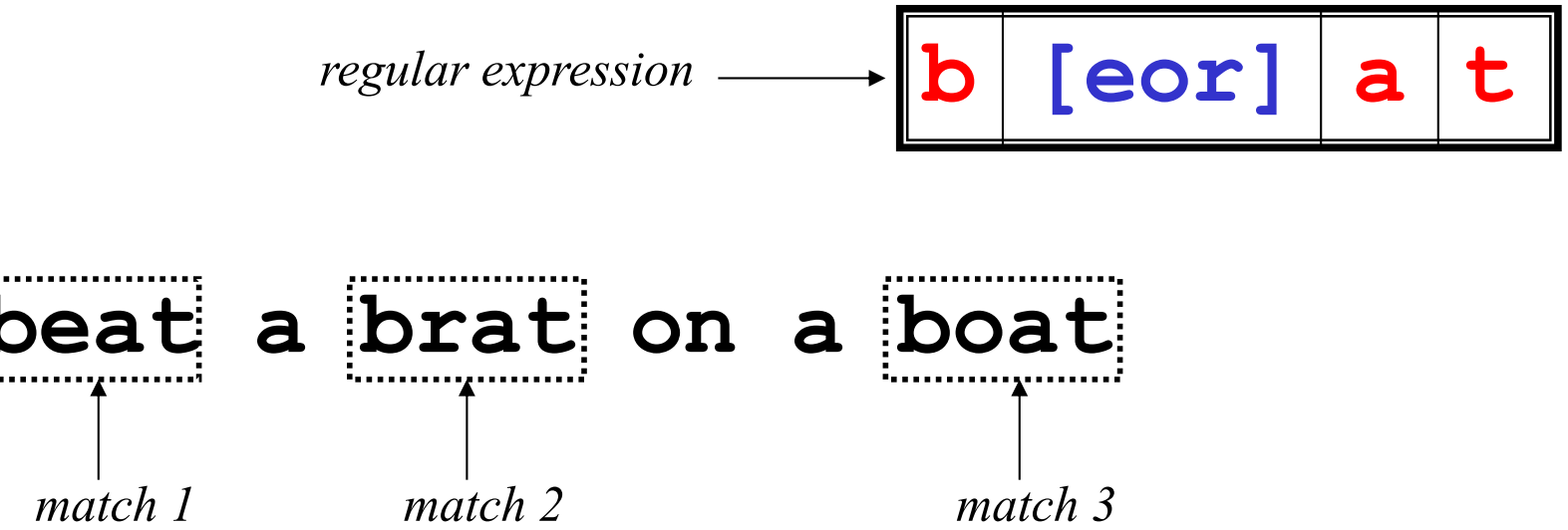
$a|b^*$ denotes {ε, "a", "b", "bb", "bbb", ...}

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

$ab^*(c)$ 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", ...}

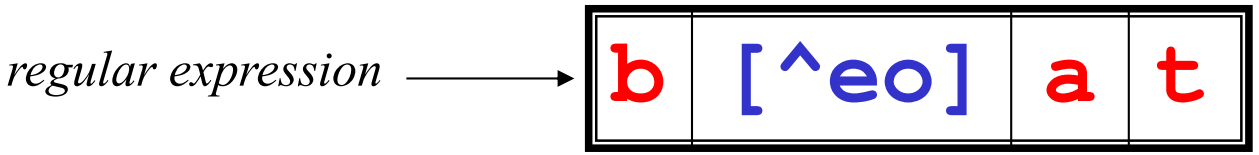
CHARACTER CLASSES – OR SHORTHAND

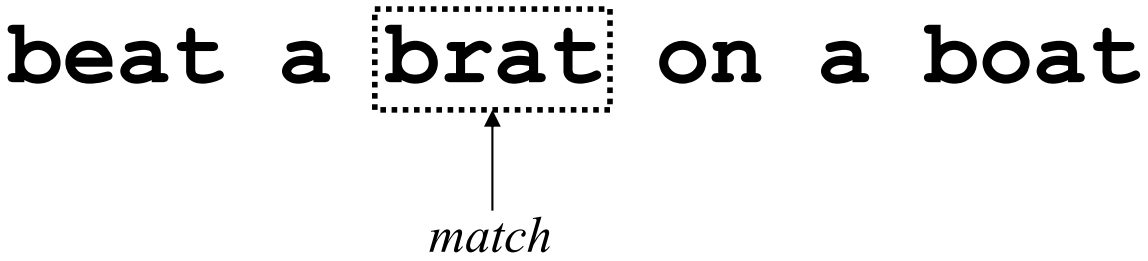
Character classes `[]` can be used to match any specific set of characters.



NEGATED CHARACTER CLASSES

Character classes can be negated with the `[^]` syntax.





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 ***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 (*alpha*, *lower, upper, alnum*, *digit*, *punct*, *cntrl*)

Syntax [:*name*:]

- [a-zA-Z] [[:alpha:]]
- [a-zA-Z0-9] [[:alnum:]]
- [45a-z] [45[:lower:]]

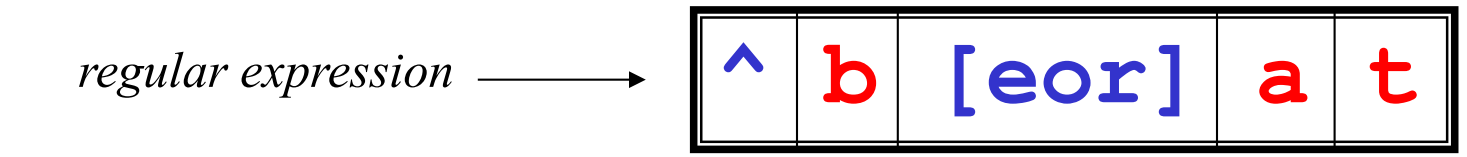
Important for portability across languages

ANCHORS

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

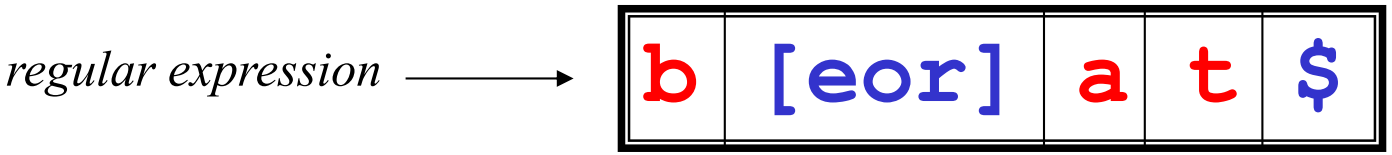
^ means beginning of the line

$ means end of the line



beat a brat on a boat

match



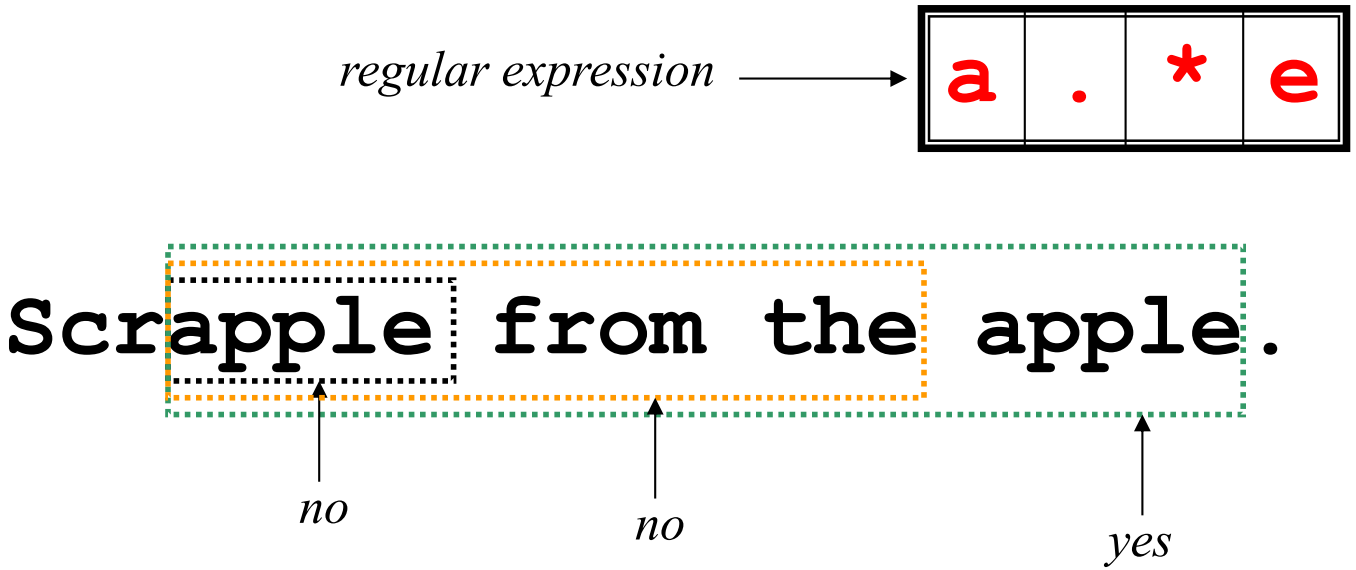
beat a brat on a boat

match

`^word$` `^$`

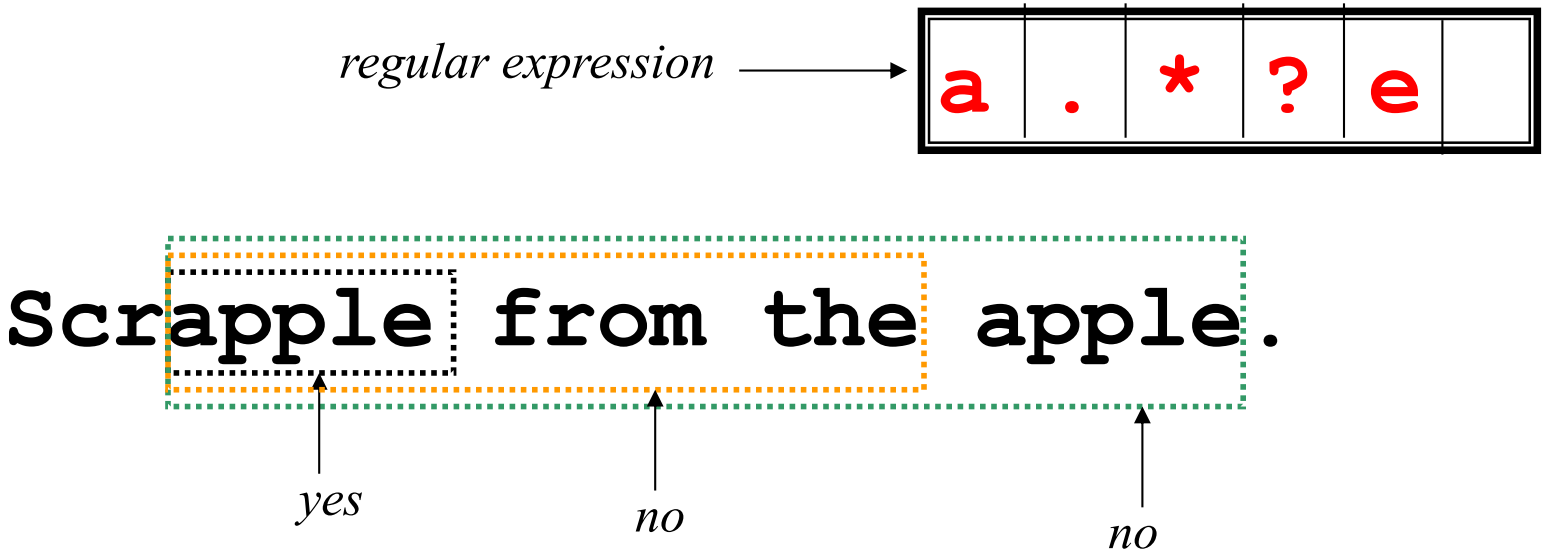
MATCH LENGTH

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



MATCH LENGTH

Append a ? to match the shortest string possible:



PRACTICAL REGEX EXAMPLES

Dollar amount with optional cents

- `\$[0-9]+(\.[0-9][0-9])?`

Time of day

- `(1[012]|[1-9]):[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

FAMILY DIFFERENCES

grep - uses regular expressions for pattern matching

fgrep - file grep, does not use regular expressions, only matches fixed strings but can get search strings from a file

egrep - extended grep, uses a more powerful set of regular expressions but does not support backreferencing, generally the fastest member of the grep family

agrep – approximate grep; not standard

GREP DEMO

```
grep '\"text\": ".*location.*"' twitter.json
```

"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",

BACKREFERENCES

Sometimes it is handy to be able to refer to a match that was made earlier in a regex

This is done using *backreferences*

- *n* is the backreference specifier, where *n* is a number

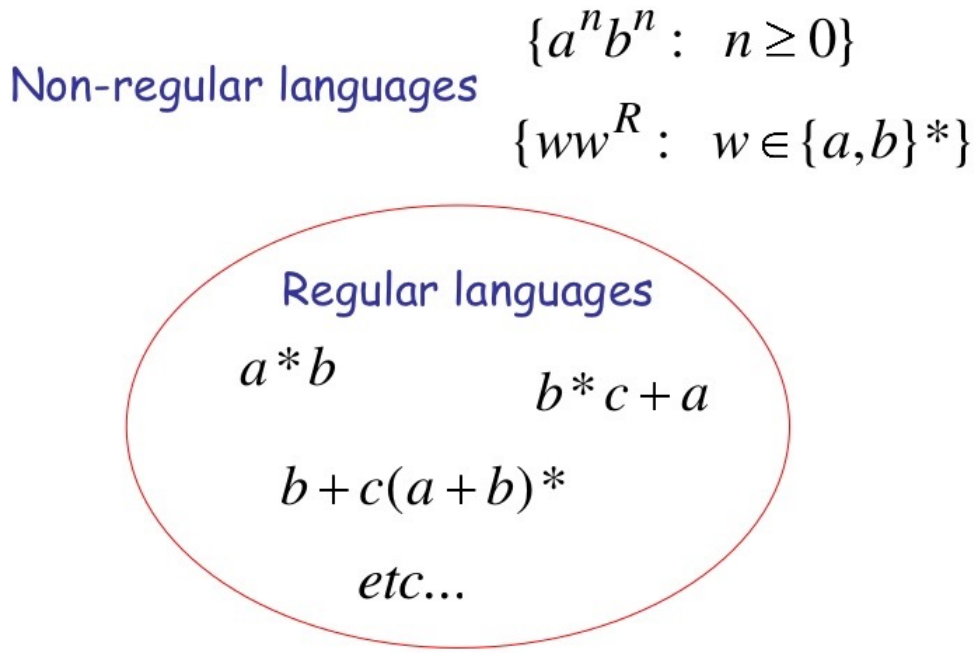
Looks for *n*th subexpression

For example, to find if the first word of a line is the same as the last:

- `^([[:alpha:]]+) .* \1$`
- Here, `([[:alpha:]]+)` matches 1 or more letters

FORMALLY

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



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

([ab]*)\1

https://clicker.csail.mit.edu/6.s079/

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 EXTREMELY 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 intput, 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/\(His job is to\).*/\1 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

```
[
  {
    "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/XweGngmxlP",
    "truncated": false,
…
```

```
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
```

```
sed -E 's/^([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)[^\"]*\"([^\"]*)\".*/\1,\2/g' apache.txt
```

IP Address — Stuff up to quote — URL

```
192.168.2.20,GET /cgi-bin/try/ HTTP/1.0
127.0.0.1,GET / HTTP/1.0
```

THREE EXTREMELY POWERFUL TOOLS

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 = gensub("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:

https://www.gnu.org/software/gawk/manual/html_node/Built_002din.html

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?

```
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

$$sim(s1, s2) = \frac{s1 \cap s2}{s2 \cup s2}$$

sim({tim,loves,korn}, {tim, loves, eating, corn}) = 2 / 5

sim({tim,absolutely,adores,the,band,korn}, {tim, loves, korn}) = 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")

EXAMPLE


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.
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.60
0.50
0.40
0.30
0.20
0.10

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

```
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"]

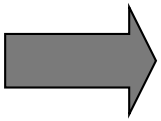
STEMMING

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 equival 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*

STEMMING IN PYTHON

```
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

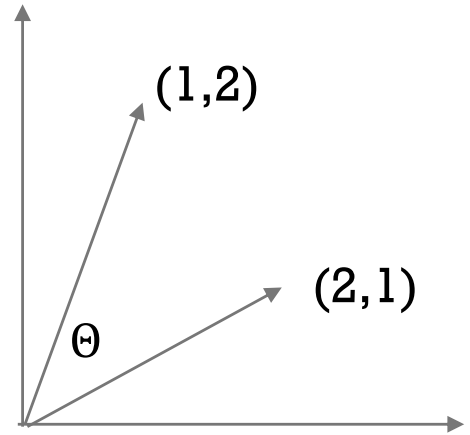
```
sen = [
   "Tim loves the band Korn.",
   "Tim adores the rock group Korn.",
   "Tim loves eating corn.",
   "Tim used to love Korn, but now he hates them.",
   "Tim absolutely loves Korn.",
   "Tim completely detests the performers named Korn",
   "Tim has a deep passion for the outfit the goes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

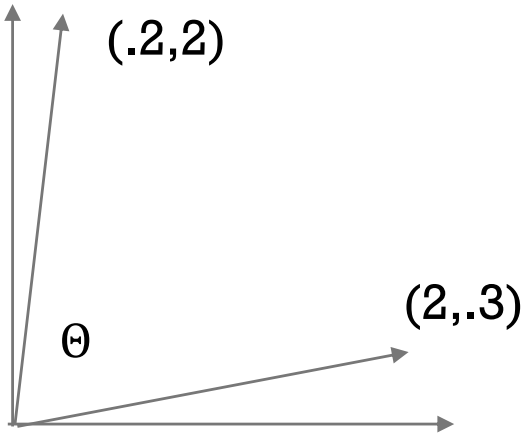
Given two vectors, a standard way to measure how similar they are

Cos(v1, v2) = closeness of two vectors (smaller is closer)



$$\text{Cos}(\Theta) = V1 \bullet V2 \,/\, ||V1|| \times ||V2||$$

$$\text{Cos}(\Theta) = [1\ 2] \bullet [2\ 1] \,/\, (\text{sqrt}(5))\ \hat{}\ 2$$
$$\text{Acos}(4 / 5) = 36.8°$$



$$||V1|| = 2.01,\ ||V2|| = 2.02$$
$$\text{Cos}(\Theta) = [.2\ 2] \bullet [2\ .3] \,/\, 2.015$$
$$= 1/2.015$$
$$\text{Acos}(1/2.015) = 60.2°$$

COSINE SIMILARITY OF WORD VECTORS

$\cos(\Theta) = V1 \bullet V2 / \|V1\| \times \|V2\|$

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 • V2 = 1
||V1|| = sqrt(3)
||V2|| = sqrt(4)

1 / sqrt(3) * sqrt(4) = .29

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

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

V1 • V2 = 2
||V1|| = sqrt(3)
||V2|| = sqrt(7)

2 / sqrt(3) * sqrt(7) = .43

Typically, when using cosine similarity, we don't take the acos of the values (since acos is expensive)

JACCARD VS COSINE

S1 = Tim loves Korn
S2 = Tim loves eating corn

CosSim(S1,S2) = .29
Jaccard(S1,S2) = .4

S3 = Tim absolutely adores the band Korn
CosSim(S1,S3) = .43
Jaccard(S1,S3) = .28

Jaccard more sensitive to different document lengths than CosSim

CosSim can incorporate repeated words (by using non-binary vectors)

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())
```

band korn love tim

```
[[0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0]
 [0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0]
 [0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 1 1]
 [1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0]
 [0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 1 0 1 0]
 [0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 1 0 0 1 0]
 [0 0 1 0 1 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 0]]
```

```
['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 loves eating corn
Tim used to love Korn, but now he hates them

```
[[1.   0.45 0.5  0.67 0.75 0.41 0.41 0.76]
 [0.45 1.   0.22 0.4  0.45 0.37 0.37 0.34]
 [0.5  0.22 1.   0.45 0.5  0.2  0.2  0.76]
 [0.67 0.4  0.45 1.   0.67 0.37 0.37 0.51]
 [0.75 0.45 0.5  0.67 1.   0.41 0.41 0.57]
 [0.41 0.37 0.2  0.37 0.41 1.   0.33 0.31]
 [0.41 0.37 0.2  0.37 0.41 0.33 1.   0.31]
 [0.76 0.34 0.76 0.51 0.57 0.31 0.31 1.  ]]
```

COSINE SIMILARITY PLOT

Includes stemming

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, its less important

TF-IDF EQUATIONS

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

t = t

d = document

$f_{t,d}$ = frequency of t in d

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

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

N = number of documents

D = set of all documents

$|\{d \in D : t \in d\}|$= # 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

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \qquad idf(t,D) = log \frac{N}{|\{d \in D : t \in d|}$$

$$tf\text{-}idf(t,d,F) = tf(t,d) \bullet idf(t,D)$$

t = t

d = document

$f_{t,d}$ = frequency of t in d

N = number of documents

D = set of all documents

$|\{d \in D : t \in d\}|$= # documents which use term t

TF-IDF EXAMPLE

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

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

S1 = Tim loves Korn

S2 = Tim loves eating corn

S1 = [0, 0, .23]
S2 = [0, 0, .17, .17]

Terms = Tim, loves, Korn, eating Korn

tf-idf(Tim,s1) = tf(Tim,s1) x idf(Tim) = 1/3 x log (2/2) = 0

tf-idf(loves,s1) = tf(loves,s1) x idf(loves) = 1/3 x log (2/2) = 0

tf-idf(Korn,s1) = tf(Korn,s1) x idf(Korn) = 1/3 x log (2/1) = 1/3 x .69 = 0.23

tf-idf(eating,s2) = tf(eating,s2) x idf(eating) = 1/4 x log(2/1) = 0.17

tf-idf(corn,s2) = tf(corn,s2) x idf(corn) = 1/4 x log(2/1) = 0.17

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 (Korn and loves)

MODERN ML TECHNIQUES

Modern deep learning has completely transformed text processing tasks like this

NLP models, e.g., BERT and GPT-3 trained to *understand* documents

Models are trained to predict missing words:

Tim loves the ____ Korn

Tim loves eating ____

We're going to try BERT, which is a slightly older model than GPT-3

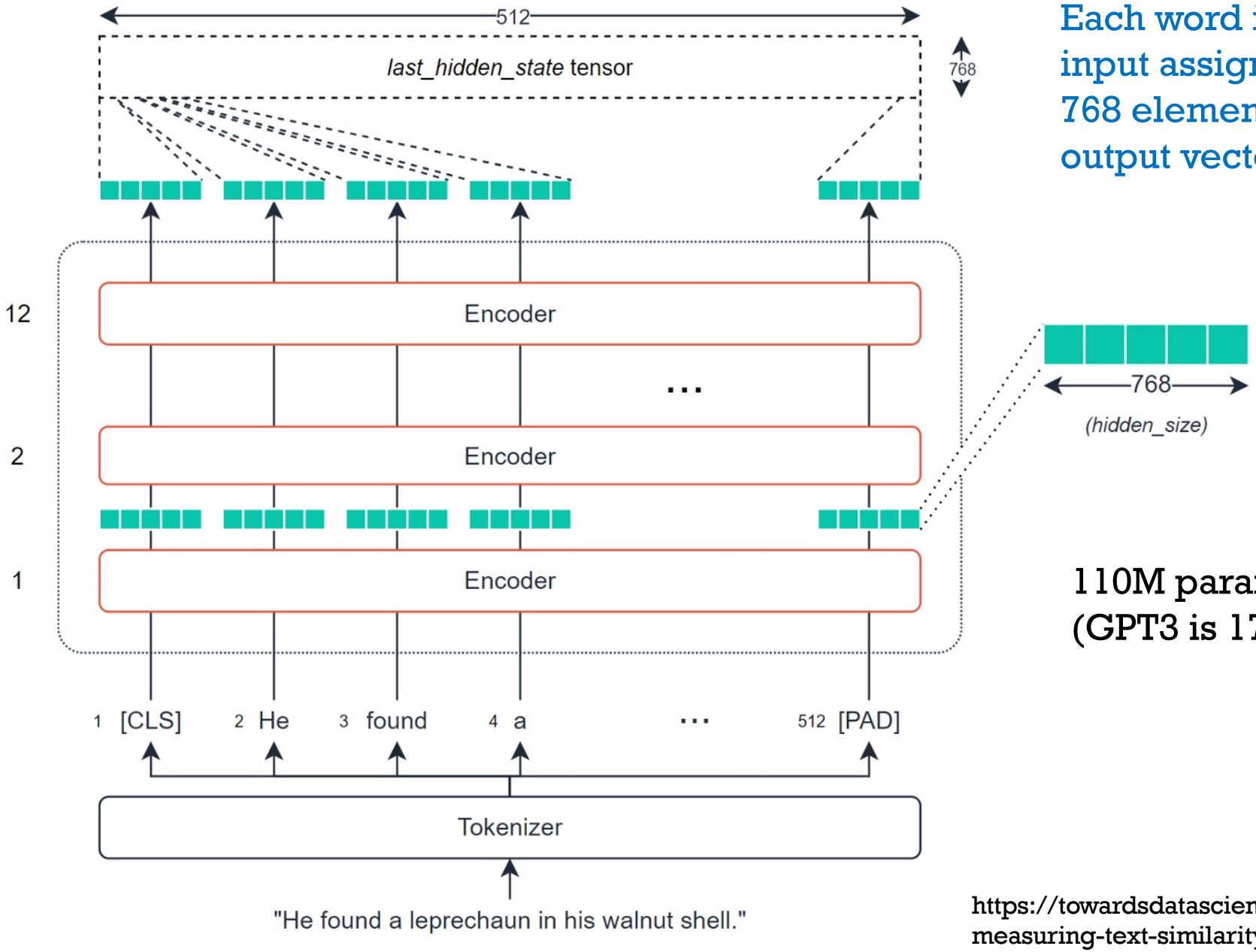
Using billions of documents on the Web (training takes years of GPU time!!!)

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 ARCHITECTURE



Each word in input assigned a 768 element output vector

110M params (GPT3 is 175B!)

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

```
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)
```

A popular BERT-like model known to perform well

Does averaging across documents

Contains a 768-element vector for each document

USING BERT VECTORS

```
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)
```

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.   0.97 0.49 0.83 0.92 0.81 0.93 0.78]
 [0.97 1.   0.46 0.82 0.91 0.81 0.93 0.77]
 [0.49 0.46 1.   0.42 0.52 0.41 0.43 0.81]
 [0.83 0.82 0.42 1.   0.83 0.86 0.8  0.67]
 [0.92 0.91 0.52 0.83 1.   0.79 0.87 0.76]
 [0.81 0.81 0.41 0.86 0.79 1.   0.8  0.66]
 [0.93 0.93 0.43 0.8  0.87 0.8  1.   0.71]
 [0.78 0.77 0.81 0.67 0.76 0.66 0.71 1.  ]]
```

Captures meaning of sentences much better than other metrics

HEAT MAP


Tim loves the band K
Tim adores the rock g
Tim loves eating corn
Tim used to love Korr
Tim absolutely loves
Tim completely detes
Tim has a deep passi
Tim loves listening to
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.
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.60
0.50

SUMMARY

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

Saw how tools like Instabase and 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 in a few weeks