CS 217 Data Management and Information Processing

Text and Natural Language Data

Up to now

- We are mostly focused on processing numeric data

Natural Language Data

- **Natural Language** means a language that has evolved naturally.
  - For example, *written English* or *spoken English*.
  - These are **human** languages.
- In contrast, a **Formal Language** is defined by a strict set of rules.
  - For example, *SQL* or *Python*.
  - These are computer or mathematical languages.

- Natural languages are difficult for machines to process because there are no clear rules for semantics (meaning).
  - Thus, natural language processing (NLP) is a difficult artificial intelligence problem. Eg., Siri, Alexa, Google search.
  - Details of state-of-the-art NLP is way beyond the scope of this course…

How to deal with natural language data?

For example, Yelp reviews:

- Ate the momos during the momo crawl.. Was the best of the lot so decided to eat at the restaurant and the mutton thali was equally good!!
- Pizza here made my night... Good people and great pizza. They can do anything you ask with a great attitude!
- Great brisket sandwich as claimed. Weird that it's a gas station/ hipster bbq lunch spot/ hallmark store carwash.
- Interesting food, great atmosphere, and great service. I like this place because there really isn't anything like this around the Charlotte area. I will definitely be coming back! Oh, and MILK BREAD.
- Our waiter did a great job of ignoring our table.

Processing Natural Language Data

- Simple tasks:
  - Pattern matching
  - Document similarity
- Complex task:
  - Understand the semantics structure
  - Understand the mood

Regular Expression

Pattern matching

- In SQL, you can use LIKE or REGEXP to find patterns
- Eg., it might be useful to find all the reviews mentioning Pizza:

```
SELECT text FROM review WHERE text LIKE "%pizza%";
```

  - Red, white and bleu salad was super yum and a great addition to the menu! This location was clean with great service and food served at just the right temps! Kids **pizza** is always a hit too with lots of great side dish options for the kiddos! When I'm on this side of town, this will definitely be a spot I'll hit up again!
  - **Pizza** here made my night... Good people and great **pizza**. They can do anything you ask with a great attitude!
  - Another case of the Emperor's New Clothes. Someone of the artsy set decided that this relatively good but overpriced fare was great **pizza** and all the lemmings followed suit. Will anyone tell the Emperor he has no clothes? The limited hours, no delivery, and lack of dining area add to the snob appeal. Don't be taken in.

Regular Expressions (REGEXP)

Regular Expressions are patterns that match text.

```
… WHERE column REGEXP "pattern" …
```

- They are much more flexible than the `LIKE` expressions we have used.
  - `LIKE` expressions use `%` to represent a sequence of unknown characters and `_` to represent a single unknown character.
- Regular Expressions can be much more specific:
  - Match different types of characters (letters, numbers, whitespace)
  - Allows sub-patterns to repeat
  - … and more
- SQLite, MySQL, and every major DBMS support `REGEXP`, although the syntax details may vary.
- Regular Expressions are also used in many other programing languages and for searching with files using:
  - `grep` command-line tool on Mac and Unix.
  - `FINDSTR` and `Select-String` commands on Windows.

A simple Regular Expression

- Returns **true** only if the pattern is present in the text.
- Is case sensitive.

`barf` matches:

- barf
- barfly
- I embarfed on my journey.
- I barfed at McDonalds.

`barf` does *not* match:

- Barf
- BARF
- This bar finally closed.
- I enjoyed my meal at McDonalds.
- arf

To experiment with Regular Expressions

- In SQLite:

```
SELECT "Some String" REGEXP "tri";
```

  - Will return 1 (true) if it matches.

- In Python:

```
import re
re.search("tri" , "Some String")
```

Beginning and end of the text

Normally, regular expressions match anywhere in the text, but we can change that behavior as follows:

- `^` represents the beginning of the text
- `$` represents the end of the text

`^Hello` matches “Hello World.” but does not match “Big Hello”

`world$` matches “hello world” but does not match “world cup”

`^hello world$` matches “hello world” and nothing else.

Sets of characters

`.` (period) matches any one character (as does `_` with `LIKE` expressions)

`m.p` matches “map” “mop” “mope”
but *not* “dimple” nor “mousetrap”

Square braces `[…]` specify a set of characters, any of which can match.

`[aA]` specifies either “a” or “A”

`[a-z]` specifies any of the characters between “a” and “z”

`[^b]` specifies any character *other than* “b”

These sets can be combined, as follows:

`[a-zA-Z013]` specifies any English letter or numbers 0, 1, or 3

`[^CDA]` specifies any character other than “C” “D” or “A

Examples

`^[Dd]atabase`

Matches: “Database”, “databases”, “Database Mgmt”

Does not match: “My Database”, “data”

`C[aeiou][b-d]`

Matches: “Cab”, “Abra **Cad**abra”, “Cob”, “Cic”

Does not match: “Clad”, “CB”

`[Cc].[^aeiou]`

Matches: “Cab”, “ccc”, “Green **cat**nip”, “Ma**cB**ride”

Does not match: “Clad”, “CB”

Repetition

`*` lets the previous thing repeat any number of times, including zero times.

`.*` matches anything because it's any one character repeated any number of times.

`+` lets the previous thing repeat one or more times.

`?` makes the previous thing optional (appears zero or one times).

`{n,m}` lets the previous thing repeat between *n* and *m* times.

The pipe character gives OR:

`(this|that)`

Examples

`^1.*t$`

Matches: “12 point”, “100.3 feet”, “111ttt”, “1t”

Does not match: “This 12 point font”

`[Cc]ats?`

Matches: “Cat behavior”, “5 cats”, “catnip”

Does not match: “cast”, “CATS”

`[0-9]+.[a-z]?`

Matches: “249032/b”, “23.”

Does not match: “a”, “aa”, “1”

`([cC]at|[Dd]og)( food)?`

Matches: “cat food”, “Dog”

Does not match: “ food”

Car license plate example

Let's say we want to match text that could be car license plates.

- Must be 6 to 8 characters, optionally with a space or dash in the middle.
- Eg., "123-AB3" or "4FDK930" or "MFI 678"

```
[A-Z0-9]{3,4}[ \-]?[A-Z0-9]{3,4}
```

3 or 4 capital letters or numbers

Optional space or hyphen

3 or 4 capital letters or numbers

"\" is needed to "escape" the normal meaning of hyphen inside square brackets. We want the literal hyphen character; we are not specifying a range of characters.

Special characters

Whitespace in text is represented by a variety of characters:

- `\t` is the tab character
- `\n` is newline
- `\r` is carriage return
  - Unix-style text ends each line with "`\n`"
  - Windows-style text ends each line with "`\r\n`"

- To match any whitespace, use `[ \t\n\r]` or simply `\s`
- `\w` represents a "word character", meaning anything other than whitespace: `[^\s]`

Regex Summary

- Regular expressions are used to match text, both in SQL and in many other data management tools.
- A match anywhere in the text returns *true*.
- `^` anchors to the beginning
- `$` anchors to the end
- `.` matches any character
- `[…]` specifies a set of possible characters
- `[a-z]` hyphen specifies a range
- `[^abc]` carrot within brackets negates the match
- Repetitions are supported:
  - `*` any number
  - `+` one or more
  - `?` zero or one
  - `{n,m}` n to m repetitions
- `|` pipe character gives OR
- `(…)` can be used for grouping

Comparing Document Similarity

Word frequency vectors

- If we treat the text as just a "**bag of words**" we lose some meaning (the *ordering* of words is lost), but easy comparisons become possible.
- A **word frequency vector** has a dimension for each possible word and the value in that dimension represents the frequency of that word.
  - Might have 30,000 dimensions:
  - [a, Aachen, aah, aardvark, …, zygote, zymurgy, zzz]
- Any *text* can be represented by such a vector:
  - "I like cats like that."
    [0, …, 0, **0.2**, 0, … 0, **0.2**, 0, …, 0, **0.4**, 0, …, 0, **0.2**, 0, … 0]

- Represent each text as a point in a space of extremely high dimension.

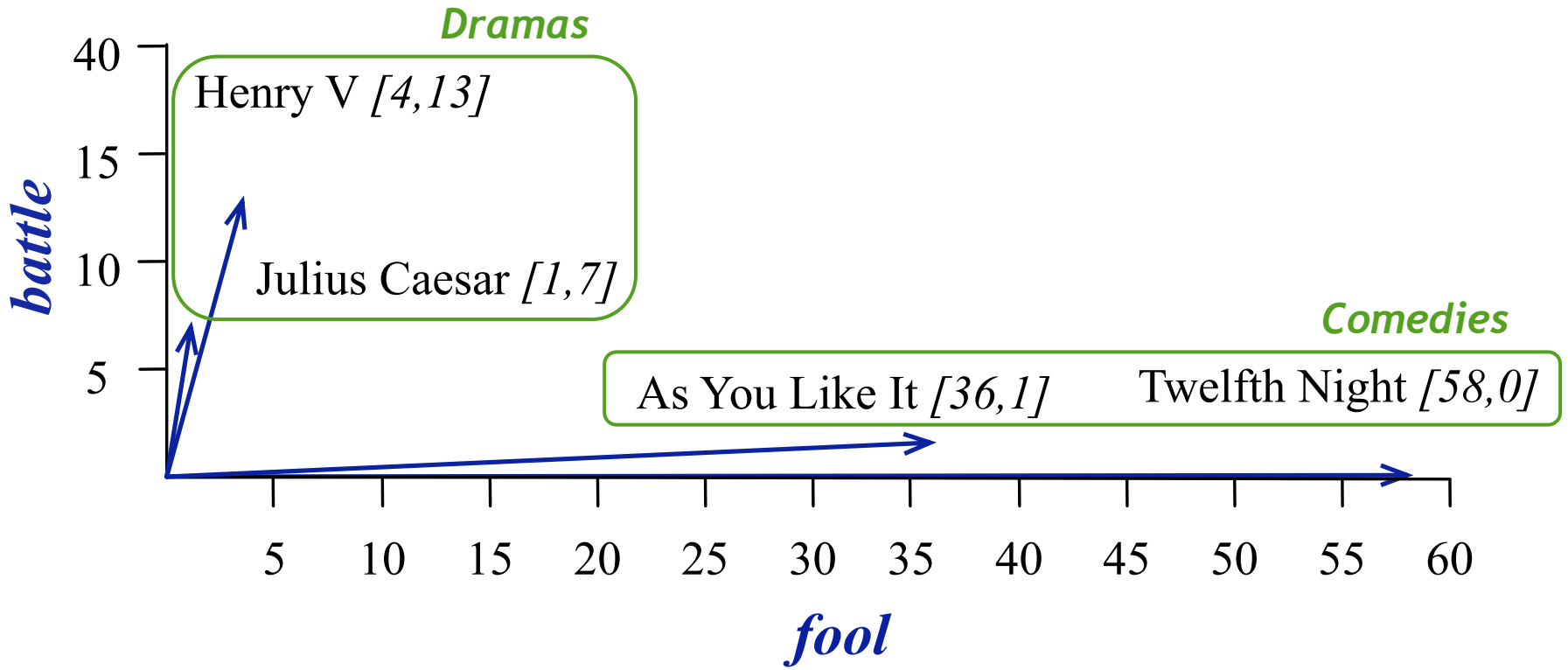
Shakespeare plays

- Word frequency vectors can be calculated for very large documents.
- For example, Shakespeare plays:

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

- Above, we are showing just four of the many dimensions of the WFV.
- Some words are more important than others for a particular task.

Looking at just two dimensions of the WFV:

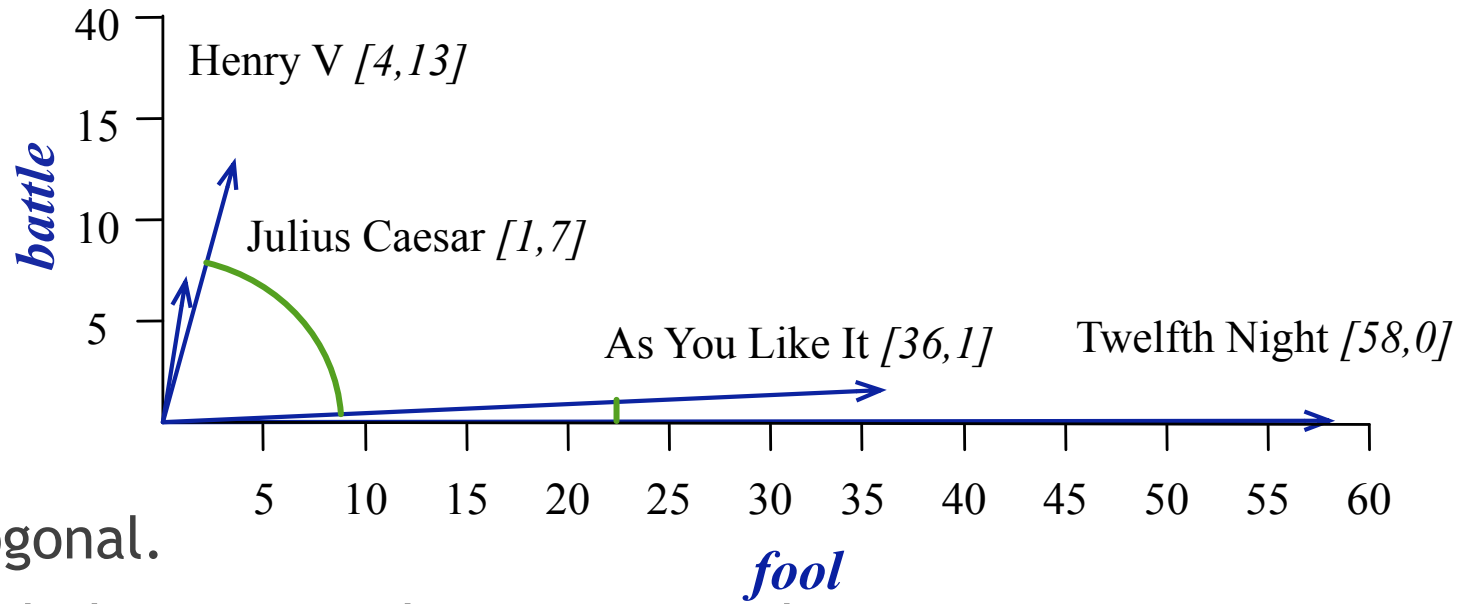


- The frequency of words *battle* and *fool* are very different for comedies and the dramas (histories and tragedies).
- These two dimensions of the WFV can help in *classifying* these documents.

Similarity of WFVs

- Word frequency vectors are often compared by the **angle** between them, which is called the **cosine similarity**:

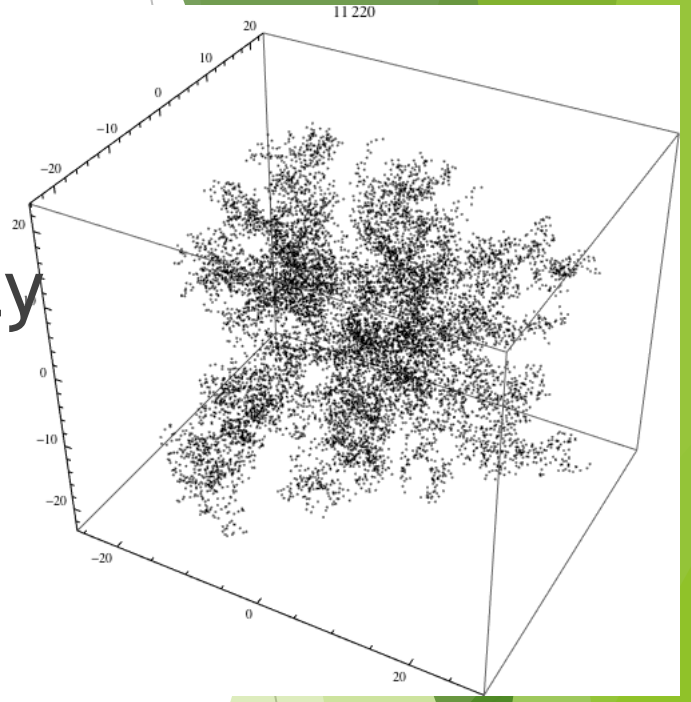
$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$



  - Cosine similarity will be 1 if WFVs are identical, 0 if they are completely orthogonal.
- Above, there is a large angle between *Julius Caesar* and *As You Like It* (their WFVs are not very similar).
- However, *As You Like It* and *Twelfth Night* are much closer (more similar)

Back to finding “positive” reviews

Each review is a point in word-frequency space:



- Using word-frequency vectors, we can represent each review as a point/vector in an extremely high-dimensional space.
  - Similar to the picture at right, but with > 10,000 dimensions instead of just 3D.
  - We assume that the positive reviews occupy some region of the space.
- If we have a few reviews that we know are positive (training data) then we can look for review that are “close” to those in the space.
  - That’s the essence of supervised machine learning.

Reducing dimension of the vectors

- If each dimension represents a word, then 10,000+ dimensions are needed.
- However, many words are similar, especially if we're ignoring ordering.
- Principle component analysis (PCA) and similar techniques can be used to reduce dimensionality.
- **Word embeddings** define each possible word as a vector in a much smaller dimension space (perhaps 200).
  - This is done by analyzing a huge language corpus, like all of Wikipedia.
  - Now, you can translate a bag of words into a vector in this 200-dimensional space by multiplying the frequency of each word by it's word embedding vector and summing them all up.