Centrality of Similarity

Lecture 5: Text Analytics for Discovering Meaning

Mark Keane, Insight/CSI, UCD

Selling Things

stock-markets

social media

science

news

polls

sentiment-id

sentiment-use

time-series

summaries

VSMs

Classifiers

Clustering

***

cosine

jaccard

dice

levenschtein

***

TF-IDF

LLR

PMI

Entropy

simple frequencies

Outline

- Centrality of Similarity
- Featural Similarity (Binary, Presence/Absence)
  - Jaccard Index/Coefficient
  - Dice Coefficient
- Featural Similarity (Frequencies) in VSMs
  - Cosine Similarity
  - Euclidean Distance
  - Manhattan Distance
- Transformational Similarity: Levenshtein, Hamming
- Structural Similarity : Analogy

Basically, its all Similarity

- Similarity is fundamental to all text analytics
- Intuitively, everyone knows what it means but formally it can take on many guises
- Maybe three distinct, high-level categories: featural, structural, transformational

Similarity is Ubiquitous

- *Retrieval*: what text-item (doc, record) is most similar to my query?
- *Clustering*: what text-items (docs, profiles, data) are similar enough to group together
- *Recommend*: which product/buyer/product-buyer-pair is most similar to another product/buyer/product-buyer-pair
- *Analogy*: are two systems of ideas the same

What to Compare…

- Ultimately, all text-items to be compared are broken down into text-pieces (word, phrases, bigrams, POS-tags, meta-tags, etc.…)
- Basic text features are often counted or measured in some way
- These frequencies can then be weighted in various ways (e.g., TF-IDF)

What Similarity to Use?

- There are distinct types of similarity: featural, structural, transformational

- Within each type many different specific similarity techniques exist

- The one you use depends on the task, what text-features you are using, how you are measuring, weighting that feature (binary, count, tf-idf score)

Similarity & Difference

- Also, keep in mind that often similarity is not just about what is the same between two things but also what is different ‘tween them

A is {a, b, c, d, e, f}
B is {a, b, c, h, i, j}
C is {a, b, c, h, i, j, x, y, z}

- Is A is as similar to B as it is to C?

Similarity

Feature Similarity:

binary, presence/absence

Simplest Case

	keyword-a	keyword-b	keyword-c	keyword-d
base	roy	harper	album	stormcock
target1	roy	rovers	comic	sport
target2	bill	harper	album	flashes
target3	roy	green	storm	cock
target4	roy	harper	album	stormcock

Intuition: Feature Overlap

Simple intuition is that if two things have a certain number of features in common then they is more similar than two things that has fewer features in common; especially, if they do not have many different features

Intuition: Feature Overlap

base	roy	harper	album	stormcock

target1	roy	rovers	comic	sport	1 (-3)
target2	bill	harper	album	flashes	2 (-2)
target3	roy	green	storm	cock	1 (- 3)
target4	roy	harper	album	stormcock	4 (0)

Intuition: Feature Overlap

base	roy	harper	album	stormcock	diffs change with size
target1	roy	rovers	comic	sport	1 (-3)
target2	bill	harper	album	flashes	2 (-2)
target3	roy	green	storm	cock	1 (- 3)
target4	roy	harper	album	stormcock	4 (0)

this is the most similar

pre-processing important

NB: this is a matrix…

	keyword-	keyword-	keyword-	keyword-d	
base	roy	harper	album	stormcock	
target1	roy	rovers	comic	sport	1 (-3)
target2	bill	harper	album	flashes	2 (-2)
target3	roy	green	storm	cock	1 (- 3)
target4	roy	harper	album	stormcock	4 (0)

NB Typically, cast as…

…a matrix of keywords with binary values…

	roy	bill	harper	comic	album	green	storm	cock	stormcock	flashes	rovers	sport
base	1	0	1	0	1	0	0	0	1	0	0	0
target1	1	0	0	1	0	0	0	0	0	0	1	1
target2	0	1	1	0	1	0	0	0	0	1	0	0
target3	1	0	0	0	0	1	1	1	0	0	0	0
target4	1	0	1	0	1	0	0	0	1	0	0	0

Jaccard Index & Distance

- **Jaccard Index** or **Jaccard Similarity Coefficient** measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets
- **Jaccard distance**, measures dissimilarity between sample sets, is obtained by subtracting the Jaccard Coefficient from 1

http://en.wikipedia.org/wiki/Jaccard_index

Jaccard Index : Formula

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

(If A and B are both empty, we define $J(A,B) = 1$.) Clearly,

$$0 \leq J(A,B) \leq 1.$$

http://en.wikipedia.org/wiki/Jaccard_index

Jaccard Distance : Formula

$$d_J(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}.$$

http://en.wikipedia.org/wiki/Jaccard_index

Jaccards: EG1

base	roy	harper	album	stormcock	
target1	roy	rovers	comic	sport	0.14
target2	bill	harper	album	flashes	0.6
target3	roy	green	storm	cock	0.14
target4	roy	harper	album	stormcock	1

Python 3.4.1: jaccard.py - /Users/user/Dropbox/Teaching.TextAnalytics/Lect5.Bits.Similarity/j...

```
def JaccardIndex(str1, str2):
    set1 = set(str1.split())
    set2 = set(str2.split())
    ans = float(len(set1 & set2)) / len(set1 | set2)
    return round(ans, 2)

base = "roy harper album stormcock"
target1 = "roy rovers comic sport"
target2 = "roy harper album flashes"
target3 = "roy green storm cock"
target4 = "roy harper album stormcock"

ans1 = JaccardIndex(base, target1)
ans2 = JaccardIndex(base, target2)
ans3 = JaccardIndex(base, target3)
ans4 = JaccardIndex(base, target4)

print([ans1, ans2, ans3,ans4])

>>>
[0.14, 0.6, 0.14, 1.0]
```

Ln: 22 Col: 0

Jaccards: EG

base	roy	harper	album	stormy			
target1	roy	rovers	comic	sport			0.14
target2	bill	harper	album	flashes			0.6
target3	roy	green	storm	cock	blip	blop	0.11
target4	roy	harper	album	stormy			1

0.14
0.6
0.14
1

Python 3.4.1: jaccard.py - /Users/user/Dropbox/Teaching.TextAnalytics/Lect5.Bits.Similarity/j...

```
def JaccardIndex(str1, str2):
    set1 = set(str1.split())
    set2 = set(str2.split())
    ans = float(len(set1 & set2)) / len(set1 | set2)
    return round(ans, 2)

base = "roy harper album stormy"
target1 = "roy rovers comic sport"
target2 = "roy harper album flashes"
target3 = "roy green storm cock blip blop"
target4 = "roy harper album stormy"

ans1 = JaccardIndex(base, target1)
ans2 = JaccardIndex(base, target2)
ans3 = JaccardIndex(base, target3)
ans4 = JaccardIndex(base, target4)

print([ans1, ans2, ans3,ans4])

>>>
[0.14, 0.6, 0.11, 1.0]
```

Ln: 21 Col: 22

Jaccards: EG

0.14
0.6
0.14
1

0.14
0.6
0.11
1

base	bleep	roy	harper	album	stormy
target1	roy	rovers	comic	sport	
target2	bill	harper	album	flashes	
target3	roy	green	storm	cock	
target4	roy	harper	album	stormy	

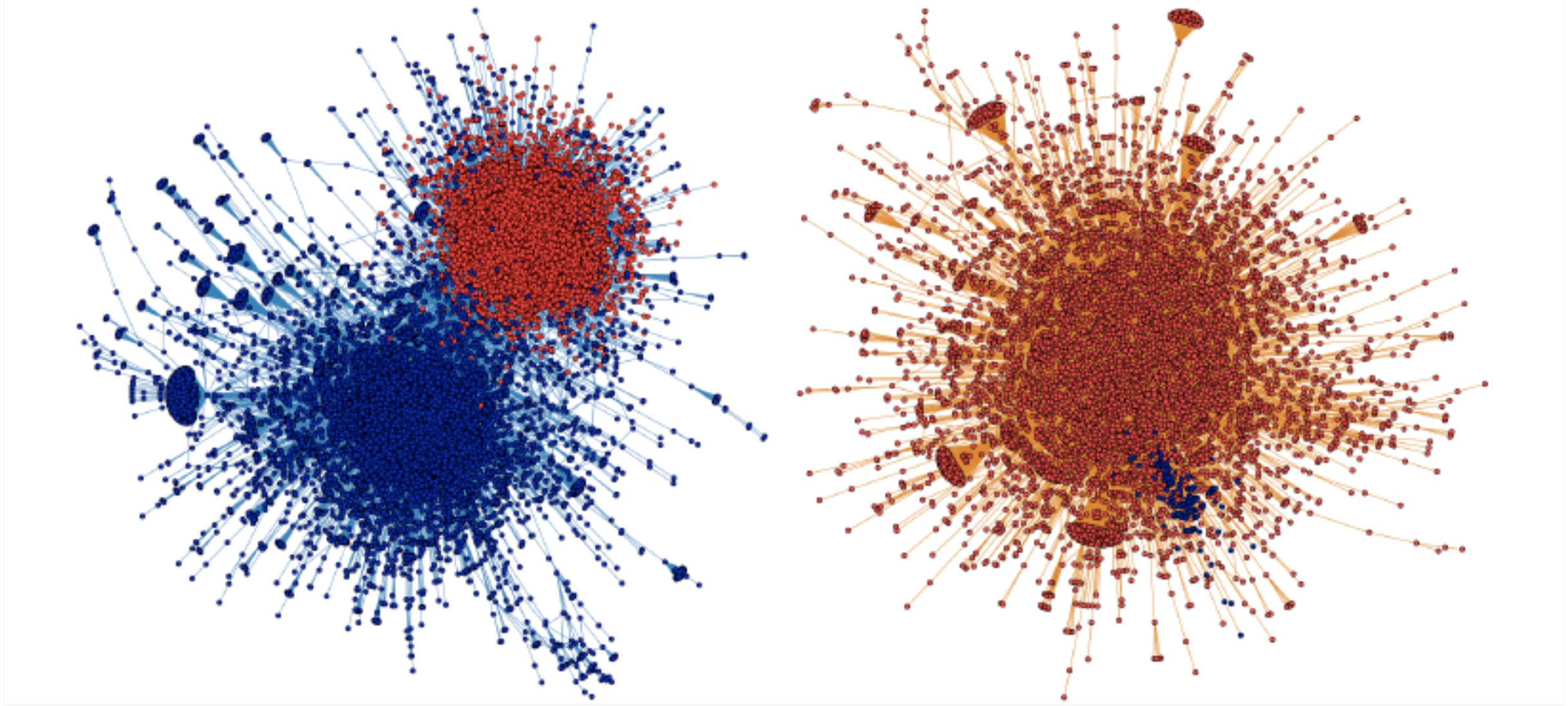
		0.12
		0.5
blip	blop	0.10
		0.80

Jaccards #1: Political Polarities

- People with different political biases post on Twitter in subtly different ways; that may be reflected in the hashtags they use (#irishwater, #dontpayforwater)
- 250K Tweets from 2010 US Congress Election
- Political retweets highly segregated, no cross-posting between left- and right-leaning users; but mention-networks are heterogenous

Conover, M., Ratkiewicz, J., Francisco, M., Gonçalves, B., Menczer, F., & Flammini, A. (2011, July). Political polarization on twitter. In ICWSM.

Jaccards #1: Political Polarities



(a) retweet network

(b) mention network

Conover, M., Ratkiewicz, J., Francisco, M., Gonçalves, B., Menczer, F., & Flammini, A. (2011, July). Political polarization on twitter. In ICWSM.

we performed a simple tag co-occurrence discovery procedure. We began by seeding our sample with the two most popular political hashtags, `#p2` ("Progressives 2.0") and `#tcot` ("Top Conservatives on Twitter"). For each seed we identified the set of hashtags with which it co-occurred in at least one tweet, and ranked the results using the Jaccard coefficient. For a set of tweets S containing a seed hashtag, and a set of tweets T containing another hashtag, the Jaccard coefficient between S and T is

$$\sigma(S,T) = \frac{|S \cap T|}{|S \cup T|}. \quad (1)$$

Thus, when the tweets in which both seed and hashtag occur make up a large portion of the tweets in which either occurs, the two are deemed to be related. Using a similarity threshold of 0.005 we identified 66 unique hashtags (Table 1), eleven of which we excluded due to overly-broad or ambiguous meaning (Table 2). This process resulted in a corpus of 252,300 politically relevant tweets. There is substantial overlap between streams associated with different political hashtags because many tweets contain multiple hashtags. As a result, lowering the similarity threshold leads to only modest increases in the number of political tweets in our sample — which do not substantially affect the results of our analysis — while introducing unrelated hashtags.

Conover, M., Ratkiewicz, J., Francisco, M., Gonçalves, B., Menczer, F., & Flammini, A. (2011, July). Political polarization on twitter. In ICWSM.

Jaccards #1: Political Polarities

Table 1: Hashtags related to #p2, #tcot, or both. Tweets containing any of these were included in our sample.

Just #p2	#casen #dadt #dc10210 #democrats #dul #fem2 #gotv #kysen #lgf #ofa #onenation #p2b #pledge #rebelleft #truthout #vote #vote2010 #whyimvotingdemocrat #youcut
Both	#cspj #dem #dems #desen #gop #hcr #nvsen #obama #ocra #p2 #p21 #phnm #politics #sgp #tcot #teaparty #tlot #topprog #tpp #twisters #votedem
Just #tcot	#912 #ampat #ftrs #glennbeck #hhrs #iamthemob #ma04 #mapoli #palin #palin12 #spwbt #tsot #tweetcongress #ucot #wethepeople

Conover, M., Ratkiewicz, J., Francisco, M., Gonçalves, B., Menczer, F., & Flammini, A. (2011, July). Political polarization on twitter. In ICWSM.

Jaccards: Final Comments

MinHash min-wise independent permutations **locality sensitive hashing** (LSH) :

- Computes Jaccard for sets, each set is represented by a constant-sized signature derived from the minimum values of a hash function
- So, LSH has been used to quickly group tweets when streaming data; parallelised in MapReduce

Dice Coefficient

- ***Dice Coefficient*** or ***Sørensen-Dice Coefficient*** is, very similar to Jaccards; used to compare similarity of two samples (sets);
- They were botanists doing research on the presence/absence of species in two locations
- Again, binary-feature similarity

Dice Coefficient: Formula

$$QS = \frac{2C}{A+B} = \frac{2|A \cap B|}{|A| + |B|}$$

where *A* and *B* are the number of species in samples A and B, respectively, and *C* is the number of species shared by the two samples; QS is the quotient of similarity and ranges between 0 and 1.[5]

http://en.wikipedia.org/wiki/Sorensen_Dice_coefficient

Dice: Eg String Similarity

When taken as a string similarity measure, the coefficient may be calculated for two strings, *x* and *y* using bigrams as follows:[7]

$$s = \frac{2n_t}{n_x + n_y}$$

where n_t is the number of character bigrams found in both strings, n_x is the number of bigrams in string *x* and n_y is the number of bigrams in string *y*. For example, to calculate the similarity between:

`night`

`nacht`

We would find the set of bigrams in each word:

{`ni`, `ig`, `gh`, `ht`}

{`na`, `ac`, `ch`, `ht`}

Each set has four elements, and the intersection of these two sets has only one element: `ht`.

Inserting these numbers into the formula, we calculate, $s = (2 \cdot 1) / (4 + 4) = 0.25$.

http://en.wikipedia.org/wiki/Sorensen_Dice_coefficient

Dice V Others

$$d = 1 - \frac{2|X \cap Y|}{|X| + |Y|}$$

- Dice and Jaccard are very similar
- Main difference is that Dice is not a proper distance metric; it does not have the property of ***triangle inequality,*** whereas Jaccard does
- Compared to Euclidean distance (later), Dice distance retains sensitivity in more heterogeneous data sets and gives less weight to outliers

Related Techniques

- Tanimoto Index
- Tversky: Soft Cardinality; Asymmetric
- and many more….

Similarity

Feature Similarity:

VSMs with TF-IDF

Intro…

- More often than not, you are not just dealing with text-items characterised by the presence/absence of certain words
- Rather, you have frequency information about the text-bits in the item; e.g. a count of the word's occurrence in the text-item
- This…changes everything

Recall…

REM

…a matrix of keywords with binary values…

	roy	bill	harper	comic	album	green	storm	cock	stormcock	flashes	rovers	sport
base	1	0	1	0	1	0	0	0	1	0	0	0
target1	1	0	0	1	0	0	0	0	0	0	1	1
target2	0	1	1	0	1	0	0	0	0	1	0	0
target3	1	0	0	0	0	1	1	1	0	0	0	0
target4	1	0	1	0	1	0	0	0	1	0	0	0

Now, you may have…VSM

…a matrix of keywords with counts (unweighted)…

	roy	bill	harper	comic	album	green	storm	cock	stormcock	flashes	rovers	sport
base	5	0	3	0	4	0	0	0	7	0	0	0
target1	5	0	0	10	0	0	0	5	0	0	3	10
target2	0	5	1	0	4	7	0	0	0	7	0	0
target3	5	0	0	0	0	3	1	1	0	0	0	0
target4	5	0	7	0	3	0	0	0	10	0	0	0

VSMs all the way down…

- Google, Amazon everything runs on vector-space models (VSMs)…Lucene
- Matrix with word-terms as rows, docs/web-pages/tweets as columns (or vice versa)
- You count frequencies of terms, weighting them on their rarity in the corpus (TF-IDF)

VSMs

- Then you use this matrix in different ways for different tasks

- Use different similarity metrics to tell you how close/far-apart vectors are in md-space

- Many different metrics: cosine similarity, euclidean distance…

G. Salton, A. Wong, and C. S. Yang (1975) A Vector Space Model for Automatic Indexing, Communications of the ACM, 18, 613-620.

Steps to Build a VSM…

1. Pre-process the terms in your text-items for the matrix (as words, stemmed-words, bigrams…); and decide on its structure (rows and columns)

2. Compute the frequency counts of the terms (for TF) and document counts (for IDF) to get TF-IDF

3. Apply your similarity method to vectors in your matrix (cosine similarity, euclidean distance)

G. Salton, A. Wong, and C. S. Yang (1975) A Vector Space Model for Automatic Indexing, Communications of the ACM, 18, 613-620.

Step 1:Pre-processing Terms…

- You want the content-bearing words in the document (not "the", "and", "there"), as in all documents
- So, remove **stop-words** (nb, no definitive set); sometimes may even have domain-specific stop-words
- 40%-50% of doc's words may be removed; nb less good as it is a language-dependent solution
- Can create some special problems for Tweets (emoji, misspellings, abbreviations, hashtags)…exclusion can sometimes dis-improve results

Step 1: Choose Matrix Structure

- Typically, Term X Document matrix (sometimes Document x Term)

- So, rows are terms that are tokenised, stop-word removed, stemmed terms…

- Columns are documents represented as collections of scored terms (tweets, web-pages, articles, news reports)

Vector Space…

- After pre-processing you have a Term X Doc Matrix or Doc X Term Matrix

	Document 1	Document 2	Document 3	Document 4	Document 5	Document 6	Document 7	Document 8
Term(s) 1	10	0	1	0	0	0	0	2
Term(s) 2	0	2	0	0	0	18	0	2
Term(s) 3	0	0	0	0	0	0	0	2
Term(s) 4	6	0	0	4	6	0	0	0
Term(s) 5	0	0	0	0	0	0	0	2
Term(s) 6	0	0	1	0	0	1	0	0
Term(s) 7	0	1	8	0	0	0	0	0
Term(s) 8	0	0	0	0	0	3	0	0

$$\begin{array}{c|cccc} & T_1 & T_2 & \ldots. & T_t \\ D_1 & w_{11} & w_{21} & \ldots & w_{t1} \\ D_2 & w_{12} & w_{22} & \ldots & w_{t2} \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ D_n & w_{1n} & w_{2n} & \ldots & w_{tn} \end{array}$$

Imagine we have 5 documents and we just count the words with little pre-processing:

d1= LSI tutorials and fast tracks

d2= books on semantic analysis

d3= learning latent semantic indexing

d4= advances in structures and advances in indexing

d5= analysis of latent structures

remove stop-words…

Imagine we have 5 documents and we just count the words with little pre-processing:

	d1	d2	d3	d4	d5
lsi					
tutorials					
fast					
tracks					
books					
semantic					
analysis					
learning					
latent					
indexing					
advances					
structures					

	d1	d2	d3	d4	d5
lsi	1	0	0	0	0
tutorials	1	0	0	0	0
fast	1	0	0	0	0
tracks	1	0	0	0	0
books	0	1	0	0	0
semantic	0	1	1	0	0
analysis	0	1	0	0	1
learning	0	1	1	0	0
latent	0	0	1	0	1
indexing	0	0	1	1	0
advances	0	0	1	2	0
structures	0	0	0	1	1

remove stop-words…choose matrix structure

Step 2: Compute TF-IDF

- Compute the frequency counts of terms (TF)
- Computer document counts (for IDF)
- Get TF-IDF for whole matrix

Imagine we have 5 documents and we just count the words with little pre-processing:

d1= LSI tutorials and fast tracks

d2= books on semantic analysis

d3= learning latent semantic indexing

d4= advances in structures and advances in indexing

d5= analysis of latent structures

remove stop-words…

do counts..

	d1	d2	d3	d4	d5
lsi	1	0	0	0	0
tutorials	1	0	0	0	0
fast	1	0	0	0	0
tracks	1	0	0	0	0
books	0	1	0	0	0
semantic	0	1	1	0	0
analysis	0	1	0	0	1
learning	0	1	1	0	0
latent	0	0	1	0	1
indexing	0	0	1	1	0
advances	0	0	1	2	0
structures	0	0	0	1	1

Formally, i*df(t, D)* is… **REM**

- The inverse of the df; the number text-items in the corpus, *N*, over the count of text-items containing the term, *t*: $N/df(t, D)$
- Typically, this is "log10ed" to smoothen the value
- And to avoid dividing by zero, we may use:

$$1 + |\{d \in D : t \in d\}|$$

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

Get TF-IDF

Then tf–idf is calculated as

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \times \mathrm{idf}(t, D)$$

	d1	d2	d3	d4	d5
lsi	1*log(5/1)	0	0	0	0
tutorials	1*log(5/1)	0	0	0	0
fast	1*log(5/1)	0	0	0	0
tracks	1*log(5/1)	0	0	0	0
books	0	1*log(5/1)	0	0	0
semantic	0	1*log(5/2)	1*log(5/2)	0	0
analysis	0	1*log(5/2)	0	0	1*log(5/2)
learning	0	0	1*log(5/1)	0	0
latent	0	0	1*log(5/2)	0	1*log(5/2)
indexing	0	0	1*log(5/2)	1*log(5/2)	0
advances	0	0	0	2*log(5/1)	0
structures	0	0	0	1*log(5/2)	1*log(5/2)

=

d1	d2	d3	d4	d5
0.6990	0	0	0	0
0.6990	0	0	0	0
0.6990	0	0	0	0
0.6990	0	0	0	0
0	0.6990	0	0	0
0	0.3979	0.3979	0	0
0	0.3979	0	0	0.3979
0	0	0.6990	0	0
0	0	0.3979	0	0.3979
0	0	0.3979	0.3979	0
0	0	0	1.3979	0
0	0	0	0.3979	0.3979

= A

number of documents

1*log(5/1) → number of documents with this term

term frequency (the term LSI occurs just once)

http://yen.tw/?p=43

…compute Tf-IDF…

So, a VSM is

Then tf–idf is calculated as

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \times \mathrm{idf}(t, D)$$

In the classic vector space model proposed by Salton, Wong and Yang [1] the term-specific weights in the document vectors are products of local and global parameters. The model is known as term frequency-inverse document frequency model. The weight vector for document *d* is $\mathbf{v}_d = [w_{1,d}, w_{2,d}, \ldots, w_{N,d}]^T$, where

$$w_{t,d} = \mathrm{tf}_{t,d} \cdot \log \frac{|D|}{|\{d' \in D \,|\, t \in d'\}|}$$

and

- $\mathrm{tf}_{t,d}$ is term frequency of term *t* in document *d* (a local parameter)
- $\log \frac{|D|}{|\{d' \in D \,|\, t \in d'\}|}$ is inverse document frequency (a global parameter). $|D|$ is the total number of documents in the document set; $|\{d' \in D \,|\, t \in d'\}|$ is the number of documents containing the term *t*.

G. Salton, A. Wong, and C. S. Yang (1975) A Vector Space Model for Automatic Indexing, Communications of the ACM, 18, 613-620.

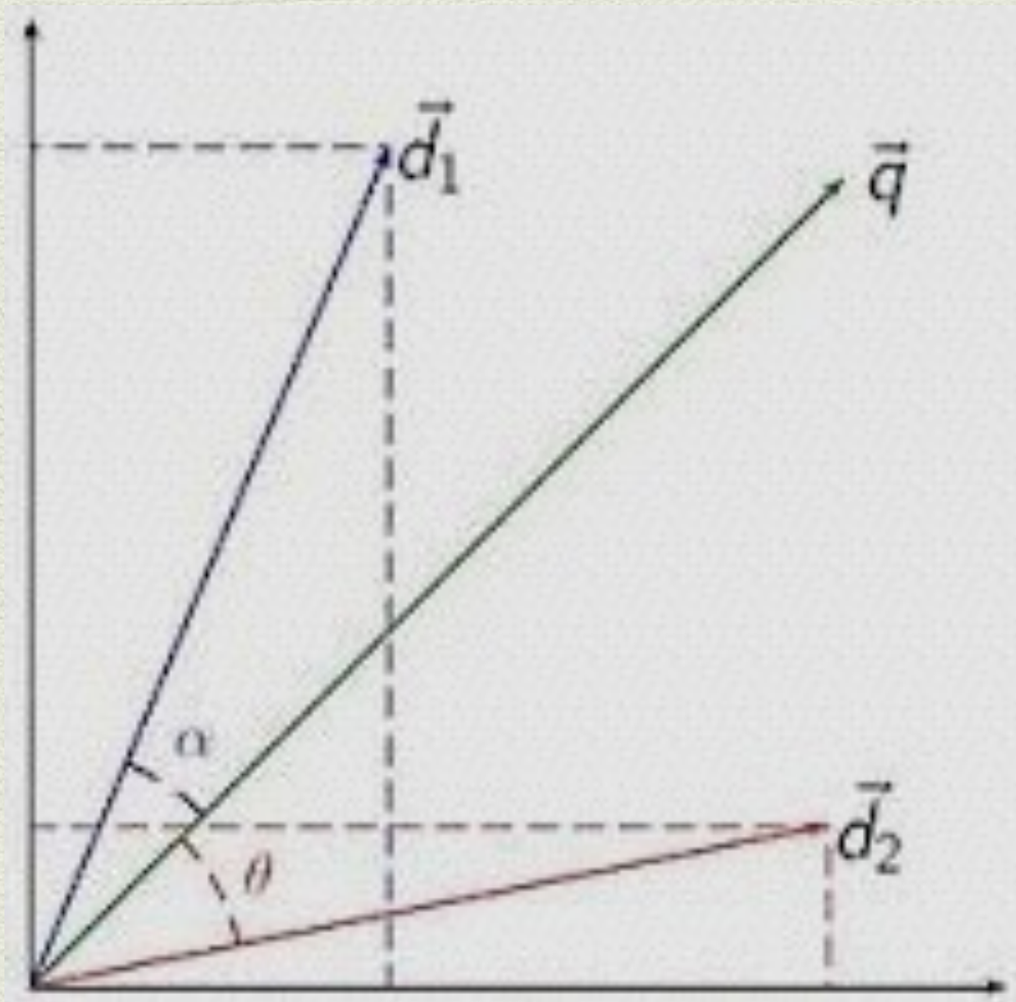
Step 3: Apply Similarity Method

- Similarity method is applied to vectors
- Tells you about similarity of documents to one another or similarity to a query-vector; how close they are
- *Distance* between vectors tells you how dissimilar they are; how far apart they are
- Many ways to compute these: cosine similarity, euclidean distance…

Cosine Similarity: The Idea

- You are getting the size of the angle between one document and the next (d1 and d2) or between a query (1) and some document (d1)

- The smaller the angle (cosine) the more similar are the vectors



Cosine Similarity

Definition [edit]

The cosine of two vectors can be derived by using the Euclidean dot product formula:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta$$

Given two vectors of attributes, *A* and *B*, the cosine similarity, *cos(θ)*, is represented using a dot product and magnitude as

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

The resulting similarity ranges from −1 meaning exactly opposite, to 1 meaning exactly the same, with 0 usually indicating independence, and in-between values indicating intermediate similarity or dissimilarity.

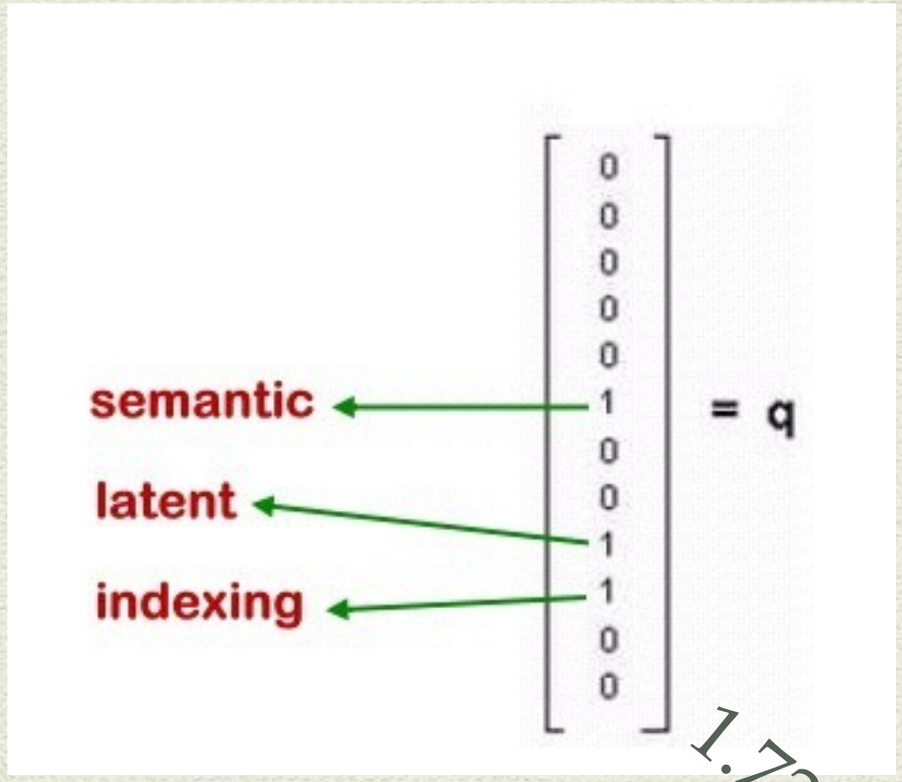
For text matching, the attribute vectors *A* and *B* are usually the term frequency vectors of the documents. The cosine similarity can be seen as a method of normalizing document length during comparison.

In the case of information retrieval, the cosine similarity of two documents will range from 0 to 1, since the term frequencies (tf-idf weights) cannot be negative. The angle between two term frequency vectors cannot be greater than 90°.

http://en.wikipedia.org/wiki/Cosine_similarity

d1	d2	d3	d4	d5	
0.6990	0	0	0	0	
0.6990	0	0	0	0	
0.6990	0	0	0	0	
0.6990	0	0	0	0	
0	0.6990	0	0	0	
0	0.3979	0.3979	0	0	
0	0.3979	0	0	0.3979	= A
0	0	0.6990	0	0	
0	0	0.3979	0	0.3979	
0	0	0.3979	0.3979	0	
0	0	0	1.3979	0	
0	0	0	0.3979	0.3979	

1.3980 0.8973 0.9816 1.5069 0.6891



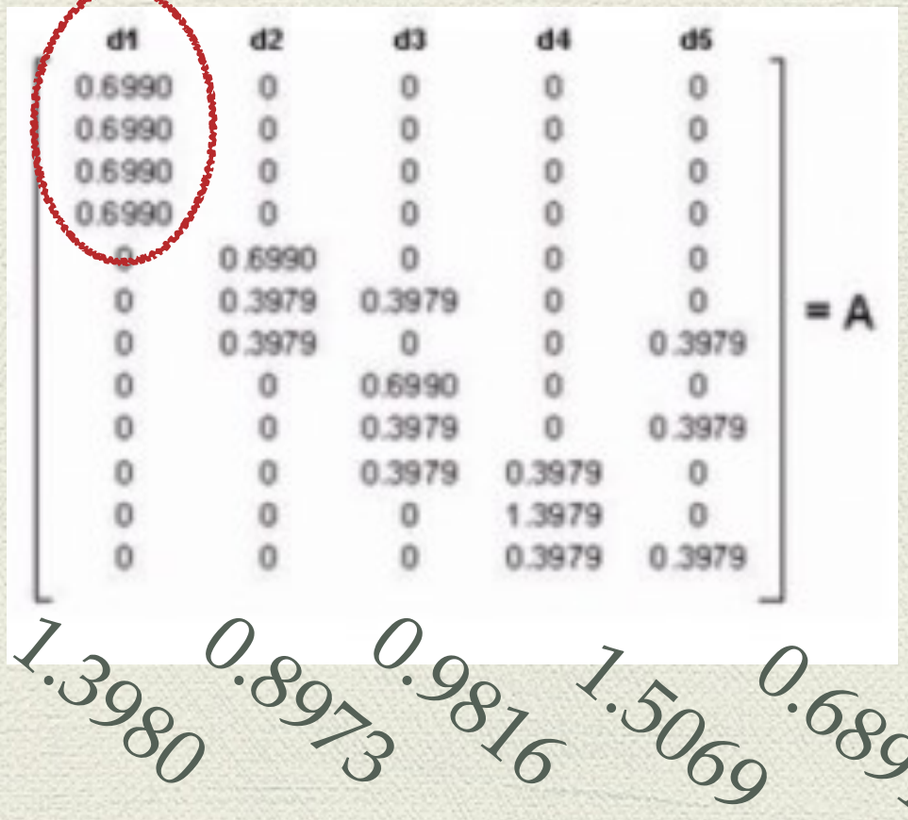
<- vector length ->

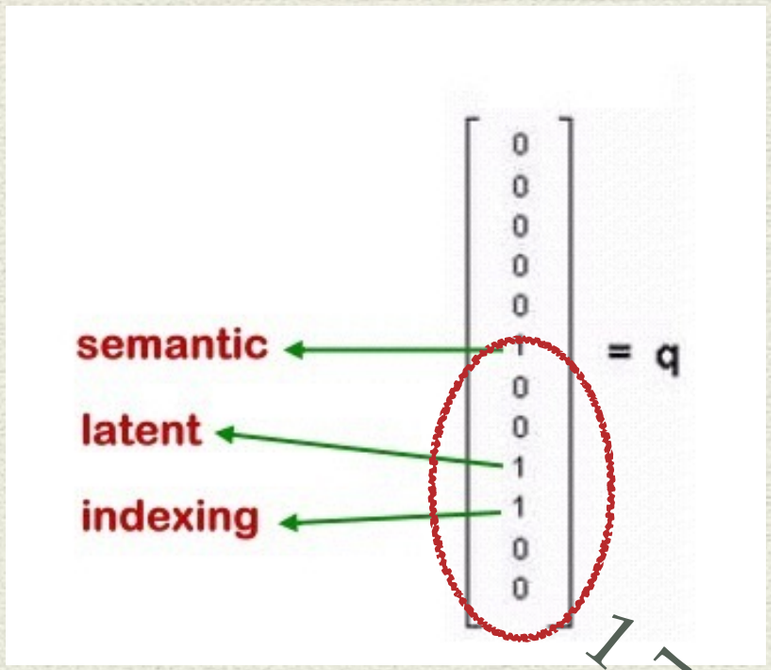
d1 = sq-root((0.699 x 0.699) +
(0.699 x 0.699) +
(0.699 x 0.699) +
(0.699 x 0.699)) = 1.3980

$$\frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

q = sq-root((1 x 1)+(1 x 1)+(1 x 1)) = 1.7321

http://yen.tw/?p=43





d1 . q = 0

d2 . q = 0.2560

d3 . q = 0.7022

d4 . q = 0.1524

d5 . q = 0.3334

Trivially, **d1** and **q** must be 0.

$$\frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

Can see why **d3** and **q** are most similar

“learning latent semantic indexing” -> “latent semantic indexing”

relevance ordering to **q** is **d3>d5>d2>d4>d1**

http://yen.tw/?p=43

```
__author__ = 'user'
# bits from http://stackoverflow.com/questions/15173225/how-to-calculate-cosine-similarity-given-2-sentence-strings-python
# load_docs, process_docs and compute_vector by MK
import math
from collections import Counter

vector_dict = {}

#Just loads in all the documents
def load_docs():
 print("Loading docs...")
 doc1=('d1', 'LSI tutorials and fast tracks')
 doc2=('d2', 'books on semantic analysis')
 doc3=('d3', 'learning latent semantic indexing')
 doc4=('d4', 'advances in structures and advances in indexing')
 doc5=('d5', 'analysis of latent structures')
 return [doc1, doc2,doc3,doc4,doc5]

#Computes TF for words in each doc, DF for all features in all docs; finally whole Tf-IDF matrix
def process_docs(all_dcs):...


#computes TF-IDF for all words in all docs
def compute_vector_len(doc_dict, no, df_counts):...

def get_cosine(text1, text2):...

#RUN
all_docs = load_docs()
process_docs(all_docs)
vector_dict['q'] = {'semantic' : 1, 'latent' : 1, 'indexing' : 1}

for keys,values in vector_dict.items(): print(keys, values)

text1 = 'd3'
text2 = 'q'
cosine = get_cosine(text1, text2)
print('Cosine:', cosine)
```

```
__author__ = 'user'
# bits from http://stackoverflow.com/questions/15173225/how-to-calculate-cosine-similarity-given-2-sentence-strings-python
# load_docs, process_docs and compute_vector by MK
import math
from collections import Counter

vector_dict = {}

#Just loads in all the documents
def load_docs():...

#Computes TF for words in each doc, DF for all features in all docs; finally whole Tf-IDF matrix
def process_docs(all_dcs):
 stop_words = [ 'of', 'and', 'on','in' ]
 all_words = []
 counts_dict = {}
 for doc in all_dcs:
    words = [x.lower() for x in doc[1].split() if x not in stop_words]
    words_counted = Counter(words)
    unique_words = list(words_counted.keys())
    counts_dict[doc[0]] = words_counted
    all_words = all_words + unique_words
 n = len(counts_dict)
 df_counts = Counter(all_words)
 compute_vector_len(counts_dict, n, df_counts)


#computes TF-IDF for all words in all docs
def compute_vector_len(doc_dict, no, df_counts):
  global vector_dict
  for doc_name in doc_dict:
    doc_words = doc_dict[doc_name].keys()
    wd_tfidf_scores = {}
    for wd in list(set(doc_words)):
        wds_cts = doc_dict[doc_name]
        wd_tf_idf = wds_cts[wd] * math.log(no / df_counts[wd], 10)
        wd_tfidf_scores[wd] = round(wd_tf_idf, 4)
    vector_dict[doc_name] = wd_tfidf_scores

def get_cosine(text1, text2):...

#RUN
all_docs = load_docs()
process_docs(all_docs)
vector_dict['q'] = {'semantic' : 1, 'latent' : 1, 'indexing' : 1}
```

```
__author__ = 'user'
# bits from http://stackoverflow.com/questions/15173225/how-to-calculate-cosine-similarity-given-2-sentence-strings-python
# load_docs, process_docs and compute_vector by MK
import math
from collections import Counter

vector_dict = {}

#Just loads in all the documents
def load_docs():...

#Computes TF for words in each doc, DF for all features in all docs; finally whole Tf-IDF matrix
def process_docs(all_dcs):...


#computes TF-IDF for all words in all docs
def compute_vector_len(doc_dict, no, df_counts):...

def get_cosine(text1, text2):
     vec1 = vector_dict[text1]
     vec2 = vector_dict[text2]
     intersection = set(vec1.keys()) & set(vec2.keys())
     numerator = sum([vec1[x] * vec2[x] for x in intersection])
     sum1 = sum([vec1[x]**2 for x in vec1.keys()])
     sum2 = sum([vec2[x]**2 for x in vec2.keys()])
     denominator = math.sqrt(sum1) * math.sqrt(sum2)
     if not denominator:
        return 0.0
     else:
        return round(float(numerator) / denominator, 3)

#RUN
all_docs = load_docs()
process_docs(all_docs)
vector_dict['q'] = {'semantic' : 1, 'latent' : 1, 'indexing' : 1}

for keys,values in vector_dict.items(): print(keys, values)

text1 = 'd3'
text2 = 'q'
cosine = get_cosine(text1, text2)
print('Cosine:', cosine)
```

Cosine.py - Lect5.Progs - [~/Dropbox/Teaching.TextAnalytics/Lect5.Similarity/Lect5.Progs]

Lect5.Progs > Cosine.py

Proj...

Lect5.Progs (~/Dropb
- Cosine.py
- CosineCommented.
- jaccard.py
- levenschtein.py
- saasda.xlsx

External Libraries

levenschtein.py | Cosine.py | jaccard.py

```
__author__ = 'user'
# bits from http://stackoverflow.com/questions/15173225/how-to-calculate-cosine-similarity-given-2-sentence-strings
# load_docs, process_docs and compute_vector by MK
import math
from collections import Counter

vector_dict = {}

#Just loads in all the documents
def load_docs():...

#Computes TF for words in each doc, DF for all features in all docs; finally whole Tf-IDF matrix
def process_docs(all_dcs):...


#computes TF-IDF for all words in all docs
def compute_vector_len(doc_dict, no, df_counts):...

def get_cosine(text1, text2):...

#RUN
all_docs = load_docs()
process_docs(all_docs)
vector_dict['q'] = {'semantic' : 1, 'latent' : 1, 'indexing' : 1}

for keys,values in vector_dict.items(): print(keys, values)

text1 = 'd3'
text2 = 'q'
cosine = get_cosine(text1, text2)
print('Cosine:', cosine)
```

Run Cosine

```
/Library/Frameworks/Python.framework/Versions/3.4/bin/python3.4 /Users/user/Dropbox/Teaching.TextAnalytics/Lect5.Similarity/Lect5.Prog
Loading docs...
d4 {'structures': 0.3979, 'indexing': 0.3979, 'advances': 1.3979}
q {'semantic': 1, 'indexing': 1, 'latent': 1}
d1 {'tutorials': 0.699, 'fast': 0.699, 'tracks': 0.699, 'lsi': 0.699}
d5 {'analysis': 0.3979, 'structures': 0.3979, 'latent': 0.3979}
d2 {'books': 0.699, 'semantic': 0.3979, 'analysis': 0.3979}
d3 {'semantic': 0.3979, 'indexing': 0.3979, 'learning': 0.699, 'latent': 0.3979}
Cosine: 0.702

Process finished with exit code 0
```

Steps to Build a VSM… **REM**

1. Pre-process the terms in your text-items for the matrix (as words, stemmed-words, bigrams…); and decide on its structure (rows and columns)

2. Compute the frequency counts of the terms (for TF) and document counts (for IDF) to get TF-IDF

3. Apply your similarity method to vectors in your matrix (cosine similarity, euclidean distance)

G. Salton, A. Wong, and C. S. Yang (1975) A Vector Space Model for Automatic Indexing, Communications of the ACM, 18, 613-620.

Euclidean Distance: The Idea

- You are getting the distance between two vectors that characterise two things in some n-dimensional space (dissimilarity)

- The greater the distance the less similar they are to one another, the smaller the distance the more similar

Euclidean Distance

The **Euclidean distance** between points **p** and **q** is the length of the line segment connecting them ($\overline{\mathbf{pq}}$).

In Cartesian coordinates, if **p** = (p_1, p_2,..., p_n) and **q** = (q_1, q_2,..., q_n) are two points in Euclidean *n*-space, then the distance (d) from **p** to **q**, or from **q** to **p** is given by the Pythagorean formula:

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\cdots+(q_n-p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}. \qquad (1)$$

The position of a point in a Euclidean *n*-space is a Euclidean vector. So, **p** and **q** are Euclidean vectors, starting from the origin of the space, and their tips indicate two points. The **Euclidean norm**, or **Euclidean length**, or **magnitude** of a vector measures the length of the vector:

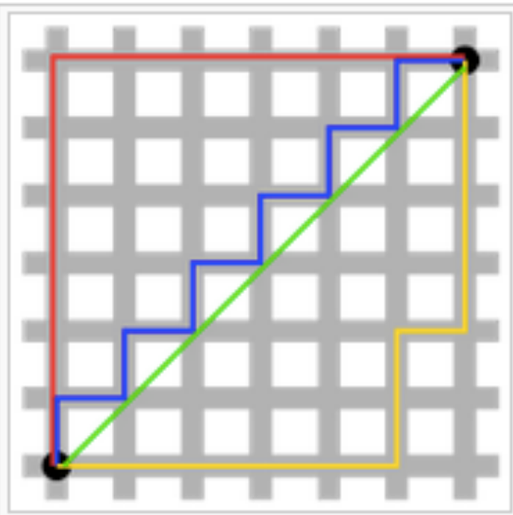
$$\|\mathbf{p}\| = \sqrt{p_1^2+p_2^2+\cdots+p_n^2} = \sqrt{\mathbf{p}\cdot\mathbf{p}}$$

where the last equation involves the dot product.

http://en.wikipedia.org/wiki/Euclidean_distance

Manhattan Distance

Taxicab geometry, considered by Hermann Minkowski in 19th century Germany, is a form of geometry in which the usual distance function of metric or Euclidean geometry is replaced by a new metric in which the distance between two points is the sum of the absolute differences of their Cartesian coordinates. The taxicab metric is also known as **rectilinear distance**, **L_1 distance** or **ℓ_1 norm** (see L^p space), **city block distance**, **Manhattan distance**, or **Manhattan length**, with corresponding variations in the name of the geometry.[1] The latter names allude to the grid layout of most streets on the island of Manhattan, which causes the shortest path a car could take between two intersections in the borough to have length equal to the intersections' distance in taxicab geometry.



Taxicab geometry versus Euclidean distance: In taxicab geometry all three pictured lines (red, yellow, and blue) have the same length (12). In Euclidean geometry, the green line has length $6\sqrt{2} \approx 8.49$, and is the unique shortest path.

The taxicab distance, d_1, between two vectors $\mathbf{p}, \mathbf{q}$ in an *n*-dimensional real vector space with fixed Cartesian coordinate system, is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes. More formally,

$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|,$$

where $(\mathbf{p}, \mathbf{q})$ are vectors

$$\mathbf{p} = (p_1, p_2, \ldots, p_n) \text{ and } \mathbf{q} = (q_1, q_2, \ldots, q_n)$$

For example, in the plane, the taxicab distance between (p_1, p_2) and (q_1, q_2) is $|p_1 - q_1| + |p_2 - q_2|$.

http://en.wikipedia.org/wiki/Taxicab_geometry

Steps to Build a VSM… **REM**

1. Pre-process the terms in your text-items for the matrix (as words, stemmed-words, bigrams…); and decide on its structure (rows and columns)

2. Compute the frequency counts of the terms (for tf) and document counts (for idf) to get tf-idf

3. Apply your similarity method to vectors in your matrix (cosine similarity, euclidean distance)

G. Salton, A. Wong, and C. S. Yang (1975) A Vector Space Model for Automatic Indexing, Communications of the ACM, 18, 613-620.

VSMs: VIMP

- Google, Amazon,Yahoo, Facebook…
- Most search nowadays runs off VSMs…often using Lucene (en.wikipedia.org/wiki/Lucene)
- Fundamentally, bag-of-words approach can be extended to structured-meaning (later)

Egs…

- So, common we don't need any !
- Most of classification and clustering lectures will be using this similarity technique; a tf-idf vector and cosine similarity…

VSM: Pros

- Simple model based on linear algebra
- Allows computing a continuous degree of similarity between queries and documents
- Allows ranking documents according to their possible relevance and partial matching
- Extensions to basic form prove important

VSM: Cons

- Weighting is intuitive, not formal
- A lot of information is thrown away:
  - order in which terms appear is lost
  - doc is a bag of words; terms are statistically independent
  - need (exact) matching of strings (not words), not substrings
- Long documents are poorly represented because they have poor similarity values (a small scalar product and a large dimensionality); sparse matrices
- Semantic sensitivity; documents with similar context but different term vocabulary won't be associated

Similarity

Transformational Similarity

What Similarity to Use? **REM**

- There are distinct types of similarity: featural, transformational, structural

- Within each type many different specific similarity techniques exist….

…by far, the most commonly use form of similarity…

What Similarity to Use?

- Featural similarity is based on the intuition that if two things "look" the same they are the same; they have the same features
- But, similarity is not only skin-deep; two things may looks the same but have different deep structure (similar birds, different genetics; deja vu plans)
- The transformational similarity intuition is: if you can turn one thing into another easily then they are similar

Transformational Similarity…

- *Transformational similarity is based on the idea if you can turn one thing into another easily then they are similar*

- "turn one thing" implies you can apply some procedure / transformation / some set of steps to change one thing into the other

Transformational Similarity…

- *"ABCDEFG"* and *"ABCDEGF"* ; are similar 'cos one pair reversal changes one to other

- *"ABCDEFG"* and *"ABCEDGF"* ; are less similar 'cos I need, at least, two reversals to change one to the other

- But, note, possible transformation steps will define the similarity distance (reverse v subst.)

Levenstein: Definition

In information theory and computer science, the **Levenshtein distance** is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. It is named after Vladimir Levenshtein, who considered this distance in 1965.[1]

Levenshtein distance may also be referred to as **edit distance**, although that may also denote a larger family of distance metrics.[2]:32 It is closely related to pairwise string alignments.

- So, transformations are the simplest character edits that you can carry out: delete, insert, substitute

- Basically, transformational similarity (edit distance) is the no. of transformations required to turn one thing into the other

http://en.wikipedia.org/wiki/Levenshtein_distance

Levenstein: Formula

Mathematically, the Levenshtein distance between two strings a, b is given by $\operatorname{lev}_{a,b}(|a|,|b|)$ where

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

Note that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same.

Example [edit]

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

1. **k**itten → **s**itten (substitution of "s" for "k")
2. sitt**e**n → sitt**i**n (substitution of "i" for "e")
3. sittin → sittin**g** (insertion of "g" at the end).

Upper and lower bounds [edit]

The Levenshtein distance has several simple upper and lower bounds. These include:

- It is always at least the difference of the sizes of the two strings.
- It is at most the length of the longer string.
- It is zero if and only if the strings are equal.
- If the strings are the same size, the Hamming distance is an upper bound on the Levenshtein distance.
- The Levenshtein distance between two strings is no greater than the sum of their Levenshtein distances from a third string (triangle inequality).

Levenshtein: EG

“dr keane” -> “mr bean”

“mr atkinson” -> “mr bean”

- “rain” -> “sain” -> “shin” -> “shine”.
- operations could have been done in other orders,
- but at least three steps are needed.

levenschtein.py - Lect5.Progs - [~/Dropbox/Teaching.TextAnalytics/Lect5.Similarity/Lect5.Progs]

levenschtein

Lect5.Progs > levenschtein.py

Project

Lect5.Progs (~/Dropbox/Teaching.TextAnalytics/Lec

- jaccard.py
- levenschtein.py
- saasda.xlsx
- VSMCosine.py
- VSMCosine2.py

External Libraries

VSMCosine.py | VSMCosine2.py | levenschtein.py | jaccard.py

```
__author__ = 'user'

import nltk.metrics
import distance

#  transposition flag allows transpositions edits (e.g., "ab" -> "ba"),

s1 = 'dr mark keane'
s2 = 'mr mark bean'

s3 = 'rain'
s4 = 'shine'

s5 = 'mr rowan atkinson'
s6 = 'mr bean'

ans = nltk.metrics.distance.edit_distance(s1, s2, transpositions=False)
print(ans)

ans = nltk.metrics.distance.edit_distance(s3, s4, transpositions=False)
print(ans)

ans = nltk.metrics.distance.edit_distance(s5, s6, transpositions=False)
print(ans)

ans = distance.levenshtein(s1, s2)
print(ans)

ans = distance.levenshtein(s3, s4)
print(ans)

ans = distance.levenshtein(s5, s6)
print(ans)
```

Run levenschtein

```
/Library/Frameworks/Python.framework/Versions/3.4/b
3
3
12
3
3
12

Process finished with exit code 0
```

1: Project | 7: Structure | 2: Favorites

Python Console | Terminal | 4: Run | 6: TODO

Packages installed successfully: Installed packages: 'Distance' (4 minutes ago)

Levenshtein: Eg#1

- Plagiarism seen as insert, deletion, substitution at multiple-levels of a doc: word (0), period (1), paragraph (2)
- Special case of duplicate document detection
- Two chunks edit distance equivalent if plagiarism function > threshold

Let $s1$ be a sub-sequence of chunks belonging to the same level L extracted from the original document. Let $s2$ be the sequence of chunks produced by the plagiarist starting from $s1$.
Let a,b,c be chunks of the same level L. We observe that the plagiarist may perform the following simple actions:

- Insertion
  The plagiarist may start from the original sequence $s1$ and insert a chunk.
  $s1 = ab \rightarrow s2 = acb$
- Deletion
  The plagiarist may start from the original sequence of chunks $s1$ and delete a chunk.
  $s1 = acb \rightarrow s2 = ab$
- Change
  The plagiarist may start from the original sequence of chunks $s1$ and substitute a chunk.
  $S1 = acb \rightarrow s2 = adb$

Zini, M., et al. (2006). Plagiarism detection through multilevel text comparison. *AXMEDIS'06* (pp. 181-185). IEEE.

Levenshtein: Eg#1

$$\lambda(c) := \text{the number of contained sub-chunks} \quad (1)$$

$$|c| := \text{the character length of chunk c} \quad (2)$$

$\sigma(c^L)$ is the set of sub-chunks of level L-1 in c^L. (3)

Let c_i^L and c_j^L be two level L chunks such that $c_i^L \in D_i$ and $c_j^L \in D_j$,

We define chunk similarity of level L:

$$\xi(c_i^L, c_j^L) = \frac{\sum w(c_i^{L-1}, c_j^{L-1}) P(c_i^{L-1}, c_j^{L-1})}{\sum w(c_i^{L-1}, c_j^{L-1})} \quad (5)$$

where $(c_i^{L-1}, c_j^{L-1}) \in \sigma(c_i^L) \times \sigma(c_j^L)$,

$$w(c_i^{L-1}, c_j^{L-1}) = \max\left(\frac{|c_i^{L-1}|}{|c_i^L|}, \frac{|c_j^{L-1}|}{|c_j^L|}\right)$$

and σ is defined as in (3)

Let D_i and D_j be two documents. Let c_i^L and c_j^L be two level L chunks such that $c_i^L \in D_i$ and $c_j^L \in D_j$.

We define the structural similarity as:

$$\zeta(c_i^L, c_j^L) = 1 - \frac{ed(c_i^L, c_j^L)}{\max(\lambda(c_i^L), \lambda(c_j^L))} \quad (4)$$

where $ed()$ is the edit distance between the two chunks as defined in section 4.2 and $\lambda(c)$ is defined as in (1).

Having defined the structural and chunk similarity function we are now able to define the plagiarism function as:

$$P(c_i^L, c_j^L) = \begin{cases} \frac{\alpha_L \xi(c_i^L, c_j^L) + \beta_L \zeta(c_i^L, c_j^L)}{\alpha_L + \beta_L} & if\, L > 0 \\ \zeta(c_i^L, c_j^L) & if\, L = 0 \end{cases} \quad (6)$$

where α_L and β_L are adjustable weight parameters.

Zini, M., et al. (2006). Plagiarism detection through multilevel text comparison. *AXMEDIS'06* (pp. 181-185). IEEE.

Levenshtein: Eg#2

- SPAM Tweets are very common
- Need to cleaned from analyses of trending hashtags
- Removed from what you are following

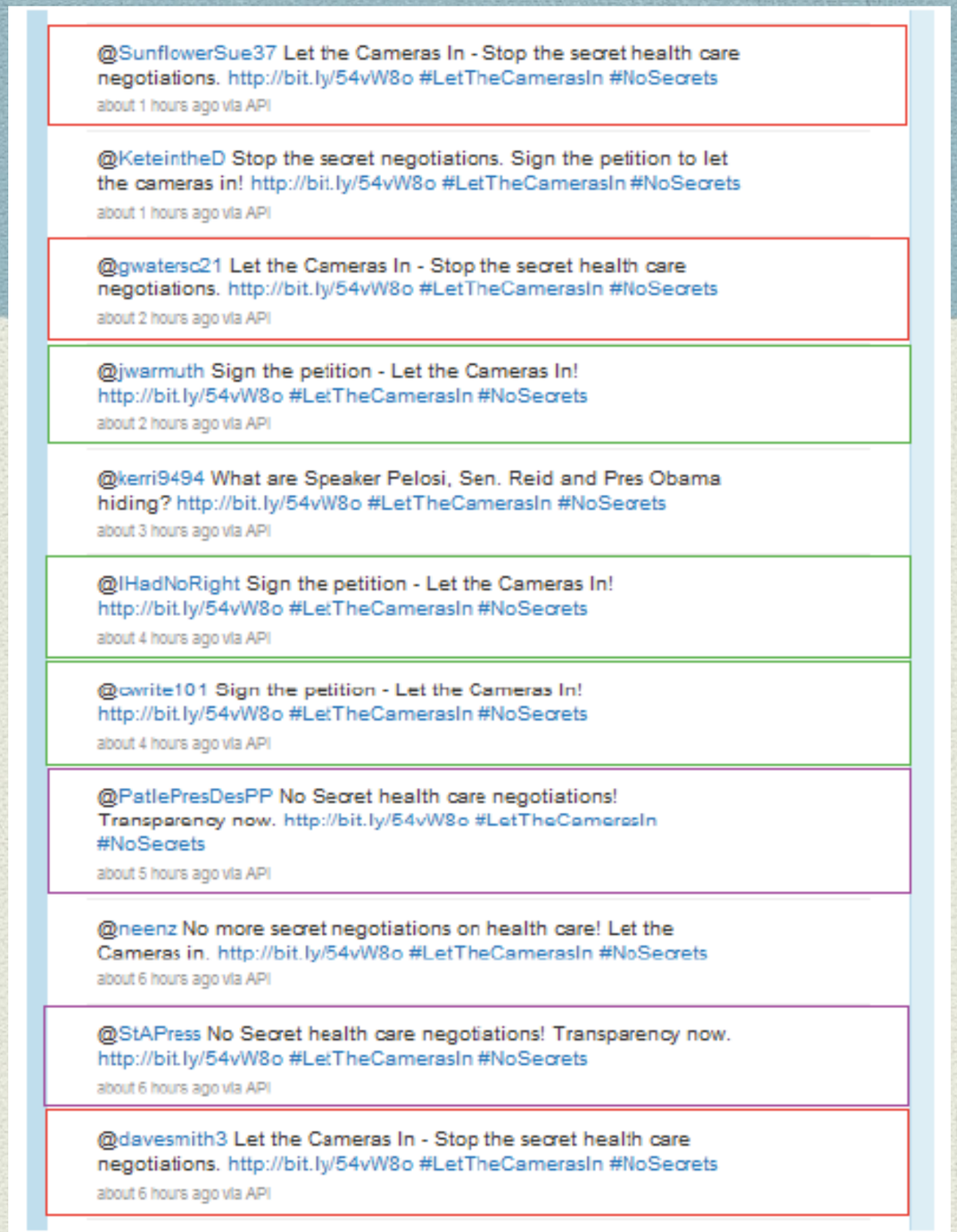


Figure 4: A Twitter spam page (Duplicate tweets are circled in the same color rectangles)

Wang, A. H. (2010, July). Don't follow me: Spam detection in twitter. In Security and Cryptography (SECRYPT) (pp. 1-10). IEEE.

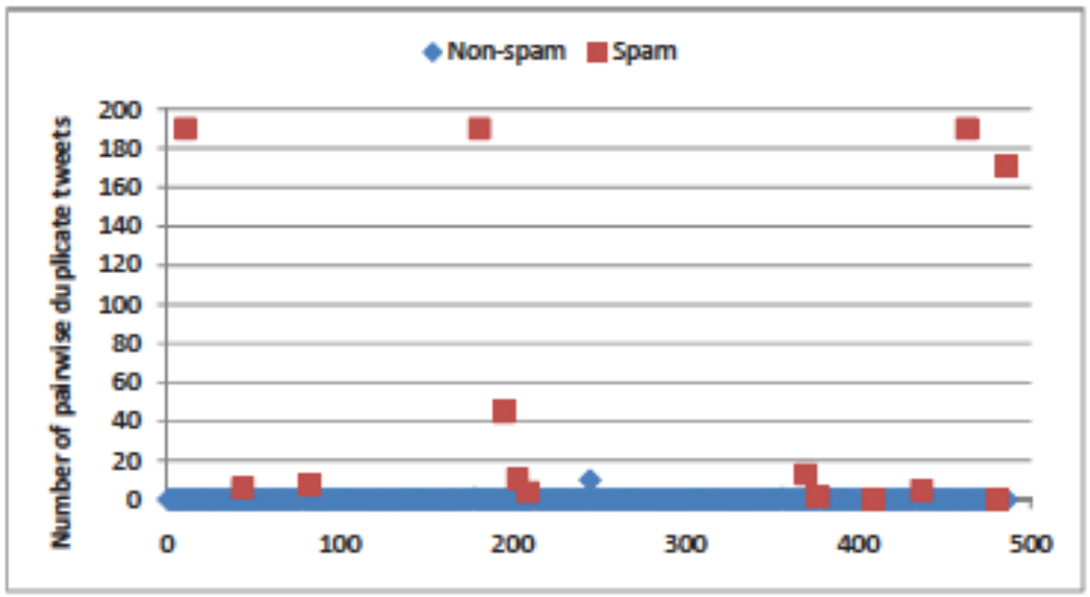
Levenstein: Eg2

- SPAM Tweets take common forms: changes to username, tiny-url, include picture, insert multiple hashtags

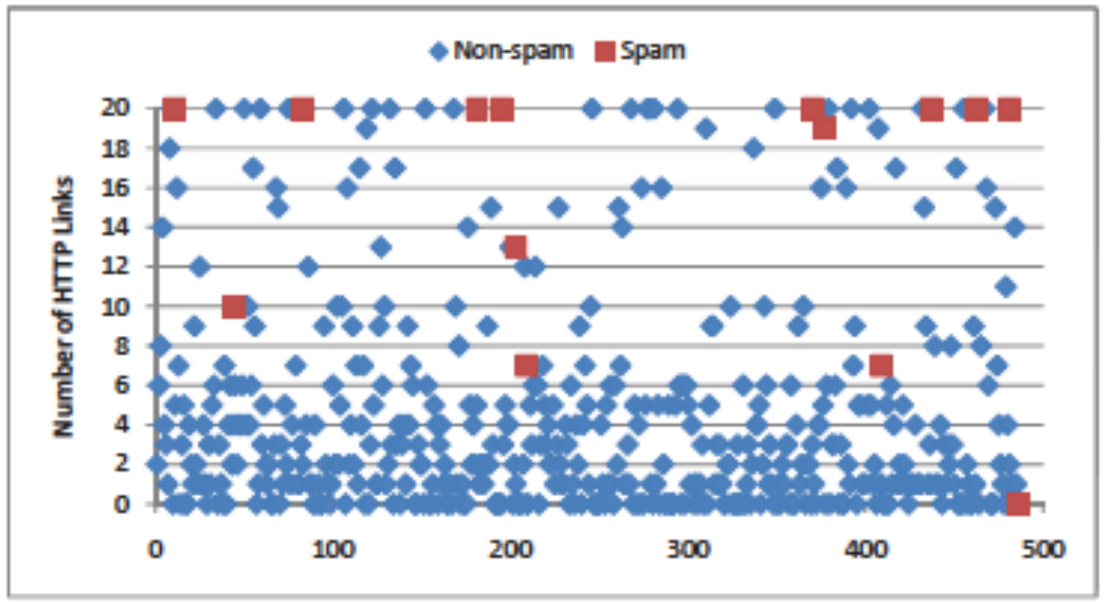
- Uses classifier and manual evaluation of the last 20 tweets from spam and non-spam accounts to look at features of importance

Wang, A. H. (2010, July). Don't follow me: Spam detection in twitter. SECRYPT-10, 1-10. IEEE.

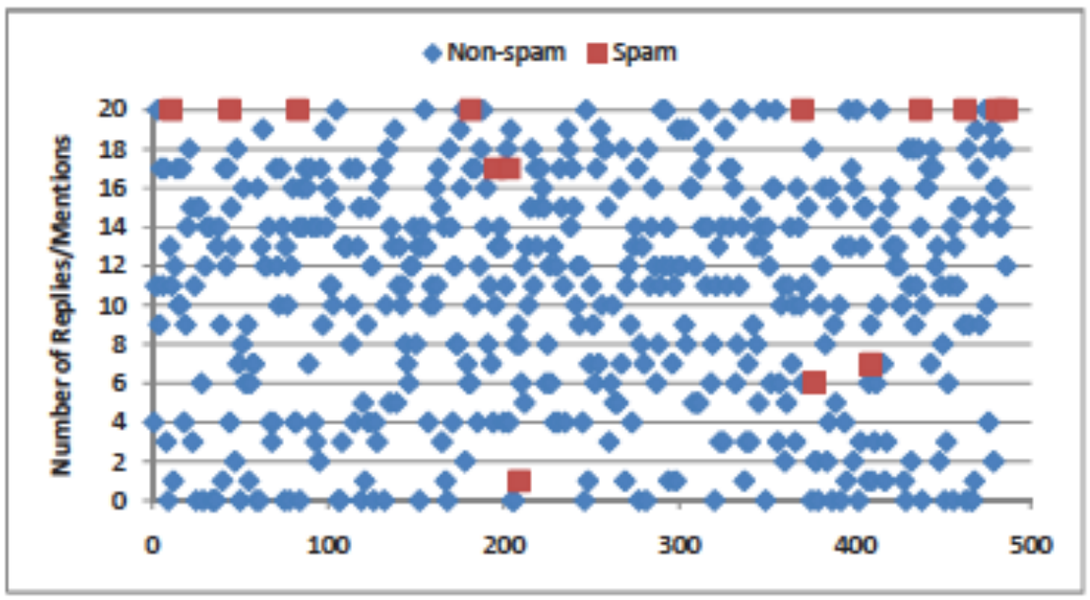
Levenstein: Eg2



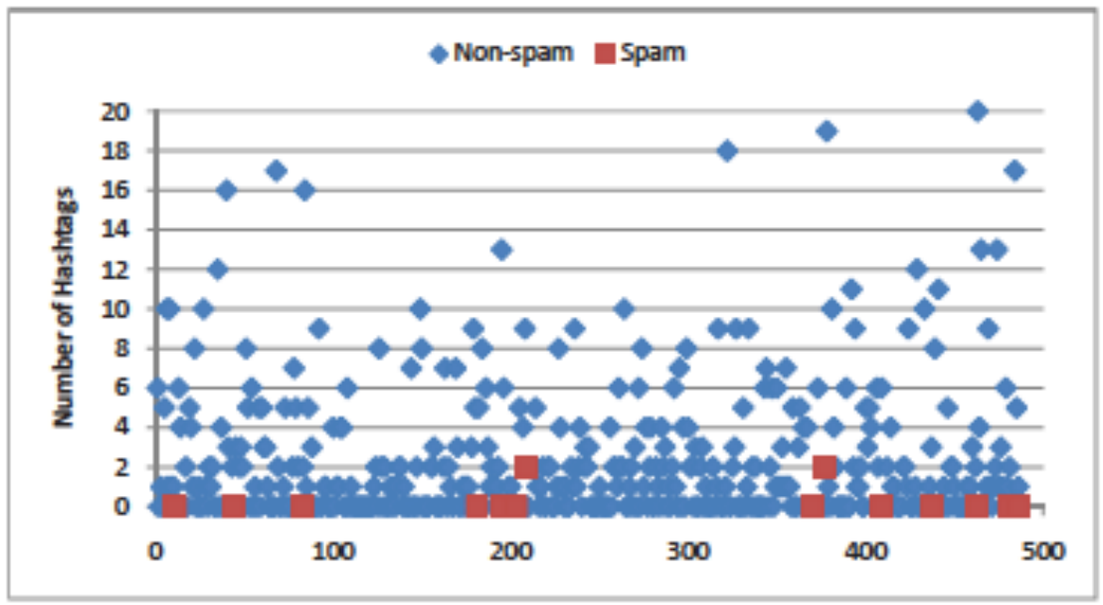
(a) The Number of Pairwise Duplications



(c) The Number of Links



(b) The Number of Mention and Replies



(d) The Number of Hashtags

Wang, A. H. (2010, July). Don't follow me: Spam detection in twitter. SECRYPT-10, 1-10. IEEE.

Hamming Distance

In information theory, the **Hamming distance** between two strings of equal length is the number of positions at which the corresponding symbols are different. In another way, it measures the minimum number of *substitutions* required to change one string into the other, or the minimum number of *errors* that could have transformed one string into the other.

A major application is in coding theory, more specifically to block codes, in which the equal-length strings are vectors over a finite field.

Other well known transformation, which is more restricted; only uses substitution and applies to strings of the same length

http://en.wikipedia.org/wiki/Hamming_distance

Issues with Edit Distance…

- Only good for small text-items; complexity gets big with longer strings
- Ultimately related to Kolomgorov complexity and other interesting things (compression)
- Generally, less commonly used type of similarity but very powerful in some cases (string distance, tweet differences, plagiarism)

Similarity

Structural Similarity

Intuition: Steps to Change…

- Featural approaches and various bag-of-words approaches throw away a lot

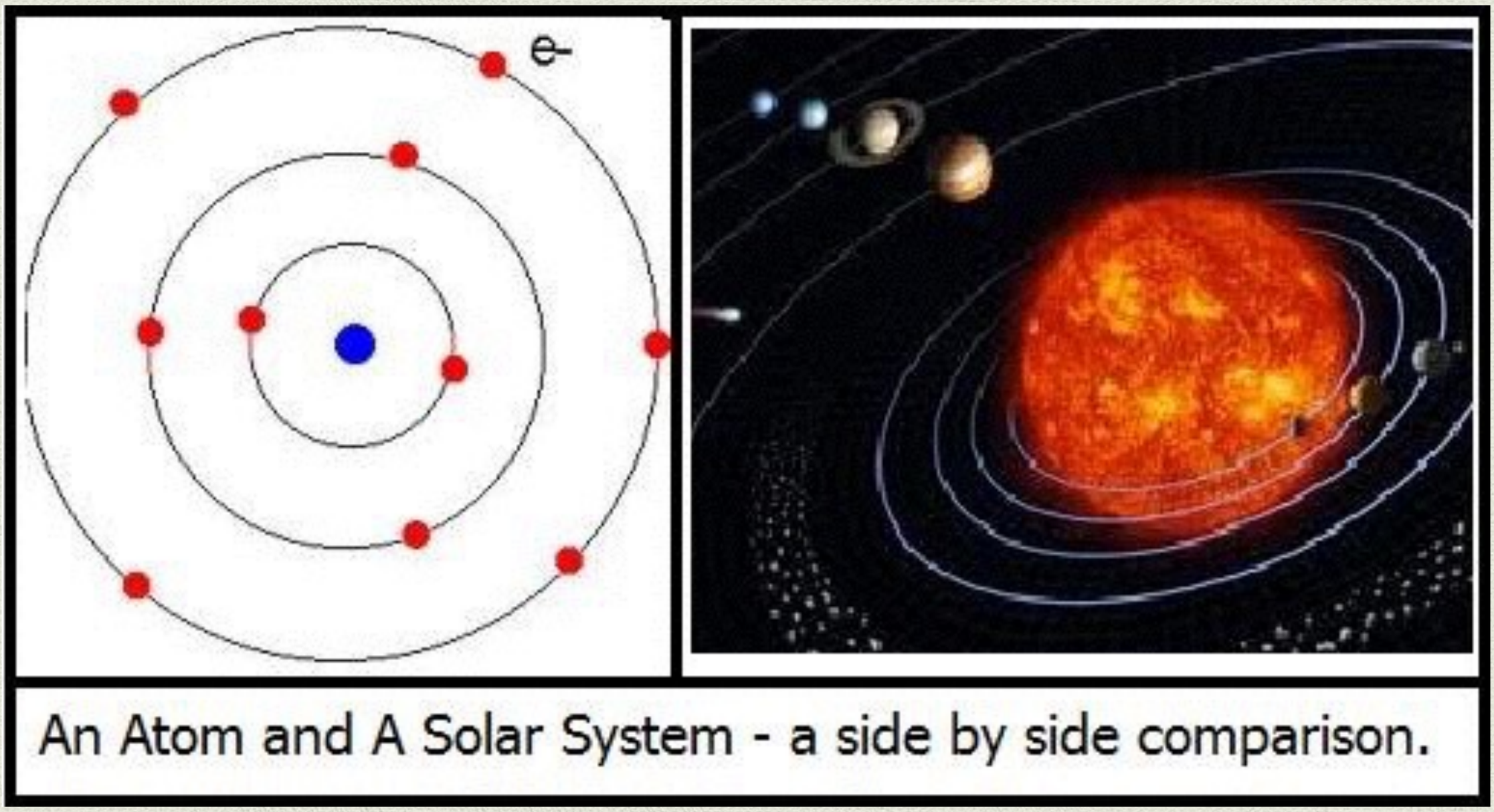
- In language, this might be the syntactic structure of the sentence:
  "man bites dog" and "dog bites man"

- In graphs, the structure of the graph may be important to shape-matching items

Intuition: Steps to Change…

- This type of structural similarity can be glossed as a special case of transformational
- Problem is, you often need to buy into more complex representations (predicate calculus)
- Caused it to fall out of favour in the face of massive success of VSM approaches

Rutherford Analogy



An Atom and A Solar System - a side by side comparison.

Analogy: Formula

STRUCTURE-MAPPING: INTERPRETATION RULES FOR ANALOGY

The analogy "A T is (like) a B" defines a mapping from B to T. T will be called the *target*, since it is the domain being explicated. B will be called the *base*, since it is the domain that serves as a source of knowledge. Suppose that the representation of the base domain B can be stated in terms of object nodes $b_1, b_2, \ldots, b_n$ and predicates such as A, R, R', and that the *target* domain has object nodes $t_1, t_2, \ldots, t_m$.[5] The analogy maps the object nodes of B onto the object nodes of T:

$$M: b_i \dashrightarrow t_i$$

These object correspondences are used to generate the candidate set of inferences in the target domain. Predicates from B are carried across[6] to T, using the node substitutions dictated by the object correspondences.

The mapping rules are

1. Discard attributes of objects:

$$A(b_i)] \dashrightarrow [A(t_i)$$

2. Try to preserve relations between objects:

$$R(b_i, b_j)] \dashrightarrow [R(t_i, t_j),$$

3. (The Systematicity Principle) To decide *which* relations are preserved, choose systems of relations:

$$R'(R_1(b_i, b_j), R_2(b_k, b_l)] \dashrightarrow$$

$$[R'(R_1(t_i, t_j), R_2(t_k, t_l)$$

Analogy: EG

Fig. 1. Two physical situations involving flow (adapted from [3]).

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The structure-mapping engine: Algorithm and examples. *Artificial intelligence, 41(1)*, 1-63.

Analogy: EG

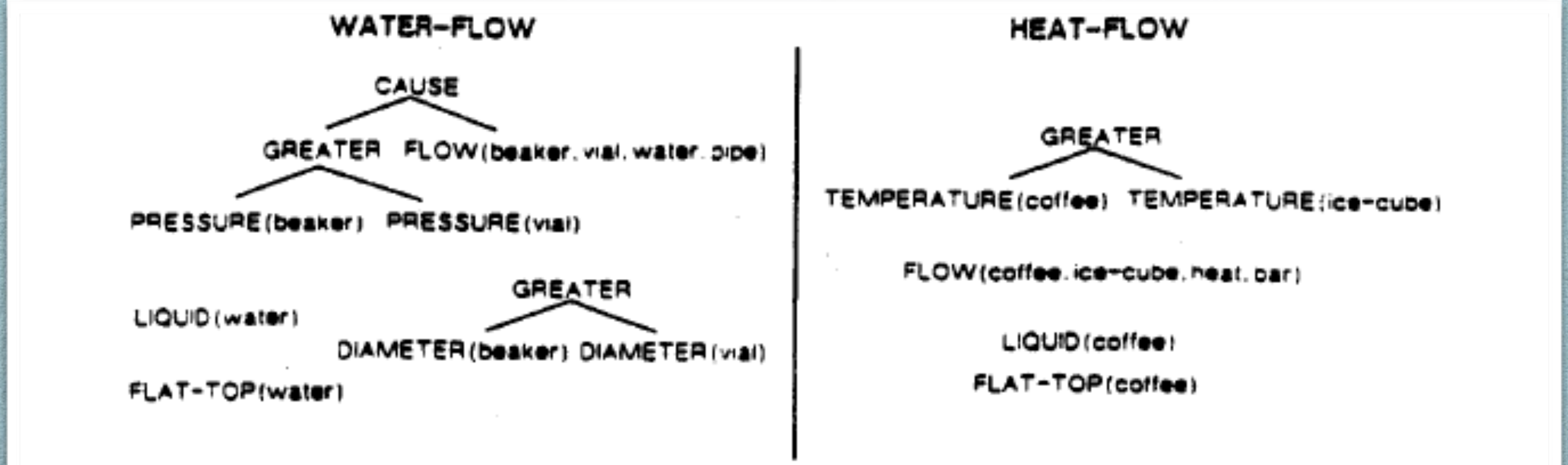


Fig. 2. Simplified water flow and heat flow descriptions.

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The structure-mapping engine: Algorithm and examples. *Artificial intelligence, 41(1)*, 1-63.

Analogy: EG

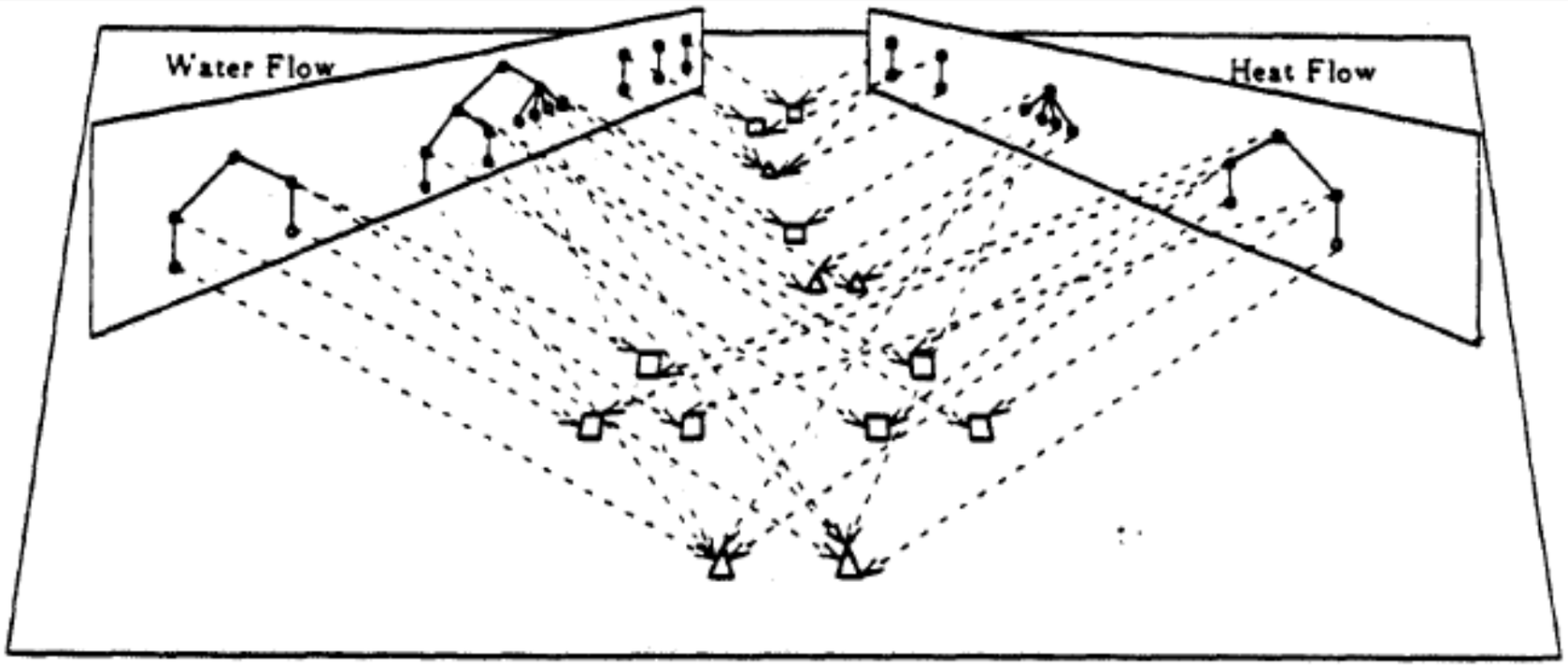
```
(defDescription simple-water-flow
  entities (water beaker vial pipe)
  expressions (((flow beaker vial water pipe) :name wflow)
               ((pressure beaker) :name pressure-beaker)
               ((pressure vial) :name pressure-vial)
               ((greater pressure-beaker pressure-vial) :name >pressure)
               ((greater (diameter beaker) (diameter vial))
                :name >diameter)
               ((cause >pressure wflow) :name cause-flow)
               (flat-top water)
               (liquid water)))
```

The description of heat flow depicted in Fig. 2 was given to SME as

```
(defDescription simple-heat-flow
  entities (coffee ice-cube bar heat)
  expressions (((flow coffee ice-cube heat bar) :name hflow)
               ((temperature coffee) :name temp-coffee)
               ((temperature ice-cube) :name temp-ice-cube)
               ((greater temp-coffee temp-ice-cube) :name >temperature)
               (flat-top coffee)
               (liquid coffee)))
```

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The structure-mapping engine: Algorithm and examples. *Artificial intelligence, 41(1)*, 1-63.

Analogy: EG

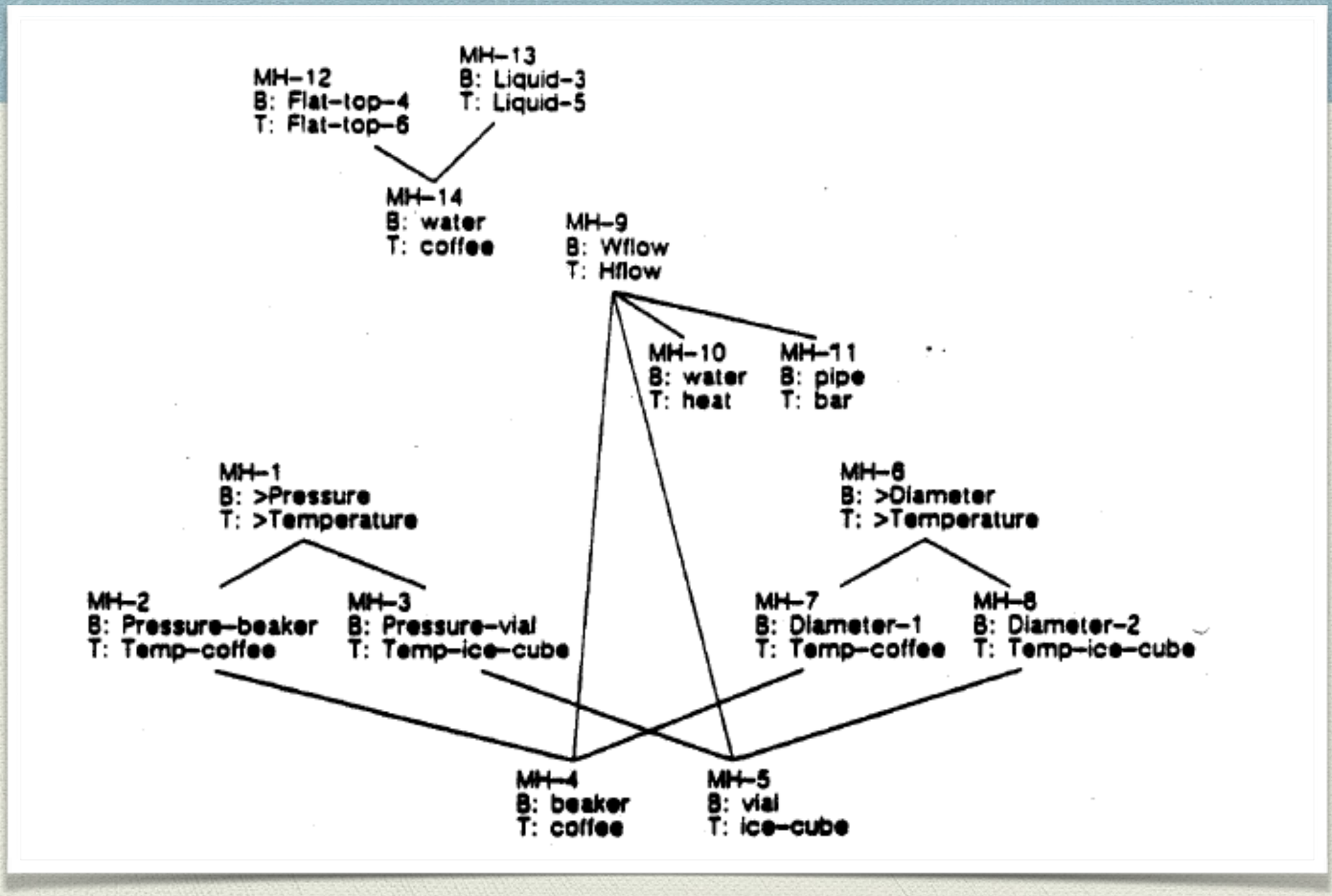


□ - MH between predicates
△ - MH between entities (Emap)

Fig. 3. Local match construction. The graphs corresponding to the water flow and heat flow descriptions of Fig. 2 are depicted on the left and right panels, respectively. The squares and triangles in the middle represent the match hypotheses created by the literal similarity rules for these dgroups. The dashed arrows indicate which base and target items are conjectured as matching by each match hypothesis. The squares represent match hypotheses involving expressions, while the triangles represent match hypotheses involving entities. Notice how sparse the match is. Expression matches are only created when relations are identical, and matches between functions and entities are only created to support expression matches. This "middle out" local match computation provides SME with much of its power.

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The structure-mapping engine: Algorithm and examples. *Artificial intelligence, 41(1)*, 1-63.

Analogy: EG



Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The structure-mapping engine: Algorithm and examples. *Artificial intelligence, 41(1)*, 1-63.

Issues

- Limited use in text analytics, everything thing is dominated by vector-space models
- Indeed, you can turn structures into features and deal with them directly in a VSM
- Also, it is argued that structural aspects can emerge from VSMs or Deep Learning

Conclusions

- Similarity is absolutely fundamental
- We will see it used repeatedly in later lectures
- Keep in mind different types: horses for courses
- Structural failure should not be overestimated
- For really good review see : Turney, P. D., & Pantel, P. (2010). From frequency to meaning: Vector space models of semantics. *JAIR, 37*, 141-188.