Finding Similar Sets: Shingling, MinHash, LSH

Jay Urbain, PhD

Credits:

J. Leskovec, A. Rajaraman, J. Ullman (Stanford University) Mining of Massive Datasets

http://infolab.stanford.edu/~ullman/mmds/ch3.pdf

MinHash

- MinHash (min-wise independent permutations locality sensitive hashing scheme) is a technique for quickly estimating how similar two sets are.
- Invented by Andrei Broder (1997).
- Initially used in the AltaVista search engine to detect duplicate web pages and eliminate them from search results.
- Has also been applied in large-scale clustering problems.

Applications of set similarity

- Many problems in information retrieval and data mining can be expressed as finding similar sets:
  - Duplicate docs
  - Finding docs with similar words, for classification by topic.
  - Netflix users with similar movie tastes
  - Movies with similar Netflix users
  - Entity resolutions

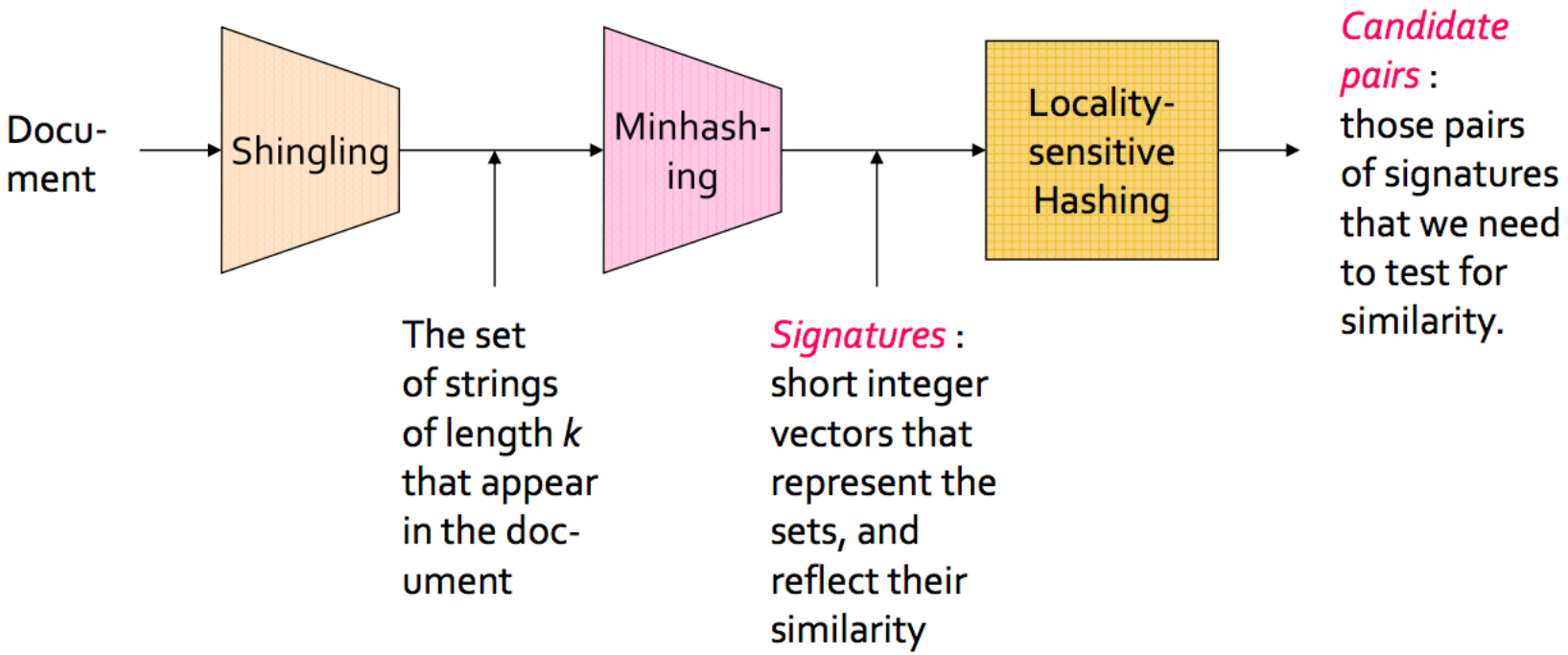
Similar Documents

- Given a body of documents, e.g., the Web, intelligence reports, medical records, with a lot of text in common:
  - Mirror sites
    - Application: remove redundant search results
  - Similar news stories
    - Cluster article by same story
  - Plagiarism, including large quotations

Techniques for similar documents

- Jaccard Similarity
- Shingling:
  - Convert documents, emails, etc., to text.
- MinHash:
  - Convert large documents to short, efficiently comparable signatures, while preserving similarity
- Locality sensitive hasing (LSH):
  - Focus on pairs of signatures likely to be similar

Overview


Docu-
ment
Shingling
Minhash-
ing
Locality-
sensitive
Hashing
Candidate
pairs :
those pairs
of signatures
that we need
to test for
similarity.
The set
of strings
of length k
that appear
in the doc-
ument
Signatures :
short integer
vectors that
represent the
sets, and
reflect their
similarity

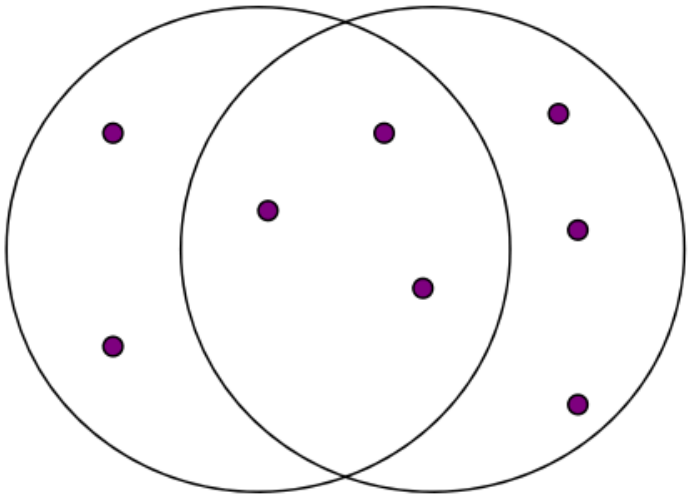
Jaccard similarity and minimum hash

- The Jaccard similarity coefficient is a commonly used indicator of the similarity between two sets.
- For sets A and B it is defined to be the ratio of the number of elements of their intersection and the number of elements of their union:

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

- This value is 0 when the two sets are disjoint, 1 when they are equal, and strictly between 0 and 1 otherwise.
- For the following two sets, J(A,B) = 3/5 = 0.6
  - Set A = new Set(["chair", "desk", "rug", "keyboard", "mouse"]);
  - Set B = new Set(["chair", "rug", "keyboard"]);

Example: Jaccard Similarity



3 in intersection.
8 in union.
Jaccard similarity
= 3/8

Documents as sets: Shingles

- We don't want to just break down a document into individual words, place them in a set and calculate the similarity.
- The following sets would be considered 100% similar:
  - Set a = new Set(["I", "went", "to", "work", "today"]);
  - Set b = new Set(["today", "I", "went", "to", "work"]);

- Instead we break a document down into what are known as shingles.
- Each shingle contains a set number of words (or characters).

Documents as sets

- "The quick brown fox jumps over the lazy dog", would be broken down into the following 5 word long shingles :
  - The quick brown fox jumps
  - quick brown fox jumps over
  - brown fox jumps over the
  - fox jumps over the lazy
  - jumps over the lazy dog

- Our document set would look like:
  - Set a = new Set(["The quick brown fox jumps", "quick brown fox jumps over", "brown fox jumps over the", "fox jumps over the lazy", "jumps over the lazy dog"]);

- Suppose our document D is the string *abcdabd*, and we pick k = 2. Then the set of 2-shingles for D is {ab, bc, cd, da, bd}.

- These sets of shingles can then be compared for similarity using the Jaccard Coefficient.

Choosing the Shingle Size

How large k should be depends on how long typical documents are and how large the set of typical characters is. The important thing to remember is:

- *k should be picked large enough that the probability of any given shingle appearing in any given document is low.*

- Thus, if our corpus of documents is emails, picking k = 5 should be fine.
  - To see why, suppose that only letters and a general white-space character appear in emails (although in practice, most of the printable ASCII characters can be expected to appear occasionally).
  - If so, then there would be 27^5 = 14,348,907 possible shingles.
  - Since the typical email is much smaller than 14 million characters long, we would expect k = 5 to work well, and indeed it does.

Optimizing the process

- We now have a way to compare two documents for similarity, but it is not an efficient process.
- Sets of shingles are large.
- To find similar documents to document ***A*** in a directory of 10000 documents, we need compare each pair individually. This will not scale well.
- To reduce computation, we can compare sets of randomly selected shingles from two documents.
- So for a document that is 10000 words long, we break it down into 9996 shingles, and randomly select say 200 of those to represent the document.
- If the second document is 20000 words long, we boil that down from 19996 shingles to another set of 200 randomly selected shingles.
- Now instead of matching 9996 shingles to 19996 other shingles, we are are comparing 200 shingles to another 200 shingles.
- We have reduced our workload at the expense of some accuracy, but this is still not good enough for most scenarios.

http://blog.cluster-text.com/tag/minhash/

Jaccard similarity and minimum hash

- Our goal is to replace large sets by much smaller representations called “signatures.”
- The important property we need for signatures is that we can compare the signatures of two sets and estimate the Jaccard similarity of the underlying sets from the signatures alone.
- Two sets are more similar when their Jaccard index is closer to 1.

Jaccard similarity and minimum hash

- The goal of MinHash is to estimate J(A,B) quickly, without explicitly computing the intersection and union.
- Let h be a hash function that maps the members of A and B to distinct integers.
  - For any set S, define $h_{min}(S)$ to be the minimal member of S with respect to h—that is, the member x of S with the minimum value of h(x).
  - If we apply h_{min} to both A and B, we will get the same value exactly when the element of the union $A \cup B$ with minimum hash value lies in the intersection $A \cap B$.
  - The probability of this being true is:

$$Pr[\ h_{min}(A) = h_{min}(B)\ ] = J(A,B)$$

Jaccard similarity and minimum hash

- The probability of this being true is:

$$\Pr[\, h_{min}(A) = h_{min}(B) \,] = J(A,B)$$

- The probability that $h_{min}(A) = h_{min}(B)$ is true is equal to the similarity J(A,B), assuming randomly chosen sets A and B.
- I.e., if r is the random variable that is one when $h_{min}(A) = h_{min}(B)$ and zero otherwise, then r is an unbiased estimator of J(A,B).
- r has too high a variance to be a useful estimator for the Jaccard similarity on its own—it is always zero or one.
- The idea of the MinHash scheme is to reduce this variance by averaging together several variables constructed in the same way.
  - Think Central Limit Theorem

Algorithm - Variant with many hash functions

- Use k different hash functions, where k is a fixed integer parameter, and represents each set S by the k values of $h_{min}(S)$ for these k functions.
- To estimate J(A,B), let y be the number of hash functions for which

  $h_{min}(A) = h_{min}(B)$, and use y/k as the estimate.
- This estimate is the average of k different 0-1 random variables, each of which is one when $h_{min}(A) = h_{min}(B)$ and zero otherwise, each of which is an unbiased estimator of J(A,B).

In practice:

- Use random row permutation, or
- use one hash algorithm and generate k different hash functions by XOR'ing the hash function value with a bounded random number.

Algorithm - Variant with single hash functions

- It may be computationally expensive to compute multiple hash functions.
- A related version of MinHash scheme avoids this penalty by using only a single hash function and uses it to select multiple values from each set rather than selecting only a single minimum value per hash function.
- Let h be a hash function, and let k be a fixed integer. If S is any set of k or more values in the domain of h, define h(k)(S) to be the subset of the k members of S that have the smallest values of h.
- This subset h(k)(S) is used as a signature for the set S, and the similarity of any two sets is estimated by comparing their signatures.

Time analysis

- The estimator |Y|/k can be computed in time O(k) from the two signatures of the given sets, in either variant of the scheme.
- Therefore, when k is constant, the time to compute the estimated similarity from the signatures is also constant.
- The signature of each set can be computed in linear time on the size of the set.
- So when many pairwise similarities need to be estimated this method can lead to a substantial savings in running time compared to doing a full comparison of the members of each set.

Special Property

There is a remarkable connection between minhashing and Jaccard similarity of the sets that are minhashed.

- *The probability that the minhash function for a random permutation of rows produces the same value for two sets equals the Jaccard similarity of those sets.*

Hash function by implicit random permutation of rows

Element	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

Element	S_1	S_2	S_3	S_4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

Although it is not physically possible to permute very large characteristic matrices, the minhash function h implicitly reorders the rows of the matrix (left) so it becomes the matrix (right). In the matrix, we can read off the values of h by scanning from the top until we come to a 1. Thus, we see that h(S2) = c, h(S3) = b, and h(S4) = a.

Computing minhash signatures

1. Compute $h_1(r), h_2(r), \ldots, h_n(r)$.
2. For each column c do the following:
   (a) If c has 0 in row r, do nothing.
   (b) However, if c has 1 in row r, then for each $i = 1, 2, \ldots, n$ set $\text{SIG}(i, c)$ to the smaller of the current value of $\text{SIG}(i, c)$ and $h_i(r)$.

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Figure 3.4: Hash functions computed for the matrix of Fig. 3.2

Computing minhash signatures

Element	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Computing signature matrix:

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

0

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

1

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

2

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

3

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

4

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

- We can estimate the Jaccard similarities of the underlying sets from this signature matrix.
- Notice that columns 1 and 4 are identical, so we guess that SIM(S1, S4) = 1.0.
- From the matrix, we see that the true Jaccard similarity of S1 and S4 is 2/3.

Computing and saving the random shingle selections - Minhash

- We could save the 200 randomly selected shingles, but storing variable length strings is not efficient. It complicates database design, and comparing strings is slow.

- The clever part about MinHash is that it allows us to save integers instead of strings, and also takes the hassle out of randomly selecting shingles. It works like this.

1. Break down the document as a set of shingles.
2. Calculate the hash value for every shingle.
3. Store the minimum hash value found in step 2.
4. Repeat steps 2 and 3 with different hash algorithms 199 more times to get a total of 200 min hash values.

Computing and saving the random shingle selections - Minhash

- At this point instead of 200 randomly selected shingles, we have 200 integer hash values.
- This is effectively a random selection of shingles, because a good hash algorithm is supposed to generate a number that is evenly distributed.
- This kind of distribution of hash codes is what hashing is all about, so selecting the smallest hash is, for all intents and purposes, a random selection of a shingle.

Locality sensitive hashing (LSH)

- We still need to compare every document to every other document. Can this be optimized?
- This is where the second neat trick of MinHash comes in.
- Locality sensitive hashing (LSH) involves generating a hash code such that similar items will tend to get similar hash codes.
- LSH allows you to pre-compute a hash code that is then quickly and easily compared to another pre-computed LSH hash code to determine if two objects should be compared in more detail or quickly discarded.

Locality sensitive hashing (LSH)

- The idea is to generalize a complex object (like our collection of minhash codes) into something that is quickly and easily compared with other complex objects, to determine with we should to a full compare.

- Take the collection of minhash values, and categorize them into bands and row.

- For simplicity, assume each document has 20 minhash values.

- We will break them down into 5 bands with 4 rows each.

- Also, for simplicity we'll assume that the hash function (and any number XORed with it) results in a number between 0 and 9.

Comparing documents fast - LSH

- So conceptually the minhash values from 5 documents in the first of the 5 bands and its rows looks like this.

		MinHash Algorithm One	MinHash Algorithm Two	MinHash Algorithm Three	MinHash Algorithm Four
Band One	Document One	1	3	6	0
	Document Two	2	3	1	0
	Document Three	1	3	6	0
	Document Four	2	1	3	1

Comparing documents fast - LSH

- What we are looking for are rows within a band that are the same between documents.
- Document One and Document Three in this case both have rows with values 1, 3, 6, 0. This tells us that these two documents should be compared for their similarity with MinHash.
- Any documents that share rows in any bands should be compared for their similarity.
- Useful when you have a document, and you want to know which other documents to compare to it for similarity.

```
SELECT DocName
FROM Documents
WHERE (
BandOneMinHashAlgorithmOne = 1 AND
BandOneMinHashAlgorithmTwo = 3 AND
BandOneMinHashAlgorithmThree = 6 AND
BandOneMinHashAlgorithmFour = 0
) OR
(
BandTwoMinHashAlgorithmOne = # AND
BandTwoMinHashAlgorithmTwo = # AND
BandTwoMinHashAlgorithmThree = # AND
BandTwoMinHashAlgorithmFour = #
) OR (
BandThreeMinHashAlgorithmOne = # AND
BandThreeMinHashAlgorithmTwo = # AND
BandThreeMinHashAlgorithmThree = # AND
BandThreeMinHashAlgorithmFour = #
) OR (
BandFourMinHashAlgorithmOne = # AND
BandFourMinHashAlgorithmTwo = # AND
BandFourMinHashAlgorithmThree = # AND
BandFourMinHashAlgorithmFour = #
) OR (
BandFiveMinHashAlgorithmOne = # AND
BandFiveMinHashAlgorithmTwo = # AND
BandFiveMinHashAlgorithmThree = # AND
BandFiveMinHashAlgorithmFour = #
)
```

- Assume that the minhash values are stored in SQL.
  To get out candidates for comparison you would write:

Implementation of Minhashing (3)

```
for each row r do begin
    for each hash function h_i do
        compute h_i(r);
    for each column c
        if c has 1 in row r
            for each hash function h_i do
                if h_i(r) is smaller than M(i, c) then
                    M(i, c) := h_i(r);
end;
```

Example

Row	C1	C2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

$h(x) = x \bmod 5$
$g(x) = (2x+1) \bmod 5$

	Sig1	Sig2
$h(1) = 1$	1	∞
$g(1) = 3$	3	∞
$h(2) = 2$	1	2
$g(2) = 0$	3	0
$h(3) = 3$	1	2
$g(3) = 2$	2	0
$h(4) = 4$	1	2
$g(4) = 4$	2	0
$h(5) = 0$	1	0
$g(5) = 1$	2	0