Com S 435/535: Notes V

1 Similar Items

We are given a collection of objects $\{O_1, O_2, \cdots O_N\}$. Our goal is to preprocess the collection and answer following questions

- Given two objects O_i and O_j estimate "similarity" between them.
- Given O_i, find all objects that are "similar" to O_i.
- From groups of objects so that objects within a group are "similar" to each other,

Similar Items problem is useful in many applications such as image search, web search, text processing and recommendation systems. To solve this problem, one first needs to define a notion of "similarity" among objects. A general approach involves three steps.

1. Extract features from objects and represent each object as a high-dimensional vector and define similarity.
2. Dimensionality Reduction. Reduce the dimension of the vectors
3. Apply Locality Sensitive Hashing to group similar objects.

We will illustrate this "documents" as our "objects".

2 Documents as Vectors

Let $D = \{D_1, D_2, \cdots, D_N\}$ be a collection of documents. We would like to answer the following questions:

- Given two documents D_a and D_b, how "similar" are they?
- Given a document D_a identify all the documents that are similar to D_a.
- Identify groups of documents so that documents with a group are similar to each other.

Before we attempt to define a notion of similarity, we do some preprocessing and extract relevant "features" of documents. We use the extracted features of documents to define similarity. An obvious choice for feature is "word". It is reasonable to assume that if two documents are highly similar to each other, they then exist a large set of words that appear in both documents. However, some words such as “the”, “a”, and “an” appear in every document and they do not contribute much to the similarity of documents. Such words are called *stop words*. Thus the stop words are not features of any document. Thus, we will first remove all stop words from every document in our document collection. Then, we can convert all characters to lower case (or upper case) as similarity of documents should not be case-sensitive. If needed (depending on application), we can do further preprocessing of documents. This includes: removing punctuation symbols, replacing words with synonyms, converting plural nouns into singular nouns (for example: “problems” to “problem”).

Next we define the notion of “term” of a document. We consider two choices for term. In the first choice, each word of the document (after pre processing is considered a term). For example if our document contains

Djokovic defeated Murray in Australian open final

then the set of terms is $\{djokovic, murray, australian, open, final\}$. Two documents D_a and D_b can be considered similar if the two documents have a large fraction of terms in common. Consider the following two documents

In Australian open final, Murray was defeated by Djokovic

Murray defeated Djokovic in Australian open final

If our notion of term is “word”, then the both these documents have the same set of terms. Thus we may conclude that the two documents are identical. However, even though these two documents contain very similar (???) information, they are not identical. The notion of *shingle* enables us to capture this. A *k-shingle* is a sequence of k-consecutive characters. For example if our document contains “James Bond”, then the all 4-shingles of the document are: “jame”, “ames”, “mes ”, “es b”, “s bo”, “ bon”, “bond”. In forming the shingles, we may chose to include white space as single character. Now the terms of a document are all k-shingles. A typical value of k is between 5 and 8. We use 5 for smaller documents (email, short articles) and 8 for larger documents (research papers).

To summarize, we can choose term to be either a word or a k-shingle. Let $D = \{D_1, D_2, \cdots, D_N\}$ be a collection of documents and let $T = \{t_1, t_2, \cdots, t_M\}$ be the set of all terms (words/shingles) that appear in the document collection. Let Tf_{ij} be the frequency of t_i in document D_j. For a given document $d \in D$, we define the term-frequency vector as

$$T_d = \langle Tf_1, Tf_2, \cdots Tf_m \rangle.$$

We can consider binary variant of the term-frequency vector. A binary variant is obtained by defining Tf_{ij} as 1 is t_i appears in D_j; otherwise Tf_{ij} equals zero. With this, the term-frequency vector of a document is a binary vector. Note that each vector that represents a document is a M-dimensional vector. If we collect all vectors that represent each document, then we obtain a $M \times N$ matrix, where columns are indexed by documents and rows are indexed by terms. Typically, this matrix is very sparse; there are many zeros as the number of terms that appear in a single document is much smaller compared to M. Another characteristic of this matrix is that all the vectors are *high-dimensional* vectors as typical value M may lie between 10000 and 40000.

3 Similarity/Distance Measures

We will now define a few similarity/distance measures of documents. Let D_a and D_b two documents and let and $T_a = \{a_1, \cdots, a_M\}$ and $T_b = \{b_1, \cdots, b_M\}$ be the two vectors that represent the documents.

Let

$$L_2(T_a) = \sqrt{a_1^2 + a_2^2 + \cdots a_m^2},$$

and

$$T_a \cdot T_b = a_1 b_1 + a_2 b_2 + \cdots + a_m b_m.$$

$L_2(T_a)$ is called the length of the vector T_a (or L_2 norm), and $T_a \cdot T_b$ is called *dot product of* T_a *and* T_b

Euclidean Distance. The Euclidean distance between the documents D_a and D_b is

$$L_2(T_a - T_b) \quad = \quad \sqrt{(a_1 - b_1)^2 + \cdots + (a_M - b_M)^2}$$

Thus Euclidean distance is the distance between the vectors T_a and T_b.

Cosine Distance The cosine distance between D_a and D_b measures the angle between two vectors T_a and T_b which equals cosine inverse of

$$\frac{T_a \cdot T_b}{L_2(T_a) \times L_2(T_b)}$$

The *cosine similarity* between D_a and D_b is simply

$$\frac{T_a \cdot T_b}{L_2(T_a) \times L_2(T_b)}$$

Note that if two documents are identical, the T_a and T_b are the same thus the angle between T_a and T_b is zero, thus the cosine similarity is 1.

Jaccard Distance The Jaccard Distance between D_a and D_b is

$$1 - \frac{T_a \cdot T_b}{[L_2(T_a)]^2 + [L_2(T_b)]^2 - T_a \cdot T_b},$$

and the Jaccard Similarity between D_a and D_b is

$$Jac(D_a, D_b) = \frac{T_a \cdot T_b}{[L_2(T_a)]^2 + [L_2(T_b)]^2 - T_a \cdot T_b}.$$

Jaccard Similarity as Set Similarity . Let D_a and D_b be two documents and S_a be the set of terms that appear in D_a and let S_b be the set of terms appearing in D_a. Let T_a and T_b be the *binary* term-frequency vectors representing D_a and D_b. It is easy to see that

$$Jac(D_a, D_b) = \frac{|S_a \cap S_b|}{|S_a \cup S_b|}.$$

In the rest of this notes, we use binary term-frequency vectors and take Jaccard similarity measure as the similarity measure.

4 Estimating Jaccard Similarity

Let D_a and D_b be two documents. Even though terms of a document are strings, it is useful to identify them as integers. Thus, for the sake of simplicity, let $\{1, 2, \cdots, m\}$ be the set of terms that appear in the two documents. Let S_a be the set of terms appearing document D_a and let S_b be the set of terms that appear in D_b. Given two documents D_a and D_b, we can compute $Jac(D_a, D_b)$ by computing the sizes of $S_a \cup S_b$ and $S_a \cap S_b$. Observe that if the sets S_a and S_b are stored as sorted lists, then this computation can be done in time $O(m)$ (otherwise the time taken is $O(m^2)$). We now present an entirely different method to estimate the similarity. For this, we need the notion of *random permutation.*

Random Permutations. Let $S = \{1, \cdots, m\}$. A *permutation* Π *on* S is a one-one, onto function from S to S. We can define the process of *picking a random permutation on* S as follows:

- Uniformly at random pick $n \in \{1, \cdots, m\}$ and set $\Pi(1)$ equals to n.
- For $i = 2$ to m DO
  - Uniformly at random pick $n \in \{1 \cdots, m\} - \{\Pi(1), \Pi(2), \cdots \Pi(i-1)\}$.
  - Set $\Pi(i)$ to n.

Claim 1. *Let $S = \{1, \cdots, m\}$ and let Π be a randomly chosen permutation on S. For every $i \in \{1, \cdots, m\}$ and for every $j \in \{1, \cdots, m\}$*

$$\Pr[\Pi(i) = j] = \frac{1}{m}.$$

Proof. Note that the event $\Pi(i) = j$ happens, when none of $1, \cdots i-1$ are mapped to j and i is mapped to j. Note that the probability that $\Pi(1) \neq j$ is $m - 1/m$; the probability that both $\Pi(1)$ and $\Pi(2)$ are not equal to j equals $(m - 1/m) \times (m - 2/m - 1)$. Thus

$$\begin{aligned}
\Pr[\Pi[i] = j] &= \Pr[\Pi(1) \neq j \cap \Pi(2) \neq j \cap \Pi(i-1) \neq j \cap \Pi(i) = j] \\
&= \frac{m-1}{m} \times \frac{m-2}{m-1} \times \cdots \times \frac{m-i+1}{m-i+2} \times \frac{1}{m-i+1} \\
&= \frac{1}{m}
\end{aligned}$$

□

We will use random permutations to arrive at a faster algorithm to estimate Jaccard similarity.

5 MinHash

For a given permutation Π, let $\min[\Pi(D_a)]$ be the smallest value of

$$\{\Pi(x) \mid x \in S_a\}.$$

We now prove a very interesting relationship among $\min[\Pi(D_a)]$, $\min[\Pi(D_b)]$, and $Jac(D_a, D_b)$. Recall that $S = \{1, 2, \cdots, m\}$ be the set of all terms that appear $D_a \cup D_b$.

Claim 2. *Randomly pick a permutation* Π*,*

$$\Pr[\min[\Pi(D_a)] = \min[\Pi(D_b)]] = Jac(D_a, D_b).$$

Proof. The claim follows by following two observations.

Observation 1. For every permutation Π ate least one of $\min[\Pi(D_a)]$ or $\min[\Pi(D_b)]$ equals 1.

Since Π is a permutation, there exist $x \in \{1, \cdots, m\}$, such that $\Pi(x) = 1$. Since $D_a \cup D_b = \{1, \cdots, m\}$, $x \in D_a \cup D_b$. Thus x must belong to at least one of D_a or D_b.

Observatin 2. Let $x \in D_a$ such that $\min[\Pi(D_a)] = \Pi(x)$, and let $y \in D_b$ such that $\min[\Pi(D_b)] = \Pi(y)$. If $\min[\Pi(D_a)] = \min[\Pi(D_b)]$, then $x = y$, and thus x belongs to $D_a \cap D_b$.

If $\min[\Pi(D_a)] = \min[\Pi(D_b)]$, then $\Pi(x) = \Pi(y)$. Since Π is one-one, it must be the case that $x = y$, and thus x appears in both D_a and D_b.

Thus,

$$\begin{aligned}
\Pr[\min[\Pi(D_a)] = \min[\Pi(D_b)]] &= \Pr[\min[\Pi(D_a)] = \min[\Pi(D_b)] = 1] \\
&= \Pr[\exists x \in D_a \cap D_b,\ \Pi(x) = 1] \\
&= \frac{|D_a \cap D_b|}{m} \text{ (by Claim 1)} \\
&= \frac{|D_a \cap D_b|}{|D_a \cup D_b|} \\
&= Jac(D_a, D_b)
\end{aligned}$$

□

This suggests the following algorithm A to estimating Jaccard similarity between two documents D_a and D_b.

1. Input: D_a, D_b. Let S be the set of terms in $D_a \cup D_b$.
2. Unifromly at random pick k permutations $\Pi_1 \cdots, \Pi_k$.
3. Let $MH_a = \langle \min[\Pi_1(D_a)], \min[\Pi_2(D_a)], \cdots, \min[\Pi_k(D_a)] \rangle$
4. Let $MH_b = \langle \min[\Pi_1(D_b)], \min[\Pi_2(D_b)], \cdots, \min[\Pi_k(D_b)] \rangle$
5. Let ℓ be the number of components where MH_a and MH_b match, i.e.

$$\ell = |\{i \mid \min[\Pi_i(D_a)] = \min[\Pi_i(D_b)], 1 \le i \le k\}|$$

6. Output ℓ/k.

If we pick k to be around 400, then the value output by the above algorithm is (with high probability) within 0.05 of actual Jaccard similarity of D_a and D_b. More formally, we can prove the following.

Claim 3. *Let* $k = O(\frac{1}{\epsilon^2}\log(1/\delta))$.

$$\Pr[Jac(D_a, D_b) - \epsilon \le \textit{Output of } A \le Jac(D_a, D_b) + \epsilon] \ge 1 - \delta.$$

Thus, the following two vectors

$$MH_a = \langle \min[\Pi_1(D_a)], \min[\Pi_2(D_a)], \cdots, \min[\Pi_k(D_a)] \rangle,$$

and

$$MH_b = \langle \min[\Pi_1(D_b)], \min[\Pi_2(D_b)], \cdots, \min[\Pi_k(D_b)] \rangle.$$

contain sufficient information to estimate Jaccard similarity of D_a and D_b.

Recall that, we can compute $Jac(D_a, D_b)$ be using T_a and T_b, where T_a and T_b are M-dimensional vectors. Compared to that MH_a and MH_b are k-dimensional vectors. While typical value of M is more than 10000, the value of k is around 400 to 800. Thus we have "compressed" the entire document D_a to a bunch of k-numbers, and these numbers (along with k permutations) are sufficient to estimate the similarity of D_a with similarity of other documents. The vector MH_a is a *fingerprint/sketch/signature* of the document D_a. Since the technique used to obtain this sketch/signature is the minimum value of (one-one) hash functions, this is called *minhash sketch/signature*.

What is the memory needed to represent a minhash sketch of a document? Since minhash sketch is a list of (around) 400 integers, and each integer can be represented using four bytes, the total size of the sketch is $1KB$. Note that size of the sketch is independent of the size of the document. So, we can store sketches of a Million documents using around $1GB$ memory

6 MinHash Matrix

Let $D = \{D_1, \cdots D_N\}$ be a collection of documents and $T = \{t_1, \cdots t_M\}$ be the set of all terms appearing in the collection D. Again for convince, view the terms as set of integers: $\{1, \cdots, M\}$. The *term-document* Matrix is the following $M \times N$ matrix, where b_{ij} equals 1 if the term t_i appears in document D_j; otherwise b_{ij} equals 0.

$$\begin{array}{ccccccc}
 & D_1 & \cdot & \cdot & \cdot & \cdot & D_N \\
t_1 & b_{11} & \cdot & \cdot & \cdot & \cdot & b_{1N} \\
t_2 & b_{21} & \cdot & \cdot & \cdot & \cdot & b_{2N} \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
t_M & b_{M1} & \cdot & \cdot & \cdot & \cdot & b_{MN}
\end{array}$$

The MinHah matrix of the above collection is a $k \times N$ matrix obtained by randomly picking k permutations $\Pi_1, \cdots \Pi_k$ (each permutation is from $\{1, \cdots m\}$ to $\{1, \cdots m\}$.

$$\begin{array}{cccccccc}
 & D_1 & \cdot & \cdot & \cdot & \cdot & D_N \\
\Pi_1 & \min[\Pi_1(D_1)] & \cdot & \cdot & \cdot & \cdot & \min[\Pi_1(D_k)] \\
\Pi_2 & \min[\Pi_2(D_1)] & \cdot & \cdot & \cdot & \cdot & \min[\Pi_2(D_k)] \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\Pi_k & \min[\Pi_k(D_1)] & \cdot & \cdot & \cdot & \cdot & \min[\Pi_k(D_N)]
\end{array}$$

In general the ijth entry of the MinHash matrix is $\min[\Pi_i(D_j)]$.

The following is a straight forward algorithm to compute the MinHash matrix of a document collection.

- Input: Document collection $D_1, D_2 \cdots D_N$.
- Compute terms of the collection $\{t_1, \cdots, t_M\}$
- Uniquely identify each term with a number from $\{1, \cdots, M\}$.
- Uniformly at random pick k permutations $\Pi_1, \cdots, \Pi_k$.
- For $i = 1$ to N
  - Compute MinHash Signature of D_i as

$$\begin{array}{c} \min[\Pi_1(D_i)] \\ \min[\Pi_2(D_i)] \\ \cdot \\ \cdots \\ \min[\Pi_k(D_i)] \end{array}$$

  - Set this to be $i$the column of the MinHash matrix M
- Output M.

What is the time taken by the above algorithm? Let us look at time compute each column of M. Column i is computed by computing each of $\min[\Pi_1(D_i)], \cdots, \min[\Pi_k(D_i)]$. Given ℓ, we can compute $\min[\Pi_\ell(D_i)]$ by computing Π_ℓ on every term of the document. Assuming that the time taken to compute the permutation is unit time, computation of $\min[\Pi_\ell(D_i)]$ takes $O($ Number of terms in$D_i)$ time. Thus the total time taken to compute the MinHash Matrix is

$$k \times (\text{Number of terms in } D_1 + \text{ Number of terms in } D_2 + \cdots + \text{ Number of terms in } D_N)$$