5 Min Hashing

Last time we saw how to convert documents into sets. Then we discussed how to compare sets, specifically using the Jaccard similarity. Specifically, for two sets $A = \{0, 1, 2, 5, 6\}$ and $B = \{0, 2, 3, 5, 7, 9\}$. The *Jaccard similarity* is defined

$$\begin{aligned}\mathsf{JS}(A, B) &= \frac{|A \cap B|}{|A \cup B|} \\ &= \frac{|\{0, 2, 5\}|}{|\{0, 1, 2, 3, 5, 6, 7, 9\}|} = \frac{3}{8} = 0.375.\end{aligned}$$

Although this gives us a single numeric score to compare similarity (or distance) it is not easy to compute, and will be especially cumbersome if the sets are quite large.

This leads us to a technique called *min hashing* that uses a randomized algorithm to quickly estimate the Jaccard similarity. Furthermore, we can show how accurate it is through the Chernoff-Hoeffding bound.

To achieve these results we consider a new *abstract data type*, a matrix. This format is incredible useful conceptually, but often extremely wasteful in practice.

5.1 Matrix Representation

Here we see how to convert a series of sets (e.g. a set of sets) to be represented as a single matrix. Consider sets:

$$\begin{aligned}S_1 &= \{1, 2, 5\} \\ S_2 &= \{3\} \\ S_3 &= \{2, 3, 4, 6\} \\ S_4 &= \{1, 4, 6\}\end{aligned}$$

For instance $\mathsf{JS}(S_1, S_3) = |\{2\}|/|\{1, 2, 3, 4, 5, 6\}| = 1/6$.

We can represent these four sets as a single matrix

Element	S_1	S_2	S_3	S_4
1	1	0	0	1
2	1	0	1	0
3	0	1	1	0
4	0	0	1	1
5	1	0	0	0
6	0	0	1	1

represents matrix $M = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$.

That element in the ith row and the jth column determine if element i is in set S_j. It is 1 if the element is in the set, and 0 otherwise. This captures *exactly* the same data set the set representation, but may take much more space. If the matrix is *sparse*, meaning that most entries (e.g. $> 90\%$ or maybe $> 99\%$, or more conceptually, as the matrix becomes $r \times c$ the non-zero entries grows as roughly $r + c$, but the space grows as $r \cdot c$) then it wastes a lot of space. But still it is very useful *to think about*.

5.2 Hash Clustering

The first attempt, called *hash clustering*, will not require the matrix representation, but will bring us towards our final solution to quickly estimate the Jaccard distance.

Consider the set of elements is $\mathcal{E} = \{1, 2, 3, 4, 5, 6\}$ and there is also a set of possible clusters $\mathcal{C} = \{A, B, C\}$ that is smaller in size than the set of elements $|\mathcal{C}| < |\mathcal{E}|$. Then we consider a random hash function that maps each element $e \in \mathcal{E}$ to a consistent location in $\mathcal{C}$. For instance $h : \mathcal{E} \to \mathcal{C}$ is defined so

$$h : [1, 2, 3, 4, 5, 6] \to [A, B, B, C, A, A].$$

Now we can consider the new (and smaller matrix) for our example

Cluster	S_1	S_2	S_3	S_4
A	1	0	1	1
B	1	1	1	0
C	0	0	1	1

represents matrix $M_h = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$.

Lets see how the Jaccard similarity holds up:

$$\begin{aligned} \mathsf{JS}(S_1, S_2) &= 0 & \mathsf{JS}_{\mathsf{clu}}(S_1, S_2) &= 1/2 \\ \mathsf{JS}(S_1, S_3) &= 2/6 & \mathsf{JS}_{\mathsf{clu}}(S_1, S_3) &= 2/3 \\ \mathsf{JS}(S_1, S_4) &= 1/5 & \mathsf{JS}_{\mathsf{clu}}(S_1, S_4) &= 1/3 \\ \mathsf{JS}(S_2, S_3) &= 1/4 & \mathsf{JS}_{\mathsf{clu}}(S_2, S_3) &= 1/3 \\ \mathsf{JS}(S_3, S_4) &= 0 & \mathsf{JS}_{\mathsf{clu}}(S_3, S_4) &= 0 \\ \mathsf{JS}(S_3, S_4) &= 2/5 & \mathsf{JS}_{\mathsf{clu}}(S_3, S_4) &= 2/3. \end{aligned}$$

Similarity generally increase. If there is an intersection, there is still an intersection. But also new intersections are formed.

This may appear not useful for this applications because it is hard to understand the errors induced by the clustering. But we will see later how this can be useful to find the *very frequent elements* in the count min hash.

5.3 Min Hashing

The next approach, called *min hashing*, initially seems even simpler than the clustering approach. It will need to evolve through several steps to become a useful trick.

Step 1: Randomly permute the items (by permuting the rows of the matrix).

Element	S_1	S_2	S_3	S_4
2	1	0	1	0
5	1	0	0	0
6	0	0	1	1
1	1	0	0	1
4	0	0	1	1
3	0	1	1	0

Step 2: Record the first 1 in each column

$$\begin{aligned} m(S_1) &= 2 \\ m(S_2) &= 3 \\ m(S_3) &= 2 \\ m(S_4) &= 6 \end{aligned}$$

Step 3: Estimate the Jaccard similarity $\mathsf{JS}(S_i, S_j)$ as

$$\hat{\mathsf{JS}}(S_i, S_j) = \begin{cases} 1 & m(S_i) = m(S_j) \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 5.3.1. $\boldsymbol{Pr}[m(S_i) = m(S_j)] = \boldsymbol{E}[\hat{\boldsymbol{JS}}(S_i, S_j)] = \boldsymbol{JS}(S_i, S_j)$.

Proof. There are three types of rows.

(Tx) There are x rows with 1 in both column

(Ty) There are y rows with 1 in one column and 0 in the other

(Tz) There are z rows with 0 in both column

The total number of rows is $x + y + z$. The Jaccard similarity is precisely $\mathsf{JS}(S_i, S_j) = x/(x + y)$. (Note that usually $z \gg x, y$ (mostly empty) and we can ignore these.)

Let row r be the $\min\{m(S_i), m(S_j)\}$. It is either type (Tx) or (Ty), and it is (Tx) with probability exactly $x/(x + y)$, since the permutation is random. This is the only case that $m(S_i) = m(S_j)$, otherwise S_i or S_j has 1, but not both. □

Thus this approach only gives 0 or 1, but has the right expectation. To get a better estimate, we need to repeat this several (k) times. Consider k random permutations $\{m_1, m_2, \ldots, m_k\}$ and also k random variables $\{X_1, X_2, \ldots, X_k\}$ (and $\{Y_1, Y_2, \ldots, Y_k\}$) where

$$X_\ell = \begin{cases} 1 & \text{if } m_\ell(S_i) = m_\ell(S_j) \\ 0 & \text{otherwise.} \end{cases}$$

and $Y_\ell = (1/k)(X_\ell - \mathsf{JS}(S_i, S_j))$. Let $M = \sum_{\ell=1}^{k} Y_\ell$ and $A = \sum_{\ell=1}^{k} X_\ell$. Note that $-1 \leq X_\ell \leq 1$ and $\mathbf{E}[M] = 0$. We can now apply Theorem 3.1.2 with $\Delta_i = 1$ and $r = k = (2/\varepsilon^2)\ln(2/\delta)$ to say

$$\mathbf{Pr}[|\mathsf{JS}(S_i, S_j) - A| < \varepsilon] > 1 - \delta.$$

That is, the Jaccard similarity is within ε error with probability at least $1 - \delta$ if we repeat this $k = (2/\varepsilon^2)\ln(2/\delta)$ times.

5.3.1 Fast Min Hashing Algorithm

This is still too slow. We need to construct the full matrix, and we need to permute it k times. A faster way is the *min hash algorithm*.

Make one pass over the data. Let $N = |\mathcal{E}|$. Maintain k random hash functions $\{h_1, h_2, \ldots, h_k\}$ so $h_i : \mathcal{E} \to [N]$ at random. An initialize k counters at $\{c_1, c_2, \ldots, c_k\}$ so $c_i = \infty$.

Algorithm 5.3.1 Min Hash on set S

```
for i = 1 to N do
  if (S(i) = 1) then
    for j = 1 to k do
      if (h_j(i) < c_j) then
        c_j ← h_j(i)
```

On output $m_j(S) = c_j$. If there are n elements total in the set, the first **for** and **if** can be made to just iterate over these elements so the runtime is only nk. And the output space of a single set is only $k = (2/\varepsilon^2)\ln(2/\delta)$ which is independent of the size of the original set. The space for n sets is only nk.

Finally, we can now estimate $\mathsf{JS}(S_i, S_{i'})$ as

$$\mathsf{JS}_k(S_i, S_{i'}) = \frac{1}{k}\sum_{i=1}^{k} \mathbf{1}(m_j(S_i) = m_j(S_{i'}))$$

where $\mathbf{1}(\gamma) = 1$ if $\gamma = \text{TRUE}$ and 0 otherwise.