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 if full written out.

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 as 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*. There are also sparse matrix representations built into many languages such as Matlab which do not store all of the 0s, they just store the locations of the non-zeros.

5.2 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, using a map function m. That is, given a permutation, applied to a set S, the function $m(S)$ records the element from S which appears earliest in this permutation.

$$m(S_1) = 2$$
$$m(S_2) = 3$$
$$m(S_3) = 2$$
$$m(S_4) = 6$$

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

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 $\{X_1, X_2, \ldots, X_k\}$ where

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

Now we can estimate $\mathsf{JS}(S_i, S_j)$ as $\hat{\mathsf{JS}}_k(S_i, S_j) = \frac{1}{k}\sum_{\ell=1}^{k} X_\ell$, the average of the k simple random estimates.

So how large should we set k so that this gives us an accurate measure? Since it is a randomized algorithm, we will have an error tolerance $\varepsilon \in (0,1)$ (e.g. we want $|\mathsf{JS}(S_i,S_j) - \hat{\mathsf{JS}}_k(S_i,S_j)| \leq \varepsilon$), and a probability of failure δ (e.g. the probability we have more than ε error). We will now use Theorem 3.1.1 where $M = \sum_{\ell=1}^{k} X_\ell$ and hence $\mathbf{E}[M] = k \cdot \mathsf{JS}(S_i,S_j)$. We have $0 \leq X_i \leq 1$ so each $\Delta_i = 1$. Now we can write for some value α:

$$\begin{aligned}\mathbf{Pr}[|\hat{\mathsf{JS}}_k(S_i,S_j) - \mathsf{JS}(S_i,S_j)| \geq \alpha/k] &= \mathbf{Pr}[|k \cdot \hat{\mathsf{JS}}_k(S_i,S_j) - k \cdot \mathsf{JS}(S_i,S_j)| \geq \alpha] \\ &= \mathbf{Pr}[|M - \mathbf{E}[M]| \geq \alpha] \leq 2\exp\left(\frac{-2\alpha^2}{\sum_{i=1}^{k}\Delta_i^2}\right) = 2\exp(-2\alpha^2/k).\end{aligned}$$

Setting $\alpha = \varepsilon k$ and $k = (1/(2\varepsilon^2))\ln(2/\delta)$ we obtain

$$\mathbf{Pr}[|\hat{\mathsf{JS}}_k(S_i,S_j) - \mathsf{JS}(S_i,S_j)| \geq \varepsilon] \leq 2\exp(-2(\varepsilon^2 k^2)/k) = 2\exp(-2\varepsilon^2 \frac{1}{2\varepsilon^2}\ln(2/\delta)) = \delta.$$

Or in other words, if we set $k = (1/2\varepsilon^2)\ln(2/\delta)$, then the probability that our estimate $\hat{\mathsf{JS}}_k(S_i,S_j)$ is within ε of $\mathsf{JS}(S_i,S_j)$ is at least $1-\delta$.

Say for instance we want error *at most* $\varepsilon = 0.05$ and can tolerate a failure 1% of the time ($\delta = 0.01$), then we need $k = (1/(2 \cdot 0.05^2))\ln(2/0.01) = 200\ln(200) \approx 1060$. Note that the modeling error of converting a structure into a set may be more than $\varepsilon = 0.05$, so this should be an acceptable loss in accuracy.

5.2.1 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 $\{h_1, h_2, \ldots, h_k\}$ chosen from a hash family at random so $h_i : \mathcal{E} \to [n]$ (one can use a larger range $n' > n$ where $n' = 2^t$ is a power of two). An initialize k values at $\{v_1, v_2, \ldots, v_k\}$ so $v_i = \infty$.

Algorithm 5.2.1 Min Hash on set S

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

On output $m_j(S) = v_j$. The algorithm runs in $|S|k$ steps, for a set S of size $|S|$. Note this is independent of the size n of all possible elements $\mathcal{E}$. And the output space of a single set is only $k = (1/2\varepsilon^2)\ln(2/\delta)$ which is independent of the size of the original set. The space for N sets is only $O(Nk)$.

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. This only takes $O(k)$ time, again independent of n or $|S_i|$ and $|S_{i'}|$.