6 Jaccard Similarity and Min-Hashing

We will study how to define the distance between sets, specifically with the Jaccard distance. To illustrate and motivate this study, we will focus on using Jaccard distance to measure the distance between documents. While modern approaches for *understanding and responding* to documents typically feed them into a transformer-based LLM, variants of these approaches are still often used for retrevial tasks.

We start with some big questions. This lecture will only begin to answer them.

- Given two homework assignments (reports) how can a computer detect if one is likely to have been plagiarized from the other without *understanding* the content?
- In trying to index webpages, how does Google say a webpage is similar to a keyword. How does Google avoid listing duplicates or mirrors?
- How does a computer quickly understand emails, for either detecting spam or placing effective advertisers? (If an ad worked on one email, how can we determine which others are similar?)

The key to answering these questions will be convert the data (homeworks, webpages, emails) into an object in an *abstract space* that we know how to measure distance, and how to do it efficiently. Instead we will use a different abstract distance between (unordered) *sets*.

6.1 Sets and Distances

A *set* is a (unordered) collection of objects $\{a, b, c\}$. We use the notation as elements separated by commas inside curly brackets $\{$ and $\}$. They are unordered so $\{a, b\} = \{b, a\}$.

Although we are interested in a *distance*, we will actually focus on a dual notion of a *similarity*. A distance $d(A, B)$ has the properties:

- it is small if objects A and B are close,
- it is large if they are far,
- it is (usually) 0 if they are the same, and
- it has value in $[0, \infty]$.

On the other hand, a similarity $s(A, B)$ has the properties:

- it is large if the objects A and B are close,
- it is small if they are far,
- it is (usually) 1 if they are the same, and
- it is in the range $[0, 1]$.

Often we can convert between the two as $d(A, B) = 1 - s(A, B)$, however sometimes it is better to use $d(A, B) = \sqrt{s(A, A) + s(B, B) - 2s(A, B)}$. Both restrict the distance to be a bounded (non infinite) domain, that can be converted with a $\tan$ map if one desires.

6.1.1 Jaccard Similarity

Consider two sets $A = \{0, 1, 2, 5, 6\}$ and $B = \{0, 2, 3, 5, 7, 9\}$. How similar are A and B?

The *Jaccard similarity* is defined

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

More notation, given a set A, the *cardinality* of A denoted $|A|$ counts how many elements are in A. The *intersection* between two sets A and B is denoted $A \cap B$ and reveals all items which are in *both* sets. The *union* between two sets A and B is denoted $A \cup B$ and reveals all items which are in *either* set.

Confirm that JS satisfies the properties of a similarity.

Generalized set similarities. To fully generalize set similarities (at least those that are amenable to large-scale techniques) we introduce a third set operation. The *symmetric difference* between two sets A and B is denoted $A \triangle B = (A \cup B) \setminus (A \cap B)$. Note that $\setminus$ is called *set minus* and $A \setminus B$ is all of the elements in A, except those also in B. Thus the symmetric difference of A and B describes all elements in A or B, but not in both.

We now consider the follow class of similarities. We use $\overline{A \cup B} = [n] \setminus (A \cup B)$, where $[n]$ is a superset that all sets A and B we consider a subsets from.

$$S_{x,y,z,z'}(A, B) = \frac{x|A \cap B| + y|\overline{A \cup B}| + z|A \triangle B|}{x|A \cap B| + y|\overline{A \cup B}| + z'|A \triangle B|}.$$

Now we can define several concrete instances.

- *Jaccard Similarity* is defined
$$\mathsf{JS}(A, B) = S_{1,0,0,1}(A, B) = \frac{|A \cap B|}{|A \cap B| + |A \triangle B|} = \frac{|A \cap B|}{|A \cup B|}.$$
- *Hamming Similarity* is defined
$$\mathsf{Ham}(A, B) = S_{1,1,0,1}(A, B) = \frac{|A \cap B| + |\overline{A \cup B}|}{|A \cap B| + |\overline{A \cup B}| + |A \triangle B|} = 1 - \frac{|A \triangle B|}{|[n]|}.$$
- *Andberg Similarity* is defined
$$\mathsf{Andb}(A, B) = S_{1,0,0,2}(A, B) = \frac{|A \cap B|}{|A \cap B| + 2|A \triangle B|} = \frac{|A \cap B|}{|A \cup B| + |A \triangle B|}.$$
- *Rogers-Tanimoto Similarity* is defined
$$\mathsf{RT}(A, B) = S_{1,1,0,2}(A, B) = \frac{|A \cap B| + |\overline{A \cup B}|}{|A \cap B| + |\overline{A \cup B}| + 2|A \triangle B|} = \frac{|[n]| - |A \triangle B|}{|[n]| + |A \triangle B|}.$$
- *Sørensen-Dice Similarity* is defined
$$\mathsf{S\text{-}Dice}(A, B) = S_{2,0,0,1}(A, B) = \frac{2|A \cap B|}{2|A \cap B| + 1|A \triangle B|} = \frac{2|A \cap B|}{|A| + |B|}.$$

For JS, Ham, Andb, and RT, the distance $D(A, B) = 1 - S(A, B)$ is a *metric*, and these four are amenable to LSH. We will discuss these topics later. See Chierichetti and Kumar (*LSH-Preserving Functions and their Applications*, SODA 2012) for more details.

6.2 Documents to Sets

How do we apply this set machinery to documents?

Bag of words vs. k-Grams The first option is the *bag of words* model, where each document is treated as an unordered set of words.

A more general approach is to *shingle* the document (or create k-grams). This takes consecutive words and group them as a single object. A *k-gram* is a consecutive set of k words. So the set of all 1-grams is exactly the bag of words model. An alternative name to *k-gram* is a *k-shingle*; these mean the same thing.

D_1 : `I am Sam.`

D_2 : `Sam I am.`

D_3 : `I do not like green eggs and ham.`

D_4 : `I do not like them, Sam I am.`

The $(k = 1)$-grams of $D_1 \cup D_2 \cup D_3 \cup D_4$ are: $\{$`[I], [am], [Sam], [do], [not], [like], [green], [eggs], [and], [ham], [them]`$\}$.

The $(k = 2)$-grams of $D_1 \cup D_2 \cup D_3 \cup D_4$ [1] are: $\{$`[I am], [am Sam], [Sam Sam], [Sam I], [am I], [I do], [do not], [not like], [like green], [green eggs], [eggs and], [and ham], [like them], [them Sam]`$\}$.

The set of k-grams of a document with n words is at most $n - k$. The takes space $O(kn)$ to store them all. If k is small, this is not a high overhead. Furthermore, the space goes down as items are repeated.

Character level. We can also create k-grams at the character level. The $(k = 3)$-character grams of $D_1 \cup D_2$ are: $\{$`[iam], [ams], [msa], [sam], [ami], [mia]`$\}$.

The $(k = 4)$-character grams of $D_1 \cup D_2$ are: $\{$`[iams], [amsa], [msam], [sams], [sami], [amia], [miam]`$\}$.

Modeling choices.

- **White space?** Should we include spaces, and returns? Sometimes. `plane has touch down` versus `threw a touchdown`.
- **Capitalization?** `Sam` versus sam. Can help distinguish proper nouns.
- **Punctuation?** May be indication of education level, or dialects. For instance English is punctuated differently in US and India. Punctuation is used differently in new articles (very proper style), blogs (more informal), and twitter (what is punctuation?).
- **Concatenation?** Should we concatenate consecutive documents as above, or not let k-grams span separate documents?
- **Characters vs. Words?** Long enough grams with characters can simulate words, but will have more *false positives*. Can pick up other dialect patterns. But is less interpretable.

- **How large should k be?** General rule: probability of (almost all) k-grams is low, so a collision is meaningful.
  For word-k-grams: emails $k = 2$ or 3 (small documents), research articles $k = 3$ or 4 (large documents), news articles, blog posts (in between).
  In English there are 27 characters (26 letters + 1 whitespace). With $k = 5$ there are $27^5 \approx 14$ millions

[1]Note here we treat the union of documents as concatenation. e.g. $D_1 \cup D_2 =$ `I am Sam.  Sam I am.` This is not always typical for much larger documents.

possible k-grams. (Maybe in practice closer to 20^5 since some letters (e.g. `z,q,x` are rarely used.)

- **Count replicas?** Typically *bag of words* counts replicas, but *k-gramming* does not.
- **Stop words?** Words like {`a, you, for, the, to, and, that, it, is, ...`} are very common, and called *stop words*. Sometimes omit these (typically in bag of words). In shingling can be effective to say use $k = 3$ where the first word must be a stop word: `the pizza oven`.

There are many variations of these methods. Classic Natural Language Processing (NLP) studied these variations, but also focused on finding much richer representations of bodies of text. Identifying all nouns and verbs, and disambiguating words with multiple meanings `went to the retreat` versus `the troops had to retreat`.

6.3 Jaccard with k-Grams

So how do we put this together. Consider the $(k = 2)$-grams for each D_1, D_2, D_3, and D_4:

D_1 : `[I am], [am Sam]`

D_2 : `[Sam I], [I am]`

D_3 : `[I do], [do not], [not like], [like green], [green eggs],`
`[eggs and], [and ham]`

D_4 : `[I do], [do not], [not like], [like them], [them Sam], [Sam I], [I am]`

Now the Jaccard similarity is as follows:

$$\begin{aligned}
\mathsf{JS}(D_1, D_2) &= \quad 1/3 \quad &\approx 0.333\\
\mathsf{JS}(D_1, D_3) &= \quad 0 \quad &= 0.0\\
\mathsf{JS}(D_1, D_4) &= \quad 1/8 \quad &= 0.125\\
\mathsf{JS}(D_2, D_3) &= \quad 0 \quad &= 0.0\\
\mathsf{JS}(D_3, D_4) &= \quad 2/7 \quad &\approx 0.286\\
\mathsf{JS}(D_3, D_4) &= \quad 3/11 \quad &\approx 0.273
\end{aligned}$$

Next time we will see how to use this special abstract structure of sets to compute this distance (approximately) very efficiently and at extremely large scale.

6.4 Continuous Bag of Words

Sometimes it is useful to obtain statistics for individual words from text corpuses. This will allow us to compare words in ways we will discuss later in the course. The simplest way is called *continuous bag of words (CBOW)*. For any instance of a word in a corpus (for instance "like"), it collects in a set (with multipilicity) all other words used within a distance r of that word. For large corpuses $r = 5$ has been suggested. Then a word's representation is the combination of all of these sets.

Distance between words are then calculated commonly in two ways. The first is Jaccard distance between the sets.

The second it to represent the set of all possible words as elements of a vector: each coordinate corresponds with a distinct word. Then for the representation of an instance of a word in the CBOW model, we build a vector v_{word}, initially all 0s. Each other word in a neighborhood set has its coordinate in v_{word} set to 1 (or c if it occurs c times in the neighborhood set). Then the vector representation of that word from the corpus, is the average of all vectors for each instance of that word. Finally, we can use the *cosine distance* (discussed in a later lecture, or just Euclidean distance) to determine the distance between words.

Example Consider the text corpus:
`I am Sam Sam I am I do not like green eggs and ham I do not like them Sam I am`
Then using the example word "like" and the parameter $r = 2$, the representation of the first and second instances are:

$$v^1_{\text{like}} = (0,0,0,1,1,0,1,1,0,0,0)$$
$$v^2_{\text{like}} = (0,0,1,1,1,0,0,0,0,0,1)$$

where the each coordinate is associated with the following words

$$(\texttt{I}, \texttt{am}, \texttt{Sam}, \texttt{do}, \texttt{not}, \texttt{like}, \texttt{green}, \texttt{eggs}, \texttt{and}, \texttt{ham}, \texttt{them}).$$

In reality, the above vectors are much longer, with the many other words used in English.

The average of the two vectors is then:

$$(v^1_{\text{like}} + v^2_{\text{like}})/2 = v_{\text{like}} = (0,0,0.5,1,1,0,0.5,0.5,0,0,0.5)$$

Note that the other words `do` and `not` occur commonly with `like`.

6.5 Locality Sensitive Hashing

Now we face the challenge of if we have many many documents (say n documents) and we want to find the ones that are close. But we don't want to calculate $\binom{n}{2} \approx n^2/2$ distances to determine which ones are really similar.

In particular, consider we have $n =$ 1,000,000 items and we want to ask two questions:

(Q1): Which items are similar?

(Q2): Given a query item, which others are similar to the query?

For **(Q1)** we don't want to check all roughly n^2 distance (no matter how fast each computation is), and for **(Q2)** we don't want to check all n items. In both cases we somehow want to figure out which ones might be close and the check those.

As an alternative to decomposition-based nearest neighbor searching, or HNSW, we will develop a technique called *Locality Sensitive Hashing* (or LSH). Moreover, this approach works directly for when the abstract representation is sets.

6.5.1 Properties of Locality Sensitive Hashing

We start with the goal of constructing a *locality-preserving* hash function h with the following properties (think of a random grid over spatial data). Do not confuse this with a (random) hash function discussed in **L2**. The locality needs to be with respect to a distance function $\mathbf{d}(\cdot,\cdot)$. In particular, if h is $(\gamma, \phi, \alpha, \beta)$-*sensitive* with respect to $\mathbf{d}$ then it has the following properties:

- $\mathbf{Pr}[h(a) = h(b)] > \alpha$ if $\mathbf{d}(a,b) < \gamma$
- $\mathbf{Pr}[h(a) = h(b)] < \beta$ if $\mathbf{d}(a,b) > \phi$

For this to make sense we need $\alpha > \beta$ for $\gamma < \phi$. Ideally we want $\alpha - \beta$ to be large and $\phi - \gamma$ to be small. Then we can repeat this with more random hash functions to *amplify* the effect, according to a Chernoff-Hoeffding bound. This will effectively make $\alpha - \beta$ larger for any fixed $\phi - \gamma$ gap, and will work for all such $\phi - \gamma$ simultaneously.

6.6 Minhashing as LSH

Minhashing is an instance of LSH (and perhaps the first such realized instance). We will have t hash functions $\{h_1, h_2, \ldots, h_k\}$ where each $h_i : 2^\Omega \to [n]$ is chosen at random from a family.

Documents:	D_1	D_2	D_3	D_4	...	D_n
h_1	1	2	4	0	...	1
h_2	2	0	1	3	...	2
h_3	5	3	3	0	...	1
...	...	...	...	...	...	...
h_t	1	2	3	0	...	1

It will satisfy

$$\mathsf{JS}_t(D_1, D_2) = \mathbf{E}[(1/t) \cdot (\text{ \# rows } h_i(D_1) = h_i(D_2))]$$

and in general $\mathsf{JS}(a, b) = \mathbf{Pr}[h(a) = h(b)]$.

Then for some similarity threshold $1 - \mathsf{JS}(a, b) = \mathbf{d}(a, b) = \tau$ we can set $\tau = \gamma = \phi$ and the have $\alpha = 1 - \tau$ and $\beta = \tau$.

This scheme will work for any similarity such that $s(a, b) = \mathbf{Pr}[h(a) = h(b)]$.

6.6.1 Minhashing's Hash Fucntion

We next describe how we construct the hash functions h_j in MinHashing. We start with a *random hash function* $h'_j \in \mathcal{H}$ that maps from Ω (the domain of the sets) to a large space $[n']$ (for $n' > |\Omega|^2$). For each h'_j we initialize a value $v_j = \infty$ and make one pass over the set S to build the hash h_i.

Algorithm 6.6.1 Min Hash: $h_j(S)$

```
for $i \in S$ do
  if ($h'_j(i) < v_j$) then
    $v_j \leftarrow h'_j(i)$
return $h_j(S) = v_j$
```

It is better to implement so we run k hash functions over one set S simultaneously as follows to get a vector $v = (v_1, v_2, \ldots, v_k) \in \mathbb{R}^k$ as $v_j = h_j(S)$ for $\{h_1, h_2, \ldots, h_k\} \in H$.

Algorithm 6.6.2 Fast Min Hash: $v = H(S)$

```
for $i \in S$ do
  for $j = 1$ to $k$ do
    if ($h'_j(i) < v_j$) then
      $v_j \leftarrow h'_j(i)$
return $v = (v_1, v_2, \ldots, v_k)$.
```

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 needs only be $k = (1/2\varepsilon^2)\ln(2/\delta)$ (see discussion below) which is independent of the size of the original set. The space for N sets is only $O(Nk)$.

Finally, we can now estimate $\mathsf{JS}(S, S')$ for two sets S and S' as

$$\mathsf{JS}_k(S, S') = \frac{1}{k}\sum_{j=1}^{k} \mathbf{1}(h_j(S) = h_j(S'))$$

where $\mathbf{1}(\gamma) = 1$ if $\gamma = \textsc{True}$ and 0 otherwise. This only takes $O(k)$ time, again independent of n or $|S|$ and $|S'|$.

6.7 Slow Min Hashing and Analysis

We will now describe a permutation perspective on *min hashing* (this is not how you would implement it, but makes the analysis of why it works more clear).

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 $\textsf{JS}(S_i, S_j)$ as

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

Lemma 6.7.1. $\textbf{\textit{Pr}}[m(S_i) = m(S_j)] = \textbf{\textit{E}}[\hat{\textsf{JS}}(S_i, S_j)] = \textsf{JS}(S_i, S_j)$.

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 $\textsf{JS}(S_i, S_j) = x/(x + y)$. (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 $\textsf{JS}(S_i, S_j)$ as $\hat{\textsf{JS}}_k(S_i, S_j) = \frac{1}{k}\sum_{\ell=1}^{k} X_\ell$, 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 2.5.2 (L2) 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.

Top k. It is sometimes more efficient to use the top-k (for some small number $k > 1$) hash values for each hash function, than just the top one. For instance, see Cohen and Kaplan (*Summarizing Data using Bottom-k Sketches*, PODC 2007). This approach requires a bit more intricate analysis, as well as a bit more careful implementation.