8. Locality Sensitive Hashing
Motivation, I
Find similar items in high dimensions, quickly

Could be useful, for example, in nearest neighbor algorithm.. but in a large, high dimensional dataset this may be difficult!
Motivation, II
Hashing is good for checking existence, not nearest neighbors

what is the nearest neighbor of 6?
Motivation, III
Main idea: want hashing functions that map similar objects to nearby positions using *projections*.

[FIG1] Two examples showing projections of two close (circles) and two distant (squares) points onto the printed page.
Different types of hashing functions

Perfect hashing

- Provide 1-1 mapping of objects to bucket ids
- Any two different objects mapped to different buckets (no collisions)

Universal hashing

- A family of functions $\mathcal{F} = \{h : U \rightarrow [n]\}$ is called *universal* if $P[h(x) = h(y)] \leq \frac{1}{n}$ for all $x \neq y$
- i.e. probability of collision for different objects is at most $1/n$

Locality sensitive hashing (Ish)

- Collision probability for *similar* objects is high enough
- Collision probability for *dissimilar* objects is low
A family \( \mathcal{F} \) is called \((s, c \cdot s, p_1, p_2)\)-sensitive if for any two objects \( x \) and \( y \) we have:

- If \( s(x, y) \geq s \), then \( P[h(x) = h(y)] \geq p_1 \)
- If \( s(x, y) \leq c \cdot s \), then \( P[h(x) = h(y)] \leq p_2 \)

where the probability is taken over choosing \( h \) from \( \mathcal{F} \), and \( c < 1 \), \( p_1 > p_2 \)
How to use LSH to find nearest neighbor

The main idea

Pick a hashing function $h$ from appropriate family $\mathcal{F}$

Preprocessing

- Compute $h(x)$ for all objects $x$ in our available dataset

On arrival of query $q$

- Compute $h(q)$ for query object
- Sequentially check nearest neighbor in “bucket” $h(q)$
Locality sensitive hashing I
An example for bit vectors

- Objects are vectors in \( \{0, 1\}^d \)
- Distances are measured using Hamming distance

\[
d(x, y) = \sum_{i=1}^{d} |x_i - y_i|
\]

- Similarity is measured as nr. of common bits divided by length of vector

\[
s(x, y) = 1 - \frac{d(x, y)}{d}
\]

- For example, if \( x = 10010 \) and \( y = 11011 \), then \( d(x, y) = 2 \)
  and \( s(x, y) = 1 - 2/5 = 0.6 \)
Consider the following “hashing family”: sample the $i$-th bit of a vector, i.e. $\mathcal{F} = \{ f_i | i \in [d] \}$ where $f_i(x) = x_i$

Then, the probability of collision

$$P[h(x) = h(y)] = s(x, y)$$

(the probability is taken over choosing a random $h \in \mathcal{F}$)

Hence $\mathcal{F}$ is $(s, cs, s, cs)$-sensitive (with $c < 1$ so that $s > cs$ as required)
Locality sensitive hashing III
An example for bit vectors

- If gap between $s$ and $cs$ is too small (between $p_1$ and $p_2$), we can amplify it:
  - By stacking together $k$ hash functions
    - $h(x) = (h_1(x), \ldots, h_k(x))$ where $h_i \in \mathcal{F}$
    - Probability of collision of similar objects decreases to $s^k$
    - Probability of collision of dissimilar objects decreases even more to $(cs)^k$
  - By repeating the process $m$ times
    - Probability of collision of similar objects increases to $1 - (1 - s)^m$
  - Choosing $k$ and $m$ appropriately, can achieve a family that is $(s, cs, 1 - (1 - s^k)^m, 1 - (1 - (cs)^k)^m)$-sensitive
Locality sensitive hashing IV

An example for bit vectors

Here, \( k = 5, m = 3 \)

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1
\end{array}
\]
Locality sensitive hashing V
An example for bit vectors

Collision probability is $1 - (1 - s^k)^m$
Similarity search becomes..

### Pseudocode

#### Preprocessing
- **Input**: set of objects $X$
- **for** $i = 1..m$
  - **for each** $x \in X$
    - stack $k$ hash functions and form $x_i = (h_1(x), .., h_k(x))$
    - store $x$ in bucket given by $f(x_i)$

#### On query time
- **Input**: query object $q$
- $Z = \emptyset$
- **for** $i = 1..m$
  - stack $k$ hash functions and form $q_i = (h_1(q), .., h_k(q))$
  - $Z_i = \{ \text{objects found in bucket } f(q_i) \}$
  - $Z = Z \cup Z_i$
- Output all $z \in Z$ such that $s(q, z) \geq s$
For objects in \([1..M]^d\)

The idea is to represent each coordinate in unary form

- For example, if \(M = 10\) and \(d = 2\), then \((5, 2)\) becomes \((1111000000, 1100000000)\)

- In this case, we have that the \(L_1\) distance of two points in \([1..M]^d\) is

\[
    d(x, y) = \sum_{i=1}^{d} |x_i - y_i| = \sum_{i=1}^{d} d_{\text{Hamming}}(u(x), u(y))
\]

so we can concatenate vectors in each coordinate into one single \(dM\) bit-vector

- In fact, one does not need to store these vectors, they can be computed on-the-fly
Generalizing the idea..

- If we have a family of hash functions such that for all pairs of objects $x, y$

\[ P[h(x) = h(y)] = s(x, y) \]  \hspace{1cm} (1)

- We can then amplify the gap of probabilities by stacking $k$ functions and repeating $m$ times

- .. and so the core of the problem becomes to find a similarity function $s$ and hash family satisfying (1)
Another example: finding similar sets

Using the Jaccard coefficient as similarity function

Jaccard coefficient

For pairs of sets \( x \) and \( y \) from a ground set \( U \) (i.e. \( x \subseteq U, y \subseteq U \)) is

\[
J(x, y) = \frac{|x \cap y|}{|x \cup y|}
\]
Another example: finding similar sets II
Using the Jaccard coefficient as similarity function

Main idea

- Suppose elements in $U$ are ordered (randomly)
- Now, look at the smallest element in each of the sets
- The more similar $x$ and $y$ are, the more likely it is that their smallest element coincides
Another example: finding similar sets III
Using the Jaccard coefficient as similarity function

So, define family of hash functions for Jaccard coefficient:

- Consider a random permutation \( r : U \rightarrow [1..|U|] \) of elements in \( U \)
- For a set \( x = \{x_1, .., x_l\} \), define \( h_r(x) = \min_i \{r(x_i)\} \)
- Let \( \mathcal{F} = \{h_r | r \text{ is a permutation} \} \)
- And so: \( P[h(x) = h(y)] = J(x, y) \) as desired!

Scheme known as \textit{min-wise independent permutation} hashing, in practice inefficient due to the cost of storing random permutations.