Topics

• Document similarity
• MinHash
• String similarity
  • Edit Distance / Longest Common Subsequence
  • Sharding strings
• Dimension reduction

Document Similarity

• Want to be able to identify documents that are "very close" to each other
• Very large number of documents
• Individually pre-process documents
  • Save a small amount of data per document (sketch)
  • Perform similarity tests based on sketch

Jaccard Similarity

\[
\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}
\]

Let X be the characteristic vector for A where \(x_j\) is the multiplicity of item \(j\) and Y be the characteristic vector for B where \(y_j\) is the multiplicity of item \(j\).

\[
\text{Jaccard}(A, B) = \frac{\sum_j \min(x_j, y_j)}{\sum_j \max(x_j, y_j)}
\]

Representation scheme

• Tokenize document
• Break document into shards
• Hash each shard into a domain of size \(2^{64}\) (unsigned long)
• Treat as a bag of words*
• Use Jaccard similarity measure

* In this application, we use bag of words without multiplicity.
Similarity testing
• Identify document pairs that have high similarity by doing pairwise comparison
• Precompute hashes of shards – n shards for document of n tokens
• Cost of comparison is $O(n)$
• How to improve this: reduce the amount of information stored per document

MinHash
• $U$ is the domain (in this case, the hash of the shards, $\{0 \ldots 2^64\}$)
• Choose a random permutation $\pi$ on $U$
• Let $A \subseteq U$
• $\text{MinHash}(A) = \text{argmin } x \in A \pi(x)$
• $\text{MinHash}$ is the smallest element of $A$ under the random permutation

An amazing result

$\Pr[\text{MinHash}(A) = \text{MinHash}(B)] = \frac{|A \cap B|}{|A \cup B|} = \text{Jaccard}(A, B)$

Similarity of Strings
• String edit distance – how many edits to convert $S_1$ into $S_2$
• Edit operations: Add character, Remove character, (Change character)

Using the MinHash
• Identify document pairs where Jaccard$(A, B) \geq 0.95$
• Run MinHash with $k$ independent random permutations
• Number of times $\text{MinHash}(A) = \text{MinHash}(B)$ is a good estimate of Jaccard Similarity
• Compute the $k$ MinHashes for each documents as a sketch
• Comparison of documents requires $k$ comparisons

BARTHOLEMEWSIMPSON $\rightarrow$ KRUSTYTHECLOWN
Longest Common Subsequence

- C = c_1...c_g is a subsequence of A = a_1...a_m if C can be obtained by removing elements from A (but retaining order).
- LCS(A, B): A maximum length sequence that is a subsequence of both A and B.

Example:
- occurrence: `attacgct`
- occurrence: `tacgacca`
- bartholemewsimpson
- krustytheclown

**Edit Distance and LCS**

- String A has length n and B has length m.
- Suppose that A is converted to B by removing k characters and adding j characters.
- Number of unchanged characters is c = n - k = m - j.
- Edit distance d is k + j.
- n + m = 2c + k + j = 2c + d.
- d = n + m - 2c.

Minimizing the edit distance is maximizing the length of the common sequence.

**LCS Optimization**

- A = a_1a_2...a_m
- B = b_1b_2...b_n
- Opt[j, k] is the length of LCS(a_1a_2...a_j, b_1b_2...b_k).
- Optimization recurrence:
  
  - If a_j = b_k, Opt[j, k] = 1 + Opt[j-1, k-1]
  
  - If a_j ≠ b_k, Opt[j, k] = max(Opt[j-1, k], Opt[j, k-1]).

Opt[0,0] = Opt[0,k] = 0

**Code to compute Opt[n, m]**

```java
for (int i = 0; i < n; i++)
for (int j = 0; j < m; j++)
    if (A[i] == B[j])
        Opt[i, j] = Opt[i-1, j-1] + 1;
    else if (Opt[i-1, j] >= Opt[i, j-1])
        Opt[i, j] := Opt[i-1, j];
    else
        Opt[i, j] := Opt[i, j-1];
```
Computing the Longest Common Subsequence

LCS Performance

- Runtime is $O(n^2)$ for a pair of strings of length $n$
- Space requirement is $O(n^2)$
  - Which can be reduced to $O(n)$ by reusing rows
- Recovering the actual LCS is more work, but can also be done in $O(n)$ space

Experiment: compute the length of two random bit strings (alphabet size 2)

<table>
<thead>
<tr>
<th>N</th>
<th>Base 2 Length</th>
<th>Gamma</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>8096</td>
<td>0.8096</td>
<td>00:00:01.86</td>
</tr>
<tr>
<td>20000</td>
<td>16231</td>
<td>0.8116</td>
<td>00:00:07.45</td>
</tr>
<tr>
<td>30000</td>
<td>24317</td>
<td>0.8106</td>
<td>00:00:16.82</td>
</tr>
<tr>
<td>40000</td>
<td>32510</td>
<td>0.8128</td>
<td>00:00:29.84</td>
</tr>
<tr>
<td>50000</td>
<td>40563</td>
<td>0.8113</td>
<td>00:00:46.78</td>
</tr>
<tr>
<td>60000</td>
<td>48700</td>
<td>0.8117</td>
<td>00:00:58.06</td>
</tr>
<tr>
<td>300000</td>
<td>243605</td>
<td>0.8120</td>
<td>00:28:07.32</td>
</tr>
</tbody>
</table>

Space efficient implementation

```java
public int SpaceEfficientLCS()
{
    int n = str1.Length;
    int m = str2.Length;
    int[] prevRow = new int[m + 1];
    int[] currRow = new int[m + 1];
    for (int j = 0; j <= m; j++)
        prevRow[j] = 0;
    for (int i = 1; i <= n; i++) {
        currRow[0] = 0;
        for (int j = 1; j <= m; j++) {
            if (str1[i - 1] == str2[j - 1])
                currRow[j] = prevRow[j - 1] + 1;
            else if (prevRow[j] >= currRow[j - 1])
                currRow[j] = prevRow[j];
            else
                currRow[j] = currRow[j - 1];
        }
        for (int j = 1; j <= m; j++)
            prevRow[j] = currRow[j];
    }
    return currRow[m];
}
```

String similarity

- Edit distance
- Advantages – measure of distance between strings
- Flexibility in edit operation and weighting
- Disadvantages
  - Relatively inefficient: $O(n^2)$, heuristics may help for large strings or looking for similarity
  - Requires looking at the entire string

String similarity with shards

- Same basic idea as with documents
- Consider alphabets with a small number of characters, e.g., {a, c, t, g}
- Take shards as being strings of length $k$
  - $k$ a multiple of 32 would make sense for packing into long ints
  - Hashing and minhash sketches apply as for documents
- Domain characteristics may be important
  - Mutation rate / distribution in sequences

Coming next: Dimension reduction for $R^n$

- Consider the distance function $D(x, y) = 0$ if $x = y$, $D(x, y) = 1$ if $x \neq y$
- Suppose we have a domain $U$ and want to answer distance queries between a set of $n$ elements
- Natural solution is to use $\log_2 U$ bits to describe the elements
- Can we use less space if we want to approximately answer distance queries
Of course this is going to be hashing

• Choose a good hash function $h: U \rightarrow 2^{32}$
• Let $f_i(x) = h(x) \mod 2$
• 1 bit representation
  • If $x = y$, then $f_i(x) = f_i(y)$
  • If $x = y$, then $Pr[f_i(x) = f_i(y)] \leq \frac{1}{2}$
  • Property preserved with probability at least 50%
• Repeat with $k$ independent hash functions $h_1, \ldots, h_k$
  • If $x = y$, then $f_i(x) = f_i(y)$ for all $i = 1, \ldots, k$
  • If $x \neq y$, then $Pr[f_i(x) = f_i(y) \text{ for all } i = 1, \ldots, k] \leq 2^{-k}$
• To achieve error of $\delta$, we need to use $k = \lceil \log_2 1/\delta \rceil$