1 Finding Similar Items

This chapter discusses the various measures of distance used to find out similarity between items in a given set. After introducing the basic similarity measures, we look at how to divide a document into pieces called as k-shingles and use them to compare two documents. Next, the concept of min-hashing is introduced which provides a novel signature for the documents. Finally, we explore the topic of Locality Sensitive Hashing (LSH) which is a very neat method of computing nearest neighbors in a much more computationally efficient manner. Needless to say, all these techniques discussed in this chapter find several applications in the real-world today due to the massive explosion in the scale of data. Examples of modern day LSH applications include fingerprint matching, removing duplicate results from the search results page of search engines and matching music tracks by looking at segments of them (eg: apps like *Shazam*, *SoundHound* use related techniques to perform blazingly fast similarity matches in real-time).

1.1 Popular Measures of Similarity between items

In Recommender Systems, *collaborative filtering* is a widely used technique to recommend items to users that were liked by other users having similar interests. Suppose we wish to find out if two netflix users - Alice and Bob have the same movie viewing pattern. Given their watch history, which contains a set of movies they have watched in the past few months, if we can compute some notion of how similar these two sets are, then that gives us a rough indication of the similarity between Alice and Bob. Although not the most efficient way to solve this problem, this is a good start.

Consider the following concrete example, where M_1 and M_2 are Alice's and Bob's viewing history over the past month.
M_1 = {"ride along", "the hundred foot journey", "love is in the air", "it felt like love", "interstellar"}, and
M_2 = {"interstellar", "big men", "tarzan", "into the storm", "the hundred foot journey"} .

The ***Jaccard Similarity*** of the sets M_1 and M_2 is given by $\frac{|M_1 \cap M_2|}{|M_1 \cup M_2|}$, that is, the fraction of the elements which are common to both the sets given the total number of (combined) elements in the two sets. In this example, the Jaccard Similarity between Alice and Bob would be $\frac{2}{8} = \frac{1}{4}$. ***Jaccard Distance***[1] between M_1 and M_2 is then simply defined as $1 - \frac{1}{4} = \frac{3}{4}$.

Now, suppose we took the entire set of movies available on netflix, say a million of them and formed a million-dimensional euclidean space out of them (with each dimension representing a movie). Then Alice and Bob can be represented as two points in this euclidean space, where their corresponding position vectors will have 1's for the movie they watched recently and 0's everywhere else. The problem of finding similiarity between them boils down to computing the

[1] Although this section uses both the terms similarity and distance, they are just inverses of each other. For every $sim(.)$ measure, the corresponding $dist(.)$ measure can be expressed as $1 - sim(.)$, provided the similarity measure is normalized appropriately.

cosine similarity between their vectors m_1 and m_2, which is given by:

$$sim(m_1, m_2) = cos(m_1, m_2) = \frac{m_1 \cdot m_2}{\|m_1\|\|m_2\|} \tag{1}$$

Another way to measure the distance between the vectors m_1 and m_2, is to use the ***Hamming Distance*** which is defined as the number of components in which the two vectors differ. In our example, the hamming distance between Alice and Bob would be 3 (as they differ in 3 movies out of the 5 considered in the list).

Euclidean Distance is another widely used distance measure in such cases which is defined as:

$$d(m_1, m_2) = \sqrt{\sum_{i=1}^{d} (m_1^{(i)} - m_2^{(i)})^2} \tag{2}$$

where, d is the number of components or dimensions of m_1 and m_2. Also known as the l_2 norm between the vectors m_1 and m_2, this measure is a special case of the general l_p norm family. The l_p norm between two vectors is defined as:

$$d(m_1, m_2) = (\sum_{i=1}^{d} |m_1^{(i)} - m_2^{(i)}|^p)^{\frac{1}{p}} \tag{3}$$

As can be seen, on setting $p = 2$ we obtain the Euclidean norm as a special case and setting $p = 1$ we obtain the l_1 norm (also called the Manhattan Distance) between two vectors. In the extreme case, when $p = \infty$, we obtain the l_∞ norm, which is defined as largest difference in the dimensions (as the other dimensions do not matter when p goes to ∞). ie,

$$l_\infty = \max_i |m_1^{(i)} - m_2^{(i)}| \tag{4}$$

Earlier in this section when we computed the Jaccard Similarity between the sets M_1 and M_2, we counted an item to be in the intersection of both of them if had exactly the same movie name (ie, the movie strings matched textually). For example, both M_1 and M_2 had the movie "the hundred foot journey". However, if one of these movie names was stated in a different way or had spelling errors such as "the hundred ft journey", the above method would not work. Therefore we have to look for a way to match strings approximately tolerating all of these possibilities. ***Edit Distance*** is one such method that provides a measure of how far a source string is from a destination string by recording the number of character insertions, character deletions and/or character substitutions that need to be made to convert the source string into the target string. In our case, the edit distance between the strings *"the hundred foot journey"* and *"the hundred ft journey"* is equal to 2.

All the above measures we discussed above qualify to be a valid *distance metric* because they obey the following properties:
If x, y are two points and $d(\cdot)$ is the distance metric between them, then:

- distance is non-negative. ie, $d(x, y)$ is always ≥ 0

- distance between similar points is zero. ie, $d(x, y) = 0$ if $x = y$
- distance is symmetric. ie, $d(x, y) = d(y, x)$
- distances obey the triangle inequality. ie, $d(x, y) \leq d(x, z) + d(z, y)$. What this essentially means is that, when traveling from point x to y, there is really no benefit in traveling via some particular third point z.

1.2 Shingles and Min-Hashing

Given that we discussed about measuring similarities between sets in the previous section, how do we represent two documents in the form of sets so that we can use measures like Jaccard Similarity on them? One effective way to do this is by constructing a set of all k-length substrings out of a document called *shingles of length-k* or *k-shingles*. Note that each shingle could appear one or more times in the document. Shingles help in capturing the short portions of text that are often shared across the documents. Lets look at a toy example. Suppose our document D contains the text *"abcdabd"*, the set of all 2-shingles for D is $\{ab, bc, cd, da, bd\}$. Observe that when the document contains words delimited by spaces, depending on whether spaces are ignored or not, different shingles would be produced. Also, the length of the shingle determines the level of granularity desired in capturing the repetitive text across documents. What this means is that, with very low values of k (eg: say k=1), almost all documents will have common characters and hence all of them will have high similarity. On the contrary, if k is set to very large value, say equal to average size of all the documents being compared, there is a chance that there is hardly any repetitive segment found and hence all of the documents will have zero similarity. Thus choosing the appropriate size for the k-shingle is important and two factors to be considered are how long typically documents are and how large the typical set of characters is. A good rule of thumb in general is: *k should be picked large enough that the probability of any given shingle appearing in any given document is low*.

A key question to ask now is how to represent and store these sets of shingles? Suppose we have a corpus of email documents. Typically only letters and a general whitespace character appear in emails, so the total number of shingles (assuming for instance that k=5) would be $27^5 = 14,348,907$ shingles. Such large number of shingles may sometimes not even fit in the memory and even if they did, the number of pairs to compare similarities for will be astronishingly large. We need to compress the shingles in such a way that also lets us compare them easily. Fortunately, we can do this pretty effectively by representing these sets compactly into their *signatures* and then using the signatures to estimate similarities like Jaccard. The nature of these signatures is such that they do not guarantee exactly similarity of the sets they represent, but provide very close estimates, and these estimates get better when have larger signatures.

This is great, but how do we compute these signatures efficiently? This can be achieved thanks to a neat technique called ***MinHashing***. Lets look at an example to see what MinHash computes.

Going by the same example as earlier, suppose we have the data about the users on netflix

and what movies they watched recently. Let each movie $m \in \mathcal{M}$ be indexed by a unique id from the set $\{1 \dots M\}$ and each user $u \in \mathcal{U}$ be indexed by an alphabet in the set $\{A \dots Z\}$. Clearly, a given user is a set whose elements are equal to the movie-id's he watched, and each user is thus a subset of $\mathcal{M}$. Now, consider the two sets (or users) $A = \{1, 2\}$, $B = \{2, 3\}$.

Steps to compute MinHash:

- Pick a permutation π of the set $\mathcal{M}$ uniformly at random.
- Hash each subset $\mathcal{S} \subseteq \mathcal{U}$ to the minimum value it contains according to π

Assume that $\pi = (2 < 1 < 3)$, then the minhash of A is computed as $h(A) = 2$, as 2 is the minimum value in the set A according to the ordering imposed by π. Likewise, $h(B) = 2$ as well. Now, lets enumerate all the possible permutations along with their corresponding minhashes for the sets A and B. That gives us the signature for all the set of users (or documents - if we are looking for similar text documents). This will also lead us to an interesting observation.

$\pi = (1 < 2 < 3), h(A) = 1, h(B) = 2,$	$\pi = (2 < 3 < 1),$ **h(A) = 2, h(B) = 2**
$\pi = (1 < 3 < 2), h(A) = 1, h(B) = 3,$	$\pi = (3 < 1 < 2), h(A) = 1, h(B) = 3$
$\pi = (2 < 1 < 3),$ **h(A) = 2, h(B) = 2**,	$\pi = (3 < 2 < 1), h(A) = 2, h(B) = 3$

Shown here are all the possible 6 permutations for the set $\mathcal{U}$. The minhashes of the sets A and B are *equal* for 2 out of 6 permutations, (ie: $\frac{1}{3}$ times); but more importantly observe that this fraction $\frac{1}{3}$ is actually equal to the *Jaccard Similarity* between the sets A and B.

In other words, *the probability that the minhash function $h(\cdot)$ for a random permutation π produces the same value for two sets A and B equals the Jaccard Similarity between the sets A and B.* This is a remarkable connection. What this means is that minhashes can be used to calculate Jaccard Similarities in a much more quick and space-efficient manner.

Creating the compact signatures for the documents (or, users in this example) is now easy. We can think of the each movie $m \in \mathcal{M}$ as a row of a matrix (we call this the *signature matrix*) and each user $u \in \mathcal{U}$ as a column. In our original representation, each column would thus have a value (say 0 or 1) against the relevant row, but in the signature representation we replace these with the minhash value corresponding to the user and the permutation. Note that we have shrunk M rows of binary values to just one row of minhashes for that permutation. Computing the minhashes for say a 100 permutations, we obtain our final signature matrix composed of several rows of minhashes. It should be noted that however, the minhash functions used for each row could all be different (ie. $h_1()$, $h_2()$, ..). It is easy to see that the signature matrix has size $(100 \times \mathcal{U})$ much smaller than the original matrix $(\mathcal{M} \times \mathcal{U})$.

In reality, computing the permutations of the rows from the $\mathcal{M} \times \mathcal{U}$ matrix can be very expensive. Even picking a random permutation from millions of rows could be time-consuming. Thus, hash functions are used as technique to simulate the effect of a random permutation of the rows. These functions map the row numbers to as many buckets as there are rows. There is a small chance

that one of these functions might map two different rows to the same bucket; however in practice often number of rows is so large that these minimal collisions do not matter. In summary, instead of picking n permutations of the rows, n randomly chosen hash functions $h_1(), h_2(), \cdots h_n()$ are picked to operate on the rows. Rest of the process of constructing the signature matrix remains the same as discussed previously.

1.3 Locality Sensitive Hashing (LSH)

Although the technique of minhashing helps us compute signatures effectively and compute similarity of pairs of documents quickly, it does not alleviate the task of considering all possible pairs. If we have a million documents and use signatures of length 250, even if takes just a micro-second to compute the similarity of two signatures, we might still spend around 6 days computing the similarities across all pairs of documents. Therefore *we want to avoid looking at all existing pairs and focus only on those which have very high probability of being similar. Locality-Sensitive Hashing (LSH) provides a theory for how to choose such important pairs*. In many real-world use-cases, the saving in terms of computation and time obtained by using these techniques is massive.

LSHs represent similarities between objects using probability distributions over hash functions and operate on the key idea that *Hash collisions capture object similarity*.

1.3.1 LSH: Gap Definition

The original definition of LSH (also called the Gap Definition) states the following. Consider a similarity function over a universe U of objects, $S : U \times U \rightarrow [0, 1]$. An (r, R, p, P)-LSH for a similarity S is a probability distribution defined over a set $\mathcal{H}$ of hash functions such that:

- $S(A, B) \geq R \Rightarrow Pr_{h \in \mathcal{H}}[h(A) = h(B)] > P$
- $S(A, B) < r \Rightarrow Pr_{h \in \mathcal{H}}[h(A) = h(B)] < p$

for each $A, B \in U$. Also, $r < R$ and $p < P$.

What this means is that - *if the similarity between objects A and B is at least R, then the probability that their hashes collide (or are the same) is atleast P. Likewise, if the similarity between A and B is atmost r, then the probability that their hashes will be the same is atmost p*. Note that there is a gap between R and r, and between P and p; hence the name Gap Definition.

Intuitively, this gap can be visualized as shown in the figure shown below (Figure 1) resembling a step function.
The x-axis represents the similarity between a given pair of items R and y-axis represents the probability that the pair shares some bucket. As seen, there is a sudden jump in the probability of collision when the similarity goes from low to high. We would in the ideal case like this to change smoothly and this can be done by introducing a function $f(R) = 1 - (1 - p^k(R))^L$ which causes this step function to look like in fig (Figure 2) below. k and L are parameters which denote

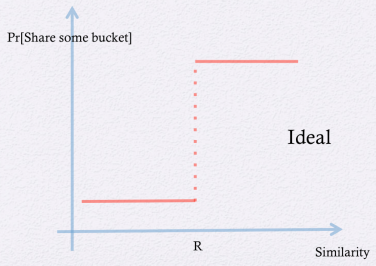

Figure 1:

number of rows per band and *number of bands*. By varying these the smoothness of the jump can be varied.

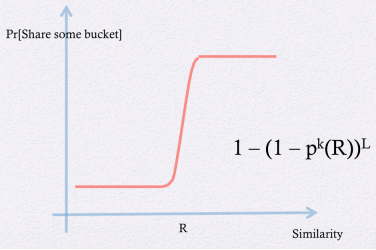

Figure 2:

The downside of reducing the gap extensively is that it would increase the computational cost.

Now lets look at how we derive the S-function shown in figure (Figure 2). Suppose we have a pair of documents with Jaccard Similarity R. The probability that the minhash signatures for this pair agree is $p(R)$. We can calculate the probability that these two documents (or rather their signatures) will become a candidate pair using the following sequence of observations:

- The probability that the signatures agree in all rows of a particular band is $p^k(R)$.
- The probability that the signatures do not agree in at least one row of a particular band is $1 - p^k(R)$.
- The probability that the signatures do not agree in all rows of any of the bands is $(1 - p^k(R))^L$.
- The probability that the signatures agree in all the rows of at least one band is $1 - (1 - p^k(R))^L$ which is the S-function. This is also the probability that the two documents become a candidate pair.

The *threshold* (or value of R) at which the probability of becoming a candidate is $\frac{1}{2}$, is a function of k and L. This threshold occurs at roughly somewhere where the rise in the S-curve is the steepest. For large values of k and L, *the pairs with similarity above the threshold are highly likely to become candidates while those below the threshold are unlikely to become candidates*. This is exactly

the ideal behavior we want. For most practical purposes, $(\frac{1}{L})^{\frac{1}{k}}$ is a good approximation of this optimal value for the threshold.

Below is an algorithm that uses LSH to find documents that are truly similar.

ALGORITHM

1: Choose a value of k and construct a set of k-shingles from the documents. An alternate is to hash the k-shingles to shorter bucket numbers to make it more efficient.
2: Sort the document-shingle pairs to order them by shingle.
3: Pick a length n for the minhash signatures. Compute the minhash signatures for all the documents.
4: Choose a threshold t that decides how similar documents have to be in order for them to be regarded as a desired *similar pair*.
5: Divide n into multiple bands L each of size k rows. Calculate the threshold t as $(\frac{1}{L})^{\frac{1}{k}}$. The choice of L and k also controls the number of false positives and false negatives. Depending on whether the priority is to avoid false negatives or false positives, the values of b and r need to be chosen to produce a threshold lower or higher than t respectively.
6: Construct candidate pairs by applying the LSH technique described earlier.
7: Examine each candidate pair's signatures and determine whether the fraction of components in which they agree is at least t.
8: Optionally, if the signatures are sufficiently similar, go to the documents themselves and check that they are indeed similar, rather than documents that, by luck, had similar signatures.

1.3.2 Constructing LSH families for common distance measures

We saw earlier that for a similarity metric (or equivalently every distance measure) to have an LSH, the gap definition needs to be satisfied. Lets see how we can find LSH's for some commonly used distance measures and the rationale behind them.

- **Hamming Similarity**
  Let HS(x, y) denote the Hamming Similarity between x and y. We then define *Sampling Hash* as $\mathcal{H} = \{h_1, .., h_n\}$, where $h_i(x) = x_i$ (ie. the i-th hash function outputs the i-th bit of x). It is easy to see that this hash function forms an LSH for the Hamming Similarity because:
  $$Pr[h(x) = h(y)] = Pr_i[h_i(x) = h_i(y)] = HS(x, y) \tag{5}$$
- **Jaccard Similarity**
  Let JS(x, y) denote the Jaccard Similarity between x and y. If we define a *MinHash* as we

discussed in section (reference section), then clearly this hash function forms an LSH for the Jaccard Similarity because:

$$Pr[h(A) = h(B)] = \frac{|A \cap B|}{|A \cup B|} = J(A, B) \tag{6}$$

This can also be easily verified using the fig below (Figure 3).

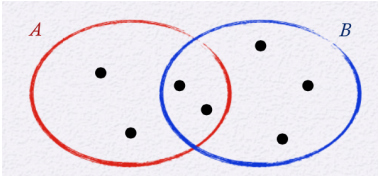

Figure 3:

- **Angle/Cosine Similarity**
  Given two vectors x and y, let $\theta(x, y)$ represent the angle between x and y (also known as the cosine similarity). It is trivial to see that $\theta(x, x) = 0$ and $\theta(x, y)$ takes the largest angle when $y = -x$. Next, we define *SimHash* as a hash function which is given as:
  $h(x) = sign(\langle x, r \rangle)$ where r is a random unit vector and $\langle , \rangle$ denotes the inner product. Lets visualize this using a picture as shown below (Figure 4). Clearly, our goal is to pick the

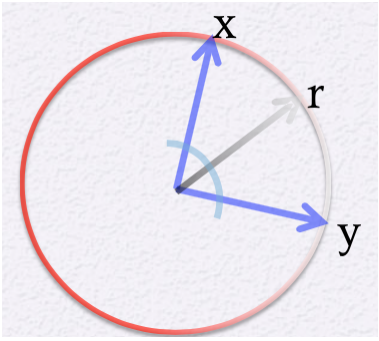

Figure 4:

random vector r in such a way that both the vectors x and y lie on the same side of r (or equivalently stated: r's inner product with both x and y should have the same sign). From the figure shown above (Figure 4), the blue arc represents $\theta(x, y)$ and as shown picking any random vector r within this angle will give us different signs for the inner product. Therefore, picking its complement angle, we can see that it provides us an LSH for the cosine similarity. ie,

$$Pr[h(x) = h(y)] = 1 - \frac{\theta(x, y)}{\pi} \tag{7}$$

1.4 Applications

- **Entity Resolution**
  Entity Resolution is the problem of finding whether two user records (such as mailing addresses of restaurants in a city) are the same or not. For simplicity lets assume the record consists of three fields only - *name, street, phone*. This is an essential problem to solve in real life applications where the records come from diverse data sources and each data sources could have encoded the address in different ways, with spelling errors and other missing information (such as zip code being absent, etc). A simple strategy that would work is to score a pair based on difference in the available fields. However, when we have large number of records, it is not feasible to score every possible pair (this number could rise up to a trillion for example) and hence we need an LSH to focus on the most likely candidates. How could we generate one? We use "three" hash functions for this purpose. The first one hashes the records based on the *name* field, second one based on the *street* field and third one based on the *phone* field. A clever trick can help us avoid computing these hashes. First, we sort the records based on name field; this would cause all records with identical names to appear consecutively and get scored for overall similarity of name, street and phone. Next, we apply another sorting based on the street field; the records get sorted by street and those with same street are scored. Finally, the records are sorted by phone field and the records with identical phones are scored.

- **Fingerprint Matching**
  Unlike many other identity matching applications, fingerprints are not matched by their image; instead a set of locations in each fingerprint which determine the uniqueness of the fingerprint (termed as *minutiae*) are identified and two fingerprints are matched by checking if a set of grid squares contains these minutiae in both the fingerprints. Typically we have two use-cases:
  - *many-one problem*: This is the case when a fingerprint has been found on a gun and we wish to compare it with all the fingerprints in a large database. This is the more common scenario.
  - *many-many problem*: In this case, we take the entire database and check if there are any pairs which represent the same individual. For instance, taking the first page of google search results and identifying if there are any pairs representing the same result.

  Lets look at how we can make these comparisons faster using a variant of LSH that suits this problem. Suppose the probability of finding a minutia in a random grid square of a random fingerprint is 20%. Next, assume that if two fingerprints come from the same finger and one has a minutia in a given grid square, then the probability that the other does too is 80%. We propose a LSH family of hash functions $\mathcal{F}$ in the following manner. Each $f \in \mathcal{F}$ is defined as below (using a context of *three specific grid squares*):
  $f(x, y) =$ yes, if two fingerprints x & y have minutiae in all three grid squares,
  $f(x, y) =$ no, otherwise.
  Another interpretation of the function f is that f sends to a single bucket all fingerprints that have minutiae in all three grid squares, and sends each other fingerprint to a bucket of

its own.

How to solve the many-one problem using this LSH? Invoking several functions from the family $\mathcal{F}$ we precompute their buckets of fingerprints to which they answer "yes". Then, given a fingerprint under investigation, we determine which of these buckets it belongs to and compare it with all the fingerprints found in any of those buckets.

How to solve the many-many problem using this LSH? We compute the buckets for each of these functions and compare all fingerprints in each of the buckets.

Analysis: How many functions do we need to achieve a reasonable probability of detecting a match? (without of course, having to compare the fingerprint on the gun with the millions of fingerprints in the database) Observe that, the probability that two fingerprints from different fingers would be in the same bucket for a function $f \in \mathcal{F}$ is $(0.2)^6 = 0.000064$. These two fingerprints would go into the same bucket if they each have a minutia in each of the three grid squares associated with f. Now, lets consider the probability that two fingerprints from the *same* finger end up in the bucket for f. Probabiliy that the first fingerprint has minutiae in each of the three grid squares belonging to f is $(0.2)^3 = 0.008$. Then, the probability that the other fingerprint also has minutiae in the same grid squares is $(0.8)^3 = 0.512$. Hence, if the two fingerprints are from the same finger, then there is a probabiliy of $0.008 \times 0.512 = 0.004096$ that they will both be in the bucket of f. Notice that this is not a big number, but if we make use of many more functions from the family $\mathcal{F}$, then we can arrive at a reasonable probability of matching fingerprints from the same finger. We also will be able to avoid having too many false positives.

1.5 Methods for High Degrees of Similarity

LSH based methods are very effective when the degree of similarity we want to accept is relatively low. However, there are use cases where we would want to find pairs which are almost identical and in such cases, there are other methods that are much faster. Since these methods are based on exact similarity, there are no false negatives as in the case of LSH.

- **Finding Identical Items**
  Assume for example that we need to find web-pages which are identical exactly, ie, character-by-character. Our goal is to avoid comparing every pair of documents. The first idea could be to hash documents based on either the first few characters of the document or the entire document itself. However, documents often have the same HTML header in which case the former fails and the later approach has a downside that we need to look at every character in the document to compute the hash, which could be expensive. An alternative approach would be to pick certain random locations in the document and compute hashes at those segments. If the documents are indeed similar, then such a hash function would return exactly the same value. Also, we do not need to consider documents of varying length anyways.
- **Length-Based Filtering**

Lets look at a more harder problem of finding from a huge collection of sets, all the pairs having a high Jaccard Similarity, say at least 0.9. We represent a set by sorting the elements of the universal set in some fixed order, and representing any set by listing its elements in this order. For instance, if the universal set contains the 26 lower-case letters and we use the normal alphabetical order; then, the set {d, a, b} is represented by the string *abd*.

Now, lets look at how we can exploit this construction. The simplest way is to sort the strings by length. Then, each string s is compared with those strings t that follow s in the list, but are not too long. Suppose there exists an upper bound J on the Jaccard Distance between two strings. Let L_x denote the length of a string x. Note that $L_x \leq L_t$. The intersection of the sets represented by s and t cannot have more than L_s elements, while their union will have at least L_t elements. Therefore, the Jaccard Similarity of s and t, $JS(s,t) \leq \frac{L_s}{L_t}$. This means in order for s and t to require comparison, it must be that $J \leq \frac{L_s}{L_t}$, or equivalently, $L_t \leq \frac{L_s}{J}$. This gives us an estimate of the length of the candidate strings t that we need to consider to compare s with in order to achieve our desired Jaccard Similarity.

- **Prefix Indexing**
  Retaining the same construction as above, lets look at other features of strings that can be exploited to reduce the number of pairwise comparisons. One such method is to create an index for each symbol from our universal set. For each string s, we select a prefix of s consisting of the first p symbols of s. How large p must be depends on L_s and J (defined in the method in previous paragraph). We add string s to the index for each of its first p symbols.

  The index for each symbol becomes a bucket of strings that must be compared. We need to be certain that any other string t such as $JS(s,t) \geq J$ will have at least one symbol in its prefix that also appears in the prefix of s.However, if $JS(s,t) \geq J$, but t has none of the first p symbols of s. Then the higest Jaccard Similarity that s and t can have occurs when t is a suffix of s, consisting of everything but the first p symbols of s. The JS of s and t would then be $\frac{(L_s-p)}{L_s}$. In order to be sure that we do not have to compare s with t, we must be certain that $J > \frac{(L_s-p)}{L_s}$. ie, $p \geq \lfloor (1-J)L_s \rfloor + 1$. Note that, we want p to be as small as possible, so that we do not index the string s in more buckets than we need to. Thus, we take $p = \lfloor (1-J)L_s \rfloor + 1$ as length of the prefix to be indexed.

- **Using Position and Suffix Indexes**
  Like the above methods, we can also index strings by the position of that character within the prefix as well as the length of the characterâĂŹs suffix âĂŞ the number of positions that follow it in the string. In the former case, we can reduce the number of pairs of strings that must be compared, because if two strings share a character that is not in the first position in both strings, then we know that either there are some preceding characters that are in the union but not the intersection, or there is an earlier symbol that appears in both strings. The later case reduces the number of pairs that must be compared, because a common symbol with different suffix lengths implies additional characters that must be in the union

but not in the intersection.

1.6 Acknowledgements

This document is a compressed version of *Chapter: 3 "Finding Similar Items"* from the book *"Mining of Massive Datasets"* by *Jure Leskovec, Anand Rajaraman and Jeffrey D. Ullman*. While most parts of this document are directly based on this chapter, some have been re-written from scratch. Another source of inspiration for this document is the talk by *Ravi Kumar (Google)* on LSH at *Machine Learning Summer School (MLSS 2012), UC Santa Cruz*. Some of the material in Section 1.3 (including the figures) are derived from slides based on this talk.