Exercise 1:

Raw code for visualisations on Diagram 1-3 is attached in the `Exercise 1 code` folder

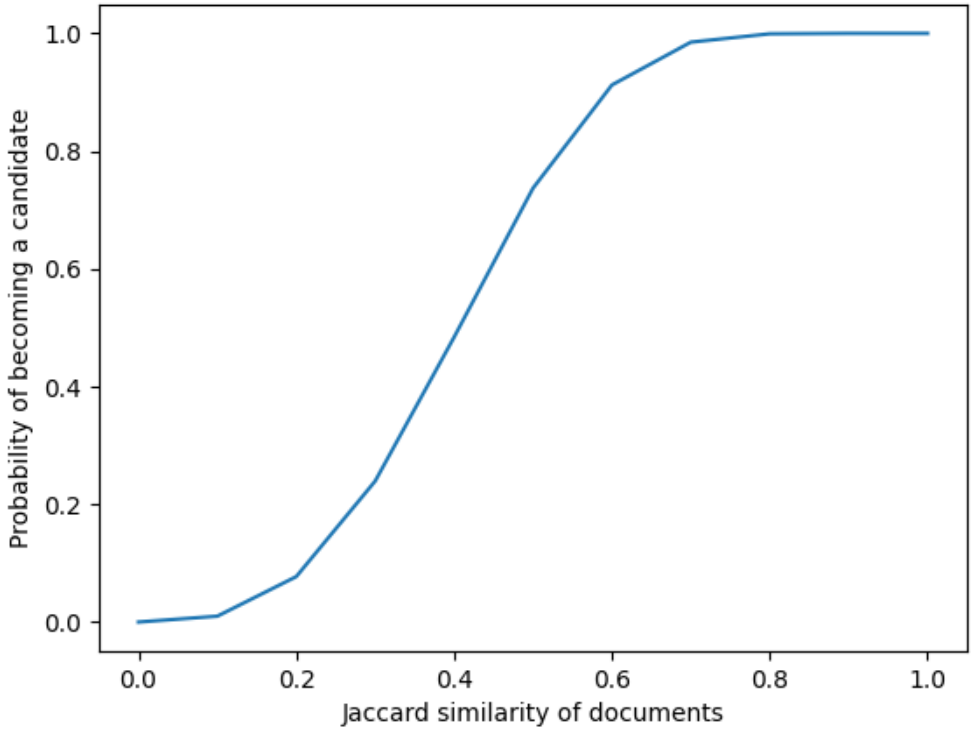


Diagram 1: r = 3 and b =10.

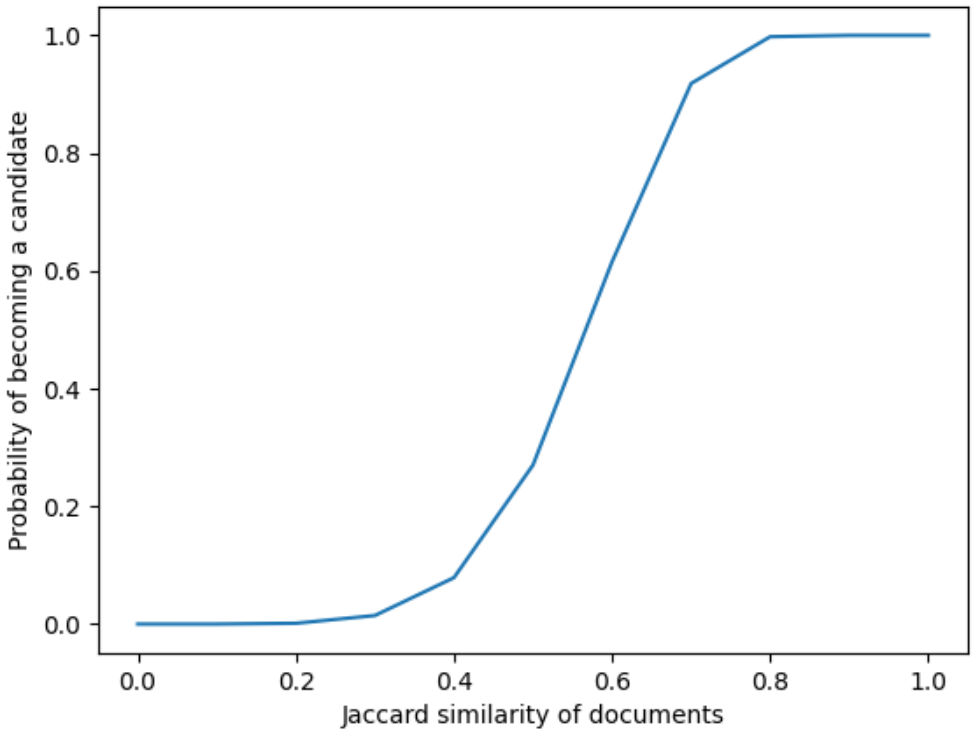


Diagram 2: r = 6 and b =20.

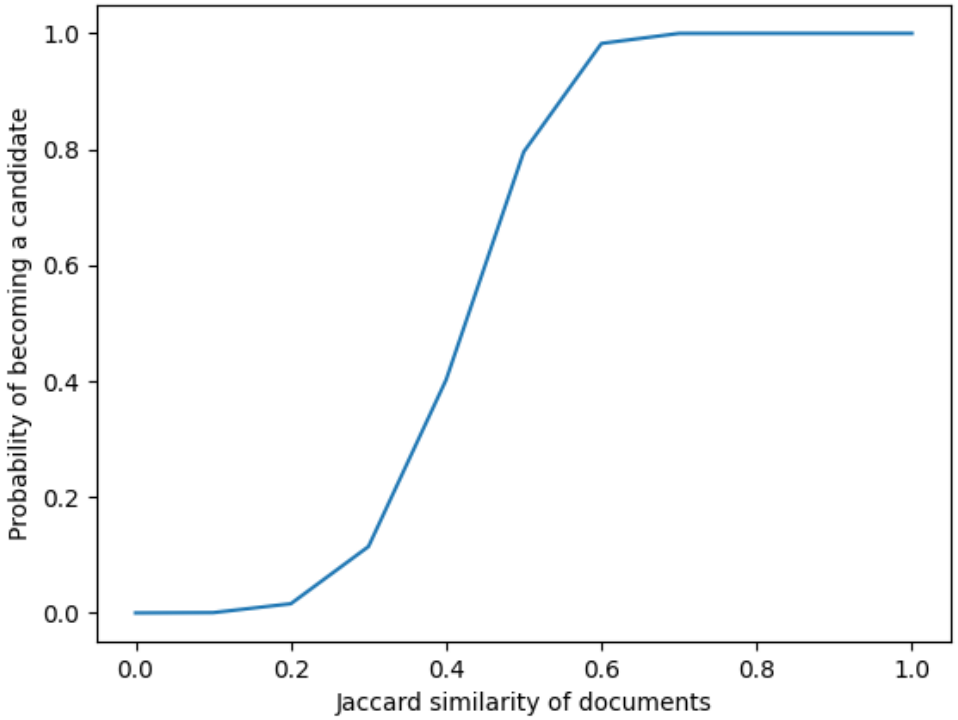


Diagram 3: r = 5 and b =50.

Exercise 2

1, $x = 10 \times 10^9$ (10 billion bits)

+) In case of using 3 hash functions :

$y = 3 \times 2 \times 10^9 = 6 \times 10^9$

- The probability that a bit remains 0 is : $e^{-\frac{6 \times 10^9}{10 \times 10^9}} = e^{-0.6}$

In order to be a false positive, a nonmember of S must hash 3 times to bits that are 1, and this probability is :

$$(1 - e^{-0.6})^3 \approx 0.0918$$

⇒ Therefore, the false positive rate when using 3 hash functions is 9.18%

+) In case of using 4 hash functions :

$y = 4 \times 2 \times 10^9 = 8 \times 10^9$

- The probability that a bit remains 0 is : $e^{\frac{-8 \times 10^9}{10 \times 10^9}} = e^{-0.8}$

In order to be a false positive, a nonmember of S must hash 4 times to bits that are 1, and this probability is :

$$(1 - e^{-0.8})^4 \approx 0.0919$$

⇒ Therefore, the false positive rate when using 4 hash functions is 9.19%

Exercise 2

2, We have false-positive rate is $f = \left(1 - e^{\frac{-km}{n}}\right)^k$

Rewrite $f = e^{\ln\left(1 - e^{\frac{-km}{n}}\right)^k}$

$= e^{k \ln\left(1 - e^{\frac{-km}{n}}\right)}$

Let $g = k \ln\left(1 - e^{\frac{-km}{n}}\right)$

Minimising g will minimise $f = e^g$

Now we have $\frac{dg}{dk} = k' \ln\left(1 - e^{\frac{-km}{n}}\right) + k\left(\ln\left(1 - e^{\frac{-km}{n}}\right)\right)'$

$= \ln\left(1 - e^{\frac{-km}{n}}\right) + k \frac{1}{1 - e^{\frac{-km}{n}}} \times \left(-e^{\frac{-km}{n}}\right) \times \left(\frac{-m}{n}\right)$

$= \ln\left(1 - e^{\frac{-km}{n}}\right) + \frac{km}{n} \frac{e^{\frac{-km}{n}}}{1 - e^{\frac{-km}{n}}}$ (1)

Let $u = e^{\frac{-km}{n}}$

$\Rightarrow \ln(u) = \frac{-km}{n}$

$\Rightarrow \frac{km}{n} = -\ln(u)$

Therefore, (1) will become: $\ln(1-u) - \ln(u)\left(\frac{u}{1-u}\right)$

Let $\frac{dg}{dk} = 0 \Rightarrow \ln(1-u) - \ln(u)\left(\frac{u}{1-u}\right) = 0$

$\Rightarrow \ln(1-u) = \ln(u)\left(\frac{u}{1-u}\right)$

$\Rightarrow (1-u)\ln(1-u) = u \ln(u)$

So, $u = (1-u)$ and $\ln(1-u) = \ln(u)$

$\Rightarrow 2u = 1$ $\Rightarrow 1 - u = u$

$\Rightarrow u = \frac{1}{2}$ $\Rightarrow 1 = 2u$

$\Rightarrow u = \frac{1}{2}$.

Since $u = e^{\frac{-km}{n}} \Rightarrow e^{\frac{-km}{n}} = \frac{1}{2}$

$\Rightarrow \frac{-km}{n} = \ln\left(\frac{1}{2}\right)$

$$\Rightarrow \frac{km}{n} = \ln(2)$$

$$\Rightarrow k = \frac{n}{m}\ln(2)$$

Plug $k = \frac{n}{m}\ln(2)$ into (1), we have: $\ln(1-e^{-\ln 2}) + (\ln 2)\frac{e^{-\ln 2}}{1-e^{-\ln 2}}$

$$= \ln\left(1-\frac{1}{2}\right) + (\ln 2)\frac{\frac{1}{2}}{1-\frac{1}{2}}$$

$$= \ln\left(\frac{1}{2}\right) + \ln\left(\frac{1}{2}\right)$$

$$= 0 \quad \text{(which is satisfied)}$$

In conclusion, $\boxed{k = \frac{n}{m}\ln(2)}$ will minimise the false-positive rate.

Exercise 3:

Write-up:
After research and review in implementations on Map Reduce system, in this program, we decided to create a new class type to redefine the adjacency list of friends inputted in the file. By creating a writable class called countMutualFriends, it assists us in reading the number of inputs through readFields and parsing through the friends as Long in write. This is further initialised as -1L in the constructor, it redefines the userID and mutual friends into respective variables using the method of the same name.

In this program, we will be implementing a similar layout in the main and run functions with the previous assignment, which is by taking in the values of the input file and return as lines of [User] [TAB] [Recommendations] in Text per line. Only one Map and Reduce functions are used in this implementation as we relied on several data structures, like HashMap for constant lookup of a list of friends to identify mutuality, List to store identified friends and TreeMap for sorting and recommendation filtering to a list of 10 friends.

The pseudocode can be explained as followed:

```
Mapper
        Split input line [UserID] [Friends]
                Parse the first value as UserID
                If there are friends detected on the user
                        Parse the remaining values as tokens of friends
                        While there is at least one friend
                                Add into the list of mutual friends
                                Write all friends as source node
                Indicate all edges of the adjacency list by using a nested for loop in the range of 0 to number of friends

Reducer
        Initialise a Hashmap for mutual friends
        Iterate all friends from mapper
                Declare Boolean to check if the current user is considered as a friend by the previous user
                If map consists of the user
                        Checks if the current user is a friend
                                If True
                                        Adds current user into the map
                                If False, but current user in map does not have a list of mutual friends
                                        Add list of friends
                        Else
                                If user and current node are not friends
                                        Add current user and the list of non-mutual friends
                                Else
                                        Add current user
        Declare a data structure (Tree Map) to store the user and list of friends' pair
        Insert all the pairs from Hash Map and sort in the descending order
                Get the first 10 recommendation entries from the Tree Map with commas
                Record the entries as strings
        Output the UserID and recommendation string
```

An alternative approach of the sorting algorithm in reducer will be the use of priority queues instead of Tree Map due to its reduction in time complexity for its nature to heapify the inputs when recommendations are inserted. However, Tree Map serves as a better match in this assignment because of its ability to implement comparator in a simpler implementation for both the key (UserID) and the value (recommendations).

Recommendations based on user ID:
924 = 15416,6995,924,43748,45881,2409,11860,439
8941 = 8939,8938,8941,8940,8943,8942,8944,8945,8946
8942 = 8939,8938,8941,8940,8943,8942,8944,8945,8946
9019 = 320,9023,9022,9021,9020,317,9019,9018,9017,9016
9020 = 320,9023,9022,9021,9020,9019,317,9018,9017,9016
9021 = 320,9023,9022,9021,9020,9019,317,9018,9017,9016
9022 = 320,9023,9022,9021,9020,9019,317,9018,9017,9016
9990 = 34642,13478,34299,9992,9993,9994,37941,9988,35667,13877
9992 = 9992,9993,9994,35667,9988,9989,9990,9991,9987
9993 = 34642,13478,34299,9992,9993,9994,37941,35667,9988,13877

Exercise 4:
For all questions given, a) is the maximum tail length for each stream element
b) is the resulting estimate of the number of distinct elements

First scenario. Data stream: 3, 1, 4, 6, 5, 9.
Q1)

Element	Hash	Binary representation	Count of trailing zeros
3	(2(3) + 1) mod 32 = 7	00111	0
1	(2(1) + 1) mod 32 = 3	00011	0
4	(2(4) + 1) mod 32 = 9	01001	0
6	(2(6) + 1) mod 32 = 13	01101	0
5	(2(5) + 1) mod 32 = 11	01011	0
9	(2(9) + 1) mod 32 = 19	10011	0

a) 0
b) $2^0 = 1$

Q2)

Element	Hash	Binary representation	Count of trailing zeros
3	(3(3) + 7) mod 32 = 16	10000	4
1	(3(1) + 7) mod 32 = 10	01010	1
4	(3(4) + 7) mod 32 = 19	10011	0
6	(3(6) + 7) mod 32 = 25	11001	0
5	(3(5) + 7) mod 32 = 22	10110	1
9	(3(9) + 7) mod 32 = 2	00010	1

a) 4
b) $2^4 = 16$

Q3:

Element	Hash	Binary representation	Count of trailing zeros
3	4(3) mod 32 = 12	01100	2
1	4(1) mod 32 = 4	00100	2
4	4(4) mod 32 = 16	10000	4
6	4(6) mod 32 = 24	11000	3
5	4(5) mod 32 = 20	10100	2
9	4(9) mod 32 = 4	00100	2

a) 4

b) $2^4 = 16$

Second scenario. Data stream: 4, 5, 6, 7, 10, 15.

Q4:

Element	Hash	Binary representation	Count of trailing zeros
4	(6(4) + 2) mod 32 = 26	11010	1
5	(6(5) + 2) mod 32 = 0	00000	0
6	(6(6) + 2) mod 32 = 6	00110	1
7	(6(7) + 2) mod 32 = 12	01100	2
10	(6(10) + 2) mod 32 = 30	11110	1
15	(6(15) + 2) mod 32 = 28	11100	2

a) 2

b) $2^2 = 4$

Q5:

Element	Hash	Binary representation	Count of trailing zeros
4	(2(4) + 5) mod 32 = 13	01101	0
5	(2(5) + 5) mod 32 = 15	01111	0
6	(2(6) + 5) mod 32 =17	10001	0
7	(2(7) + 5) mod 32 = 19	10011	0
10	(2(10) + 5) mod 32 = 25	11001	0
15	(2(15) + 5) mod 32 = 3	00011	0

a) 0

b) $2^0 = 1$

Q6:

Element	Hash	Binary representation	Count of trailing zeros
4	2(4) mod 32 = 8	01000	3
5	2(5) mod 32 = 10	01010	1
6	2(6) mod 32 = 12	01100	2
7	2(7) mod 32 = 14	01110	1
10	2(10) mod 32 = 20	10100	2
15	2(15) mod 32 = 30	11110	1

a) 3

b) $2^3 = 8$

Exercise 5:

Part 1:

Definition of Locality Sensitive Functions

It can be defined as functions that serve as an alternative for minhash functions in increasing the efficiency of the production of candidate pairs while working on measurements on space (such as set spaces) and distance (such as Jaccard distance).

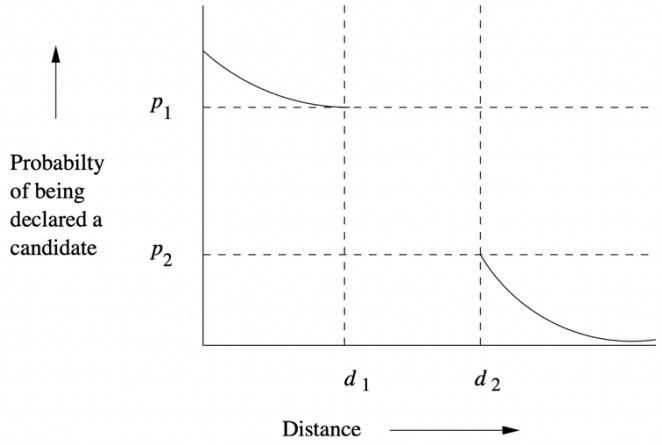
For context, a family of functions must fulfil three conditions, which are:

i) The likelihood of creating candidate pairs based on pairs with close proximity should be higher than pairs that are distant.

ii) Must be statistically independent for probability estimation while responding to events

iii) Must be more efficient in identifying candidate pairs than looking up to all the pairs in time complexity and combinable to mitigate false positives and negatives through combination of min-hash functions to produce the S-curve behaviour.

Reciting the diagram below that is taken from the textbook, given that it takes in two inputs x and y for the evaluation of candidate pairs, this requires a collection of locality sensitive functions, or a family of functions for each function of all distinct pairs. The function f decides whether x and y can be used as the candidate pair at the notation f(x) = f(y) to call f (x, y) as the function for the candidate pair. Using d1 and d2 as distances between two separate pairs depicted in the diagram below, when d1 is less than d2, if the distance between x and y is less than or equals to d1, the probability of the pairs x and y matches the notation and therefore declared as a candidate is at least the high probability p1, whereas, if it is more than d2, the probability would be at most the low probability, or p2. This illustrates a generic expectation of the probability of the function in a (d1, d2, p1, p2) sensitive family based on possible candidate pairs, regardless of the status of p1 and p2 or whether d1 and d2 are fixed.



1) Its usage for Jaccard distance

Since the detection of the family of locality-sensitive functions relies on assuming the distance measure to be the Jaccard distance, we can interpret a minhash function to the candidate pair of two item if and only if the function of the first item (h(x)) is equivalent to the function of second item (h(y)). If the Jaccard distance of x and y is less than or equals to d1, then the measure of relative possession of common attributes for x and y, or SIM(x,y), equals 1 – d(x,y), which is more than or equals to 1- d1 and less than or equals to 1- d2.

2) Further amplifications

Using previous implementation of (d1, d2, p1, p2)-sensitive family, we can construct a new family by the AND-construction of F that assign a range of members. As it mirrors the effect

of the given rows in a single band, the band creates a candidate pair from x and y if all rows in the band shows that x and y are equal. This means that the new family is (d1, d2, $(p1)^r$, $(p2)^r$) sensitive, which also means for any probability that declare (x,y) as the candidate pair by a member of F, we can exponent the probability value with a power of r to obtain the probability of a member of the new family.

Another construction to note is the OR-construction, which turns F's sensitivity by mirroring the effect of combining more than one band for x and y to become a candidate pair provided that if $f_i(x) = f_i(y)$ for any i values, which in turn determine the eligibility of x and y to be a candidate pair.

Both constructions AND and OR can be adjusted of the order, provides a p2 close to 0 and p1 close to 1. However, we will need to use a larger number of functions from the original family when more constructions and higher values of r and b are used. Therefore, the final family of functions with the best quality is dependent on the length of time for the functions from the family to be applied.

Part 2:

Families for Jaccard distance were discussed previously so in this section, we will be shown how to form locality-sensitive families for Hamming distance, cosine distance and Euclidean distance.

Hamming distance is the number of positions in which bit-vectors differ. Suppose we have a space of n-dimensional vectors, and h(x,y) denotes the Hamming distance between vectors x and y. $f_i(x)$ and $f_i(y)$ denote the ith bit of vectors x and y. We say the ith bit of vector x is equal to the ith bit of vector y (or $f_i(x) = f_i(y)$) if and only if vectors x and y agree in the ith position. The probability that $f_i(x) \neq f_i(y)$ for a randomly chosen i is equal to the number of components in which they differ, divided by the number of elements in each vector ($h(x,y) / n$). Therefore, the probability that $f_i(x) = f_i(y)$ for a randomly chosen i is exactly 1 minus the previously calculated result ($1 - h(x,y)/n$). The family **F** includes the functions $\{f_1, f_2, \ldots, f_n\}$ is a

$$(d_1, d_2, 1 - d_1/n, 1 - d_2/n)$$

-sensitive family of hash functions, for any $d_1 < d_2$. It means if the Hamming distance is at most d_1, the probability of $f(x) = f(y)$ is at least $1 - d_1/n$, while if the Hamming distance is at least d_2, the probability of $f(x) = f(y)$ is at most $1 - d_2/n$.

Cosine distance between two vectors is the angle θ between the vectors is the angle between them. For cosine distance, there is a technique called Random Hyperplanes, which is similar to min-hashing. Given two vectors x and y, we say $f(x) = f(y)$ if and only if the dot products $v_f.x$ and $v_f.y$ have the same sign, where v is a randomly picked vector which is normal to one of the hyperplanes. Random Hyperplanes method is a

$$(d_1, d_2, (180 - d_1)/180, (180 - d_2)/180)$$

-sensitive family for any d_1 and d_2. The way the sketch works is firstly picking some number of random vectors and hash our data for each vector. The result is the sketch of +1's and -1's for each data point. For example, let's say we pick a collection of random vectors $v_1, v_2, \ldots, v_n$, we first compute the dot products $v_1.x$, $v_2.x$, …, $v_n.x$. If any of the dot products produces positive value, then put +1 in the sketch vector otherwise put -1 in the sketch vector if it produces negative value. The resulting sketch will be a vector of +1's and -1's.

Euclidean distance is the distance between two points in an n-dimensional Euclidean space. The simple idea of LSH for Euclidean distance is to hash functions correspond to lines. Particularly, we pick a constant a, and partition the line into buckets of size a. Then, we hash each point to the bucket containing its projection onto the line.

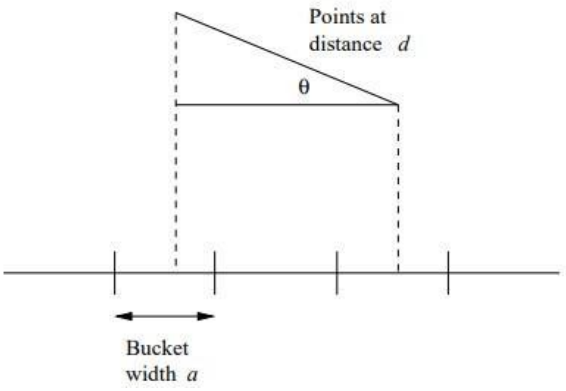


Figure 1: If $d < a$, then the chance the points are in the same bucket is small

If $d > a$, θ must be close to 90 degrees for the two points to be in the same bucket. The family **F** is a

($a/2, 2a, 1/2, 1/3$)

-sensitive family of hash functions. That is, for distance at most a/2, the probability that two points at that distance will fall in the same bucket is at least 1/2, while for distance at least 2a, the probability that two points at that distance will fall in the same bucket is at most 1/3.