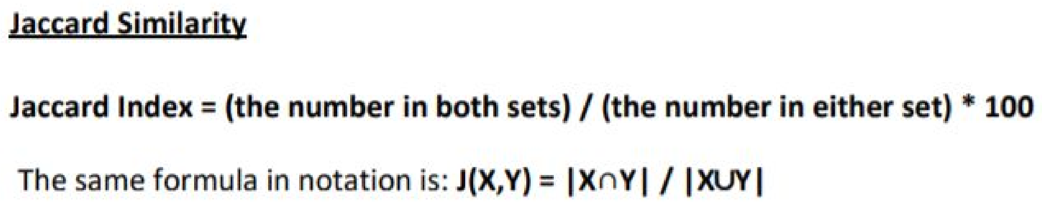
IR - Assignment - 2

Names: Abhinava Maddha (2018A8PS0844H) - [M ABHINAVA]
Nitin Chandra Surapar (2018A7PS0188H) - [NITIN CHANDRA SURAPUR]
Guddanti Mohit (2018AAPS0321H) - [GUDDANTI MOHIT]

1. **Shingling -**
   Having cleared our dataset, our next step is to split each document into shingles which we create depending on the user's input. It has to be clarified that the order of the words remained unchanged, as the sequence of the words(in our case gene code) is the one that gives meaning to the specific word shingling we applied. A k-shingle for a document is a sequence of k-tokens that appears in the doc. Tokens (or strings) can be words in our example. To be more precise our example shingles are "bags of words". The user is free to choose any natural positive number greater than 0. (k-shingles is nothing but the k-gram tokens of the sentence/gene code)
2. **Minhashing -**
   Originally the minhashing method aims to find ways to measure the similarity of documents by comparing the documents' signatures and not the documents themselves. In other words, minhashing technique is taking large sets of items and turning them to short signatures (vectors of a big number of integers) while preserving similarity. The length of the signature of each document depends on how many hash functions will be used on each document. As the assignment defines the number of hash functions as parametric for our implementation, we enabled the user to decide it.Minhashing uses Jaccard Similarity, which is given by the intersection of the sets divided by their union, for compares:



Jaccard Similarity

Jaccard Index = (the number in both sets) / (the number in either set) * 100

The same formula in notation is: **J(X,Y) = |X∩Y| / |XUY|**

All the hash functions to be used, whichever their number, come from the same function family, which is (Ax + B) % C, where x is the integer that came from a hashed shingle(shingle number), A and B are random coefficients, always different for different hash functions, and C is the prime number just greater than the total number of shingles obtained from all documents to make sure the row values being taken for the respective hash functions do not repeat as much as for a smaller c value. As x is already known each time for each document and each shingle and stored in a set, we have to give values to A, B and C. A and B took values from the function generateCoefficients(k), which every time called (once to pick as many A as the hash functions and secondly for the B) returns a unique number between 0 and the total number of shingles for one coefficient at a time. In this way, we ensure that there are no hash functions with the same A and B. Coefficient C is picked using the function NextPrime(N) to find the next prime number after the total number of shingles. We loop through all the documents and set the signature values as the minimum signature, i.e., (ax+b)%c, encountered for that document (initially set as infinity but here set as a number greater than the number of shingles in the corpus for ease). In the end we have a signature matrix(here a list with elements of signatures for every document in the corpus)

3. **LSH:**

   The user interacts with the program for the last time by giving the number of the bands. LSH input consists of the pre-calculated signature matrix, the number of the bands, document ID (whose similarity needs to be calculated) and number of nearest 'N' similar neighbours to be generated are explicitly asked by the function and the user can provide them. LSH begins by splitting the already existing signatures into band hashes. The number of bands hashes, as it was mentioned before, depends on the given band size. Next, the program creates a list of lists which includes all the pairs found in each bucket. This list is then flattened and converted into a dictionary which contains the docID and number of occurrences of the document in different buckets across the bands. After that, it collects all pairs found in the same bucket, which contain the document id that was given as input from the user. The LSH function returns at the end r the 'N' closest neighbours from the dictionary in the form of a list containing tuple pairs where the first entry contains docID and second entry it's frequency across all buckets similar to given docID. These are the candidate pairs. For the candidate pairs only, the program calculates their similarity and prints it out. In order to prove the optimality of hashing the program implements band similarity and shingle similarity and outputs the corresponding time taken and their respective similarity rates. The custom similarity measure used by us was ; [Similarity = no. of similar bands/ total number of bands]

Data Structures Used:

Numpy arrays (for faster computation), Dictionaries, Tuples and Lists.

Model Evaluation:

Precision and Recall - In order to estimate the LSH's performance, we need to calculate the False Positives and False Negatives. False positives are dissimilar pairs hashed to the same bucket, and false negatives are similar pairs that are not dispatched to the same bucket. It means that the false positives are pairs that are mistakenly considered as a candidate pair and the false negatives are pairs that are mistakenly not considered as a candidate pair. For this the assumption made is that all the documents retrieved using shingle comparison row by row are true positives (relevant).

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

TP = True positive

TN = True negative

FP = False positive

FN = False negative

Our model ensures real-time retrieval in least possible time. (No preprocessing to calculate Jaccard Similarity or finding similarity between docs is done, yet it gives quick results)