COMPUTER SCIENCE AND ENGINEERING

Project Report

Analysis & Modification Of Edit Distance Algorithm And Understanding its Applications

(FINDING THE PERCENTAGE SIMILARITY/DIFFERENCES)

Submitted By
Adarsh Baghel - 180001001
Akshay Prakash - 180001004

Under the Guidance of
Dr. Kapil Ahuja

Indian Institute Of Technology, Indore

Spring 2020

INTRODUCTION

Levenshtein Algorithm (Edit distance algorithm) is used to find the similarity between two strings. It is also referred to as Edit Distance Algorithm. The Levenshtein distance is a number that tells you how different two strings are. The higher the number, the more different the two strings are. More specifically, Levenshtein Distance tells about the total number of edits (insertions, deletions and substitutions) needed to make in one string to convert it to another.

For example, the Levenshtein distance between “kitten” and “sitting” is 3 since, at a minimum, 3 edits are required to change one into the other.

1. **k**itten → **s**itten (substitution of “s” for “k”)
2. sitt**e**n → sitt**i**n (substitution of “i” for “e”)
3. sittin → sittin**g** (insertion of “g” at the end).

An “edit” is defined by either an insertion of a character, a deletion of a character, or a replacement of a character.

We should all thank Vladimir Levenshtein, who came up with his algorithm in 1965. The algorithm hasn’t been improved in over 50 years and for good reason. According to MIT, it may very well be that Levenshtein’s algorithm is the best that we’ll ever get in terms of efficiency.

(Levenshtein_distance Wikipedia)

Motivation and resources

To detect plagiarism we have to measure similarity between two documents. We observe that most researchers use the following two types of similarity metrics.

1. String Similarity Metric: This method is commonly used by extrinsic plagiarism detection algorithms. Hamming distance is a well-known example of this metric which estimates number of characters different between two strings x and y of equal length, Levenshtein Distance is another example, that defines minimum edit distance which transform x into y, similarly, Longest Common Subsequence measures the length of the longest pairing of characters between a pair of strings, x and y with respect to the order of the characters.

2. Vector Similarity Metric: A vector based similarity metric is useful in calculating similarity between two different documents. Matching Coefficient is such a metric that calculates similarity between two equal length vectors.

Here we are implementing Levenshtein Distance Algorithm, as a sub part of a bigger problem (plagiarism detection). And we will try to understand its applications in different fields.

Expected Goals

1. **To Implementation and Algorithm Analysis of Levenshtein Algorithm.**
2. **To Modify the Levenshtein Algorithm.**
3. **To Match Human DNA sequences and find their similarities.**
4. **To Understanding Linguistic Differences using the normal and modified Levenshtein Algorithm.**
5. **To compare Images and determine their similarity using the normal and modified Levenshtein Algorithm.**

Understanding The Algorithm

The Levenshtein distance between two strings a,b (of length|a| and |b| respectively) is given by

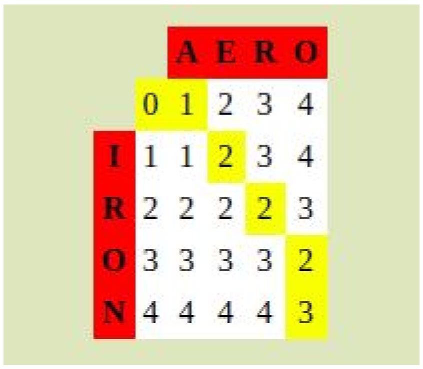
$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j)=0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise,

and $\operatorname{lev}_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b. i and j are 1-based indices.

(Levenshtein_distance Wikipedia)

Edit distance between two sequences can be computed by the dynamic programming based solution using the recurrence presented in the above Eq. Example : distance between “IRON” and “AERO”



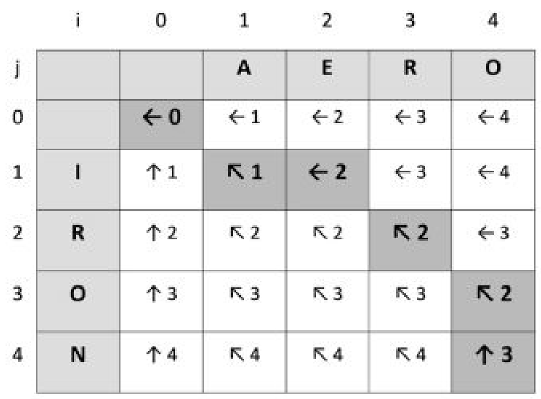
	A	E	R	O
0	1	2	3	4
I 1	1	2	3	4
R 2	2	2	2	3
O 3	3	3	3	2
N 4	4	4	4	3

Levenshtein Demo **visualiser**

Algorithm Implementation

```
#include <bits/stdc++.h>
using namespace std;

int d[100][100] = {0};

int LevenshteinDistance(string a, string b)
{
   int n = a.length();
   int m = b.length();

   //Initialization
   d[0][0] = 0;
   for (int i = 1; i <= n; ++i) //first column
       d[i][0] = i;
   for (int j = 1; j <= m; ++j) //first row
       d[0][j] = j;

   for (int i = 1; i <= n; ++i)
   {
       for (int j = 1; j <= m; ++j)
       {
           d[i][j] = min(
               d[i - 1][j - 1] + (a[i-1] == b[j-1] ? 0 : 1),//Substitution cost
               min(d[i - 1][j] + 1,                         //deletion
                   d[i][j - 1] + 1                          //insertion
                   ));
       }
   }
   return d[n][m];
}

int main()
{
   string a, b;
   a = "rover";
   b = "river";

   int n = a.length(), m = b.length();
   cout << "Number of edits to convert " << a << " to " << b << " is " <<
LevenshteinDistance(a, b) << endl;

   for (int i = 0; i <= n; ++i)
   {
       for (int j = 0; j <= m; ++j)
           cout << d[i][j] << " ";
       cout << endl;
   }
}
```

Algorithm Analysis



	i	0	1	2	3	4
j			A	E	R	O
0		← 0	← 1	← 2	← 3	← 4
1	I	↑ 1	↖ 1	← 2	← 3	← 4
2	R	↑ 2	↖ 2	↖ 2	↖ 2	← 3
3	O	↑ 3	↖ 3	↖ 3	↖ 3	↖ 2
4	N	↑ 4	↖ 4	↖ 4	↖ 4	↑ 3

There are three main operations we can perform

1) Replace a character

2) Insert a character

3) Delete a character

Note : No modification has to be performed if the particular characters match

Consider the last box to fill.

Case 1 : "N" != "O"

We take Cost To convert IRO -> AER which is 3

And REPLACE "N" with "O".

Cost = 3 + 1 = 4

Hence, Replacement Cost = 4

Case 2 : "N" != "O"

We take Cost To convert IRON -> AER which is 4

And INSERT "O".

Cost = 4 + 1 = 5

Hence, Insertion Cost = 5

Case 3 : "N" != "O"

We take Cost To convert IRO -> AERO which is 2

And DELETE "N" from IRON.

Cost = 2 + 1 = 3

Hence, Deletion Cost = 3

No Of Modifications we made, k = 3.

MAX Modification = max(String_A, String B)

M = 4

Similarity % = (M - k)/M *100

This algorithm works in **O(n*m) time.** It is visible through the two **for loops** which are required to traverse each and every possibility in this dynamic programming approach. Here the **space complexity is also O(n*m)** since we need to store the results in a 2d array**.** Further, we incorporate weighted edit distance and our own modification.

Modifying The Algorithm

- ### Modified Score Based Levenshtein Algorithm.

We looked to find a normal pattern in the original Levenshtein Algorithm, we saw that the original algorithm added a factor of 1, when doing any of the three modification **insertion**, **deletion** and **replacement.**

```
for (int i = 1; i <= n; ++i)
{
       for (int j = 1; j <= m; ++j)
       {
           d[i][j] = min(
               d[i - 1][j - 1] + (a[i-1] == b[j-1] ? 0 : 1),       //Substitution
               min(d[i - 1][j] + 1,                         //Deletion
                   d[i][j - 1] + 1                          //Insertion
                   ));
       }
}
```

But what if we can add separate costs for each operation, that's where we modified the original Levenshtein Algorithm and created a score based Levenshtein Algorithm.

```
for (int i = 1; i <= m; i++)
{
       for (int j = 1; j <= n; j++)
       {
           if (x[i - 1] == y[j - 1]) //When equal no penalty
           {
               dp[i][j] = dp[i - 1][j - 1];
           }
           else
           {
               dp[i][j] = min({dp[i - 1][j - 1] + pxy, //Replace penalty
                               dp[i - 1][j] + pgap,    //delete penalty
                               dp[i][j - 1] + pgap});  //insert penalty
           }
       }
}
```

This modification gives us the chance to penalty the replacement more than insertion or deletion and hence we were able to calculate the score, which is used in calculating similarity.

Algorithm Analysis

This new modified algorithm works in **O(n*m) time.** It is clearly visible through the two for loops which are required to traverse each and every possibility in this dynamic programming approach. Here the **space complexity is also O(n*m)** since we need to store the results in a 2d array**.** But with the score based approach we are able to get more accurate results.

Algorithm Applications

Edit distance has applications in many domains such as bioinformatics, spell checking, plagiarism checking, query optimization, speech recognition, and data mining. Traditionally, edit distance is computed by dynamic programming based sequential solution which becomes infeasible for large problems. So our analysis will be limited to a small set of problems only.

Human DNA Sequences Comparison

DNA sequencing is a process of determining three million nucleotide bases in order which consist of adenine, guanine, cytosine and thymine (A, T, G, C) in a DNA molecule. However, sequencing the genome is the determination of the nucleotide sequence of DNA bases in the genome or in the body of an organism. 11 Sequencing results are expressed in the form of a sequence of letter nucleotide bases in specific DNA sequence, for example AGTCCGCAGGCTCGGT. (**research paper [2] - in reference**)

Comparing Human # DNA Sequences

Sr no.	Dna Human 1	Dna Human 2	similarity
1	**gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag**	**gttggctctgacttgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag**	**(92.7928 lev,99.5496 mod)**
2	**gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag**	**gttggctctgagctgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag**	**(92.7928 lev,99.5496 mod)**

3	gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	gttggctctgacgtgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	(92.7928 lev,99.5496 mod)
4	gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	gttggctcggactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	(93.6364 lev,99.0909 mod)
5	gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	gttggctctgactgtaccaccatcaactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	(92.7273 lev,99.0909 mod)
6	gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	gttggctctgactgccaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	(91.8919 lev,98.6487 mod)
7	gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	gttggctcggactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	(93.6364 lev,99.0909 mod)
8	gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	gttggctctgactgtaccaccatcaactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	(92.7273 lev,99.0909 mod)
9	gttggctctgactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	gttggctcaggactgtaccaccatccactacaactacatgtgtaacagttcctgcatgggcggcatgaaccggaggcccatcctcaccatcatcacactggaagactccag	(91.8919 lev,98.6487 mod)

Comparing Human vs mice # DNA Sequences

Sr. no	Man	Mice	Similarity
1	TCTGTACTGTAGCTTAGGTAACGATCGA..	TCTGTACTGTAGCTAAGCTATCGATCGA..	(89.2857 lev,89.2857 mod)
2	TCTGTACTGTAGCTTAGGTMCGATCGA..	TCTGTACTCTAGCTMGCTATCGATCGA..	(77.7778 lev,85.1852 mod)

Understanding Linguistic Differences

To understand linguistic differences, we need to add a percent similarity calculator. Which is done as follows:

```
double CalculateSimilarity(int editDistance, int length_a, int length_b)
{
   return (1.0 - ((double)editDistance/(double)max(length_a, length_b)));
}
```

Percent formula for modified algorithm.

```
Return (1 - ((float)penT / ((float)max(length_a, length_b) * 2))) * 100;
```

This gives us a fair estimation of the percent similarity between the two strings.

Now towards **Linguistic differences** part, since languages are extremely vast comprising numerous words and sentences, to determine the linguistic differences would take us something more than Levenshtein Algorithm. But to limit out analysis and efforts, we compared some places where every language remains common. Such as

Comparing Similarity Misc (base language Hindi)

Language	Hindi <Language> (normal, modify)	Hindi <Language> (normal, modify)
Bengali Punjabi Marathi Gujariti	haathee Hati (28.5714 lev,50 mod) haathee Hathi (42.85 lev,53.57 mod) haathee Hatti (28.57 lev,46.42 mod) haathee Hathi (42.85 lev,53.57 mod)	gaay garu (50 lev,50 mod) gaay ga (50 lev,62.5 mod) gaay gaya (50 lev,62.5 mod) gaay gaya (50 lev,62.5 mod)
Bengali Punjabi Marathi Gujariti	manushy manusa (71.42 lev,75 mod) manushy adami (14.28 lev,39.28 mod) maŋusa manushy (37.5 lev,46.87 mod) manushy manasa (57.14 lev,60.7 mod)	devata debata(83.3 lev,83.3 mod) devata devata (100 lev,100 mod) devata deva (66.67 lev,75 mod) bhagavana devata(33.3 lev,52.7)
Bengali Punjabi Marathi Gujariti	paanee Jala (16.66 lev,41.66 mod) paanee Pani (33.3333 lev,50 mod) paanee Pani (33.3333 lev,50 mod) paanee Pani (33.3333 lev,50 mod)	phool phula (40 lev,50 mod) phool phula (40 lev,50 mod) phool phula (40 lev,50 mod) phool phula (40 lev,50 mod)
Bengali Punjabi Marathi Gujariti	sooraj surya (33.33 lev,45.83 mod) sooraj suraja (50 lev,58.3333 mod) sooraj surya (33.33 lev,45.83 mod) sooraj surya (33.33 lev,45.83 mod)	

Comparing Similarity week Days (base language Hindi):

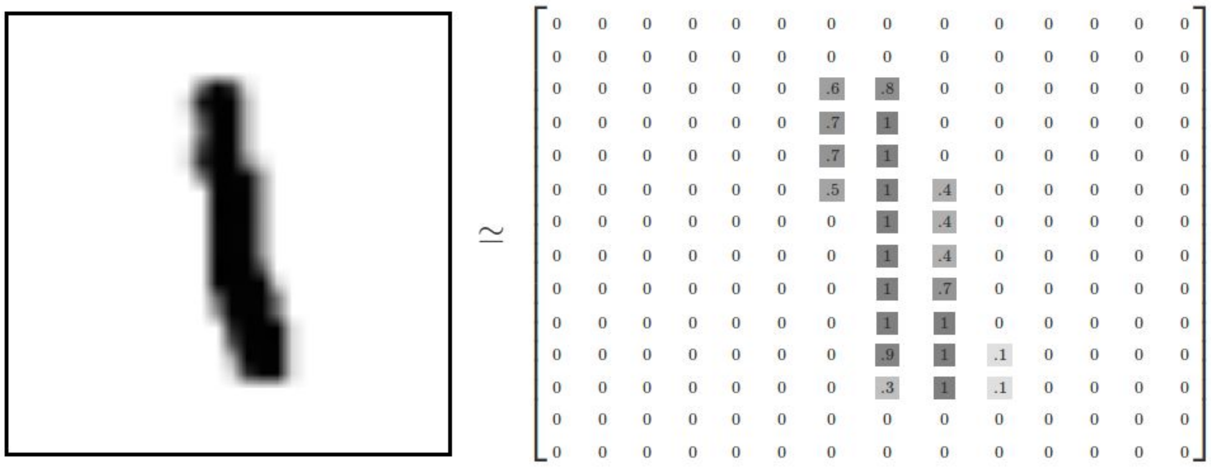
Language	Hindi <Language> (normal, modify)	Hindi <Language> (normal, modify)

Marathi Bengali Punjabi Bhojpuri	Somvaar Somavar(71.42,85.71 mod) Somvaar Shombar(57.14,71.42 mod) Somvaar Somvaar(100,100 mod) Somvaar Somaar(85.71,92.85 mod)	Mangalavaar Mangalvaar(91,95.4 mod) Mangalvaar Mongolbar(60,65 mod) Mangalvaar Mangalvaar(100,100 mod) Mangalvaar Mangar(60,80 mod)
Marathi Bengali Punjabi Bhojpuri	Budhvaar Budhavar(75,87.5 mod) Budhvaar Budhbar(75,81.25 mod) Budhvaar Budhvaar(100, 100 mod) Budhvaar Budh(50, 75 mod)	Shukravaar Sukravar (80 ,90 mod) Shukravaar Shukrobar (70,75 mod) Shukravaar Shukarvaar (80,90 mod) Shukravaar Shukkar (60,75 mod)
Marathi Bengali Punjabi Bhojpuri	Shanivaar Shanivar(88.8,94 mod) Shanivaar Shonibar(66.67,72 mod) Shanivaar Shanivaar(100,100 mod) Shanivaar Sanichar(66.67,72 mod)	Ravivaar Ravivar(87.5,93.75 mod) Ravivaar Robibar(50,56.25 mod) Ravivaar Aitvaar(50,68.75 mod) Ravivaar Etvaar(50,62.5 mod)

Comparing Similarity Numerals (base language Hindi):

Language	Hindi <Language> (normal, modify)	Hindi <Language> (normal, modify)
Marathi Bengali Punjabi Marwadi Bhojpuri	Ek Ek (100 lev,100 mod) Ek Ek (100 lev,100 mod) Ikk Ek (33.3333 lev,50 mod) Ek Ek (100 lev,100 mod) Ek Ek (100 lev,100 mod)	Don Do (66.66 lev,83.33 mod) Dui Do (33.33 lev,50 mod) Do Do (100 lev,100 mod) Doy Do (66.66 lev,83.33 mod) Dui Do (33.33 lev,50 mod)
Marathi Bengali Punjabi Marwadi Bhojpuri	Teen Tin (50 lev,62.5 mod) Teen Tin (50 lev,62.5 mod) Teen Tinn (50 lev,50 mod) Teen Tin (50 lev,62.5 mod) Teen Tin (50 lev,62.5 mod)	Chaar Char (80 lev,90 mod) Chaar Char (80 lev,90 mod) Chaar Cha (60 lev,80 mod) Chaar Chyar (80 lev,80 mod) Chaar Cari (40 lev,70 mod)
Marathi Bengali Punjabi Marwadi Bhojpuri	Paanch Pach (66.66 lev,83.33 mod) Paanch Pach (66.66 lev,83.33 mod) Paanch Punj (33.33 lev,66.66 mod) Paanch Pach (66.66 lev,83.33 mod) Paanch Pach (66.66 lev,83.33 mod)	Cheh Saha (0 lev,25 mod) Chchoy Cheh (50 lev,66.66 mod) Cheh Che (75 lev,87.5 mod) Cheh Chaw (50 lev,50 mod) Cheh Chae (50 lev,75 mod)
Marathi Bengali Punjabi Marwadi Bhojpuri	Saat Sat (75 lev,87.5 mod) Saat Shat (75 lev,75 mod) Saat Satt (75 lev,75 mod) Saat Sat (75 lev,87.5 mod) Saat Sat (75 lev,87.5 mod)	Aath Ath (75 lev,87.5 mod) Aath At (50 lev,75 mod) Aath Ath (75 lev,87.5 mod) Aath At (50 lev,75 mod) Aath Ath (75 lev,87.5 mod)
Marathi Bengali Punjabi Marwadi Bhojpuri	Nao Nau (66.6667 lev,66.6667 mod) Nao Noy (33.3333 lev,66.6667 mod) Naum Nao (50 lev,62.5 mod) Nao Naw (66.6667 lev,66.6667 mod) Nao Nao (100 lev,100 mod)	Daha Das (50 lev,62.5 mod) Dosh Das (50 lev,62.5 mod) Dass Das (75 lev,87.5 mod) Das Das (100 lev,100 mod) Das Das (100 lev,100 mod)

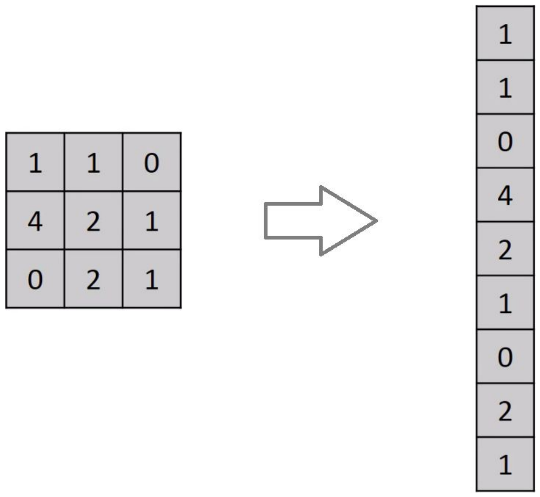
Using Modified Algorithm To Study Similarity Between Images Present In The MNIST Dataset



$$\simeq \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&.6&.8&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.5&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.7&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&.9&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&.3&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

Each image is made of pixels, a lot of them. For now we are considering only B&W images, so each pixel in the image has value ranging from 0-255. We resize the input image to 28*28 pixels and convert it to an array of 28 rows and 28 columns where each cell of this 2D array corresponds to a pixel value (ranging from 0-255). Later for each pixel we multiply it by a factor (90/255) and round it to the nearest integer. Since images in MNIST dataset does not have that many features, hence there's not much feature loss due to this transformation.

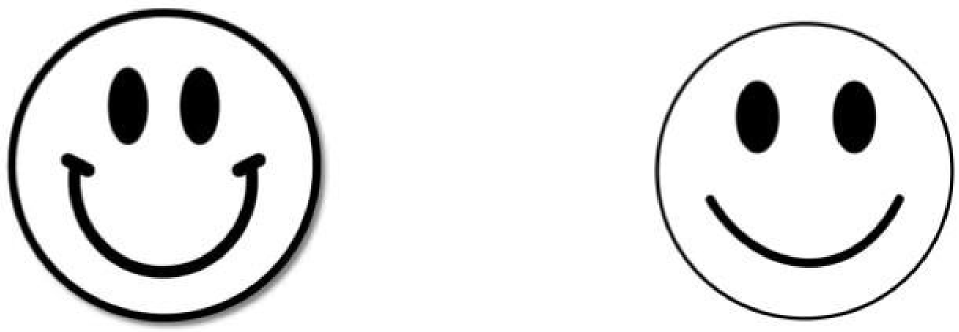
Now each pixel in the array has integral values ranging from 0-90, simultaneously we create a map of <integer, distinct_ascii_printable_characters> so that every pixel value can be represented by a distinct character.



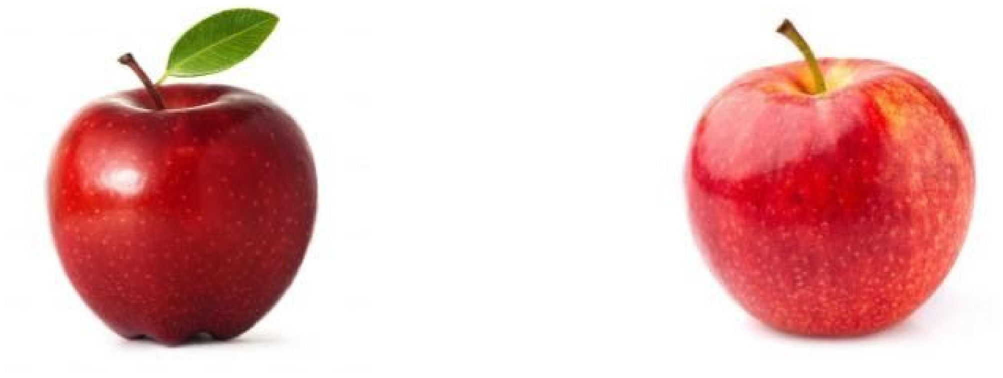
We flatten this array and assign a distinct character to a distinct pixel value. In this way we make a string representation of the image.

Now we try to get similarity % for some sample images.

```
Similarity % : 88.2653 (mod), 78.5714
```

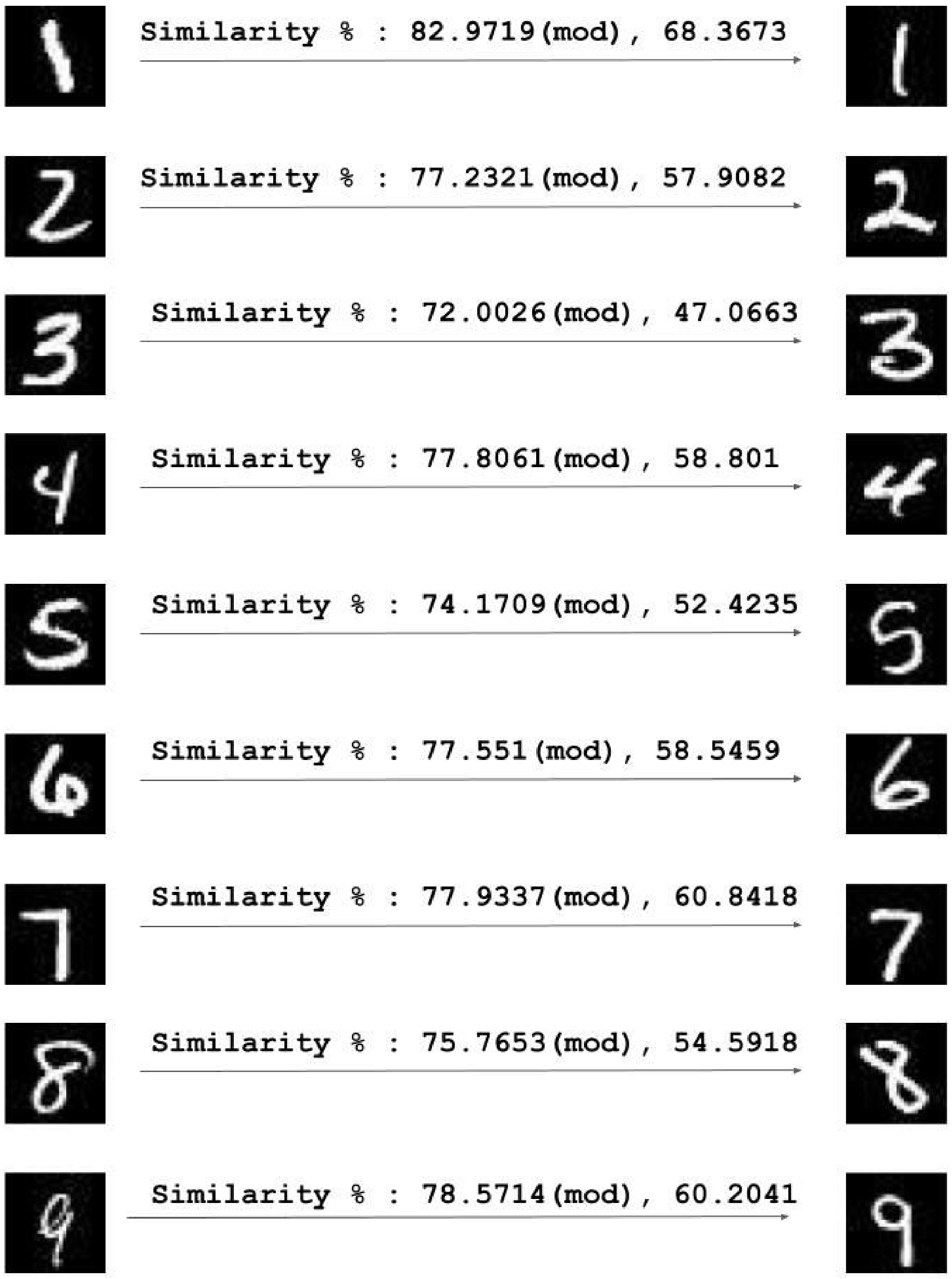


```
Similarity % : 70.4719 (mod), 42.3469
```



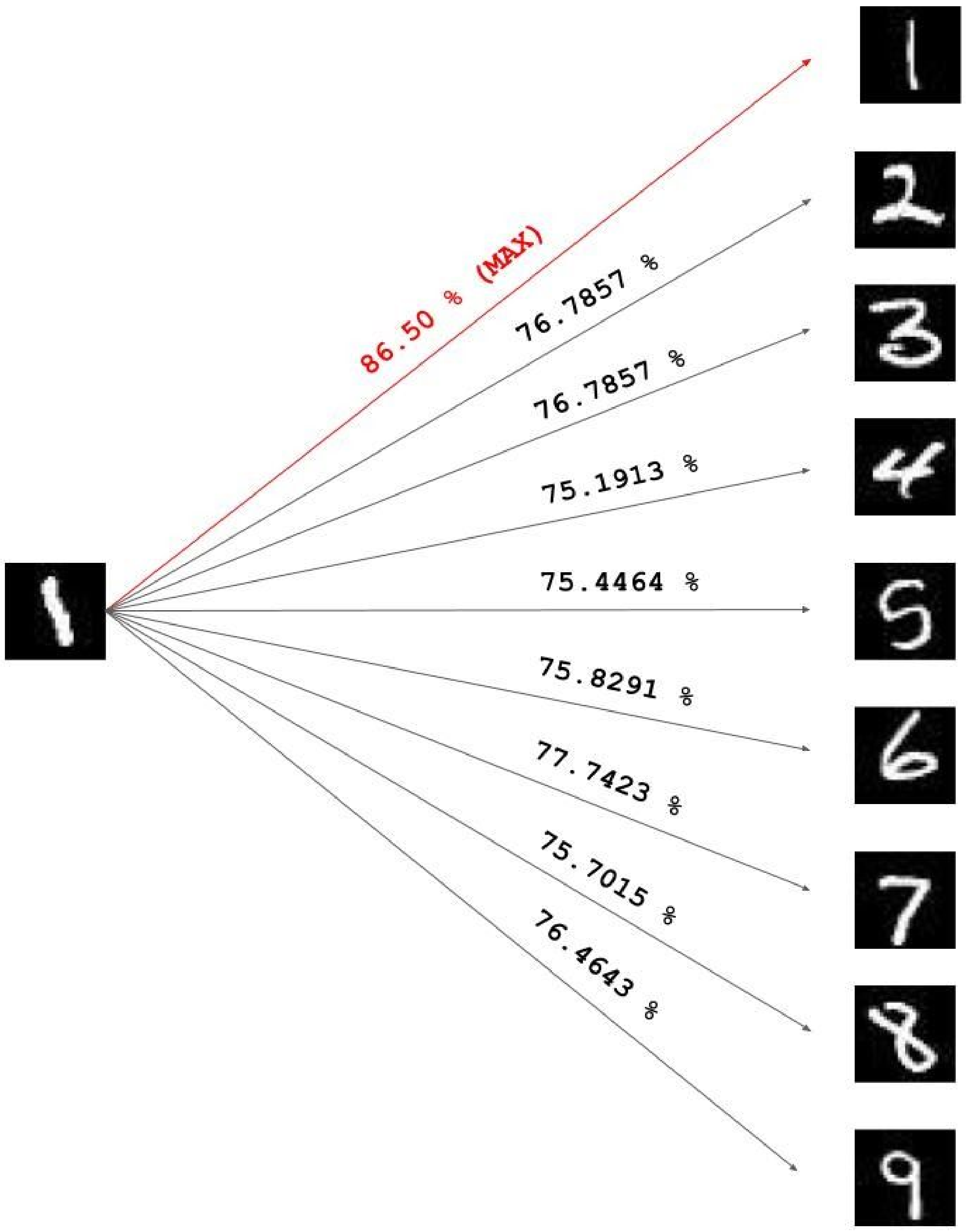
(Image source from pixabay CC0 images)

Similarity Percentage Between Various Images



```
Similarity % : 82.9719(mod) , 68.3673
Similarity % : 77.2321(mod) , 57.9082
Similarity % : 72.0026(mod) , 47.0663
Similarity % : 77.8061(mod) , 58.801
Similarity % : 74.1709(mod) , 52.4235
Similarity % : 77.551(mod) , 58.5459
Similarity % : 77.9337(mod) , 60.8418
Similarity % : 75.7653(mod) , 54.5918
Similarity % : 78.5714(mod) , 60.2041
```

Similarity Percentage Between 1 & other numbers



As we see, however we get the correct ans but the similarity % between two logically dissimilar images is also quite high and hence this algorithm is not much effective in image comparison here.One reason for this algo being not much effective here is that **most of the pixels are black** here (having pixel value of 0) and hence most part remains similar between two images even though they are logically dissimilar. **We are still making tweaks to improve this**.

REFERENCES

1. Levenshtein Distance- From Wikipedia, link
2. Hussain A Chowdhury and Dhruba K Bhattacharyya, Plagiarism: Taxonomy, Tools and Detection Techniques, link (Plagiarism detection)
3. Ethan Nam, Understanding the Levenshtein Distance Equation for Beginners link (levenshtein-distance)
4. The Levenshtein Algorithm, by Cuelogic Insights link
5. Lailil Muflikhah, Identifying Cancer Disease through Deoxyribonucleic Acid (DNA) Sequential Pattern Mining (DNA example)
6. C#: Calculating Percentage Similarity of 2 strings link (similarity of 2 strings)
7. The MNIST dataset provided in a easy-to-use CSV format MNIST in CSV (MNIST Dataset)
8. Sequence Alignment problem, GFG, Sequence Alignment problem (algorithm intuition)