Character Segmentation of Hindi and English Text

Students :

1. Kotwade Venkatesh Laxman (B18CSE023)
2. Tanmay Santosh Pandit (B18CSE055)

Guide : Dr. Gaurav Harit

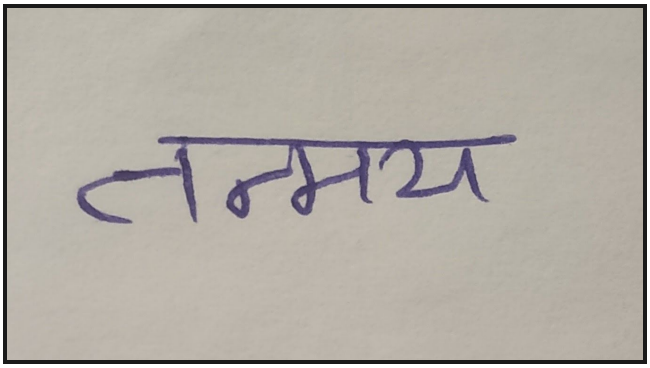
Abstract:

Character Segmentation is one of the most essential steps for any Optical Character Recognition system. The problem has attracted the attention of many researchers till now due to its challenging nature and vast applications.
In this project, we have used different approaches to segment both Hindi and English text and analysed their results. The entire work could be divided into 4 major parts :

1. A novel approach to segment Hindi text by making use of **Djikshtra's algorithm for shortest paths with randomisation.**
2. An alternative approach to segment Hindi text by using dynamic programming based zoning method
3. Implementation of a dynamic programming based **non-uniform slant correction** algorithm to deslant the slanted cursive English text as a preprocessing for segmentation.
4. A heuristic approach based on local minima to segment the deslanted and skeletonised English text.

1. Segmentation Approach using Djikshtra's Algorithm with Randomisation :

The main idea behind this approach is to find the path from top to bottom which crosses the minimum number of black pixels, with minimum horizontal deviation from the source point. The approach is mainly viable and tested on Indian languages.
We pose this as a graph problem. The image is considered as a graph with its pixels as nodes and adding an edge between every pair of neighbouring pixels. Since we want our path to have as many white pixels as possible, we assign relatively high weight to the edge containing at least one black pixel than other edges. So, now it's a shortest path problem which is solvable in polynomial time by Djikshtra's algorithm.

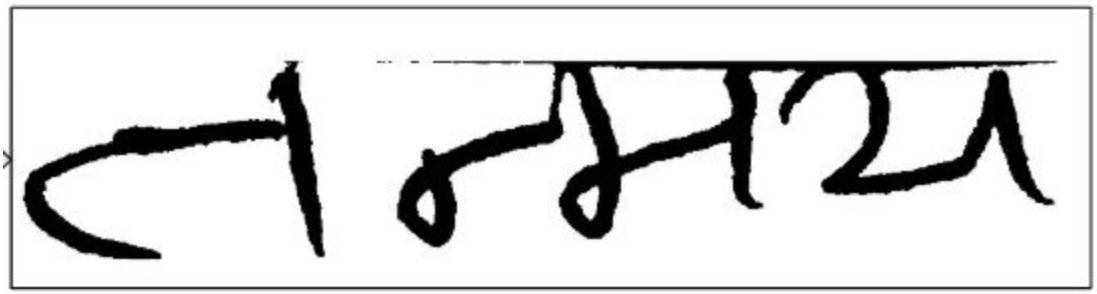
I 

Input Image

Given below is the detailed stepwise description of above approach:

1.1 Preprocessing the input image

1. We first crop the input image and scale it to a proper height (experimentally optimal H=100 pixels).
2. Then we convert it to grayscale and binarize with Otsu's method.
3. We remove its header by selecting 5 rows above and below the maximum horizontal projection.

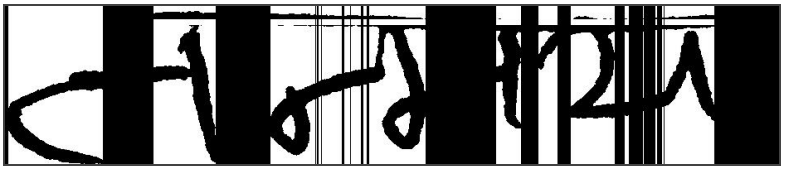


Input image after pre-processing

1.2 Finding the source points for Djikshtra's algorithm

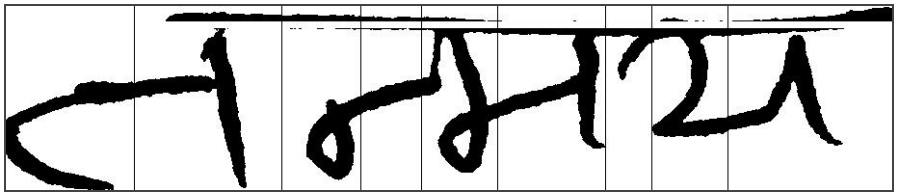
In order to apply Djikshtra's, we first have to find the candidate source points which are potentially segmentation points. We select these points based on vertical projections of their corresponding columns. The points whose vertical projection values would be less than a certain cut-off density would be considered as candidate source points.
In order to determine this cut-off density, we first find out maximum vertical projection (MXVP) and minimum vertical projection (MNVP) and define:
Cut-off density = MNVP + C * (MXVP - MNVP), where C is the cut-off factor.

The experimentally optimal value of C was found to be 0.2. So, in other words, **we are choosing the points with vertical projections in the lowest 20% range as candidate source points.**



Candidate segmentation points

Since these points would be in clusters, we choose the point with the minimum vertical projection from each cluster and finalise them as candidate source points.



Finalised Candidate Segmentation Points

1.3 Defining Cost Function

As stated before, we have to assign proper weights to the edges in order to achieve desirable results. In order to minimise the number of black pixels on the path, we should assign high weight to the edges having at least one of the vertices a black pixel . But there's a trade-off here. **If we assign very high weight to such edges, the resultant path gets tremendously deviated in horizontal direction to avoid black pixels, which is not desirable**.
Considering these trade-offs, the experimentally optimal cost function was determined as :

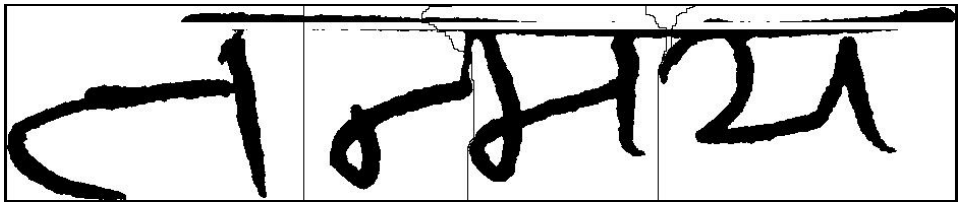
```
#color_of[] strores color of each vertices

def cost_function(edge):
  v1= edge[0] #vertices1
  v2= edge[1] #vertices2
  if color_of[v1]== black or color[v2]== black:
        return 1
  return 0.001
```

So, we assigned a cost of **0.001 to white edges** and a **cost of 1 to other edges** to balance all the factors.

1.4 Applying Djikshtra's Algorithm

Now, that we have the candidate source points and edge weights, we can apply Djikshtra's algorithm from every candidate source point by fixing bottommost row as destination (It means we would stop the algorithm as soon as it reaches any point in the bottommost row and the path traced would segment the characters).
At every point, there would be at most 4 choices for the path to continue. If we apply the standard Djikshtra's algorithm, **in case of equal edge weights, the path would be biased towards any one of the directions. To avoid this, we add a randomised seed to the tuple in Djikshtra's priority queue.**
So, now the paths traced by this algorithm are the potential segments. **To avoid over-segmentation, we reject the paths with cost greater than some cut-off cost K.**
The experimentally optimal value of K was found to be 7.



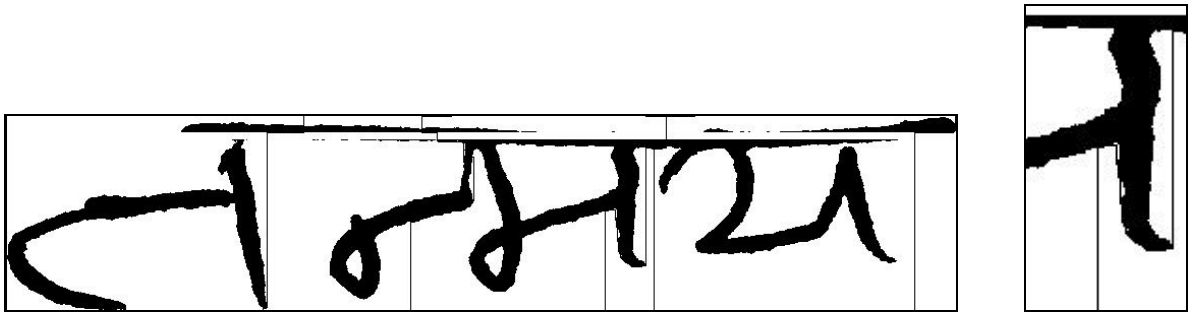
Final Result

Complexity Analysis :

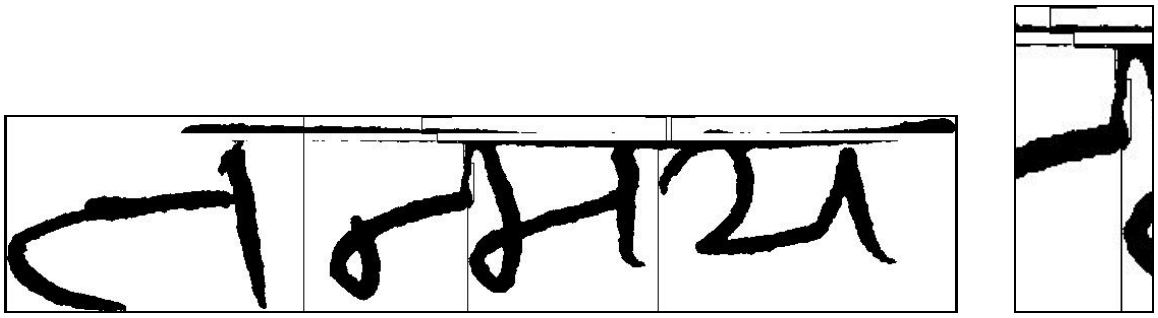
The complexity of this approach is the same as the complexity of Djikshtra's algorithm.
For an image with height H and width W,
Time complexity = $O(H\ W\ log\ (H\ W))$
Space complexity = $O(H\ W)$

The Importance of Cost Function and Randomised Seed :

As we explained previously in the report, cost function and randomised seed were very important for this method to work. To emphasize their importance, we compared the results with and without using them.

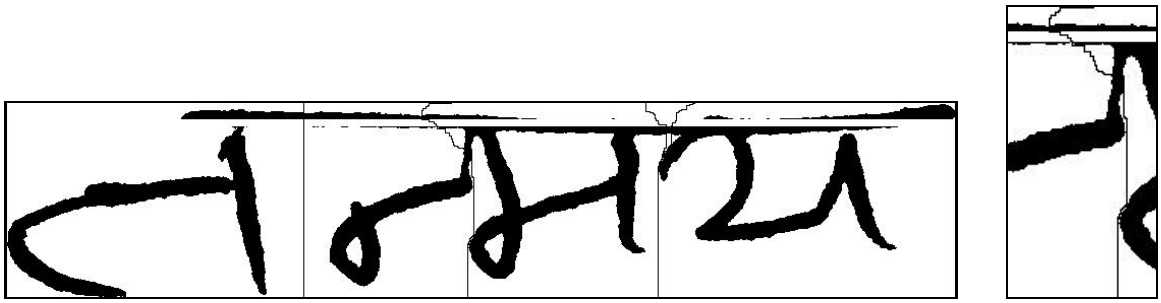


Without randomised seed and with zero weight to white edges



Without randomisation but with minor weight to white edges

With randomisation and with zero weight to white edges

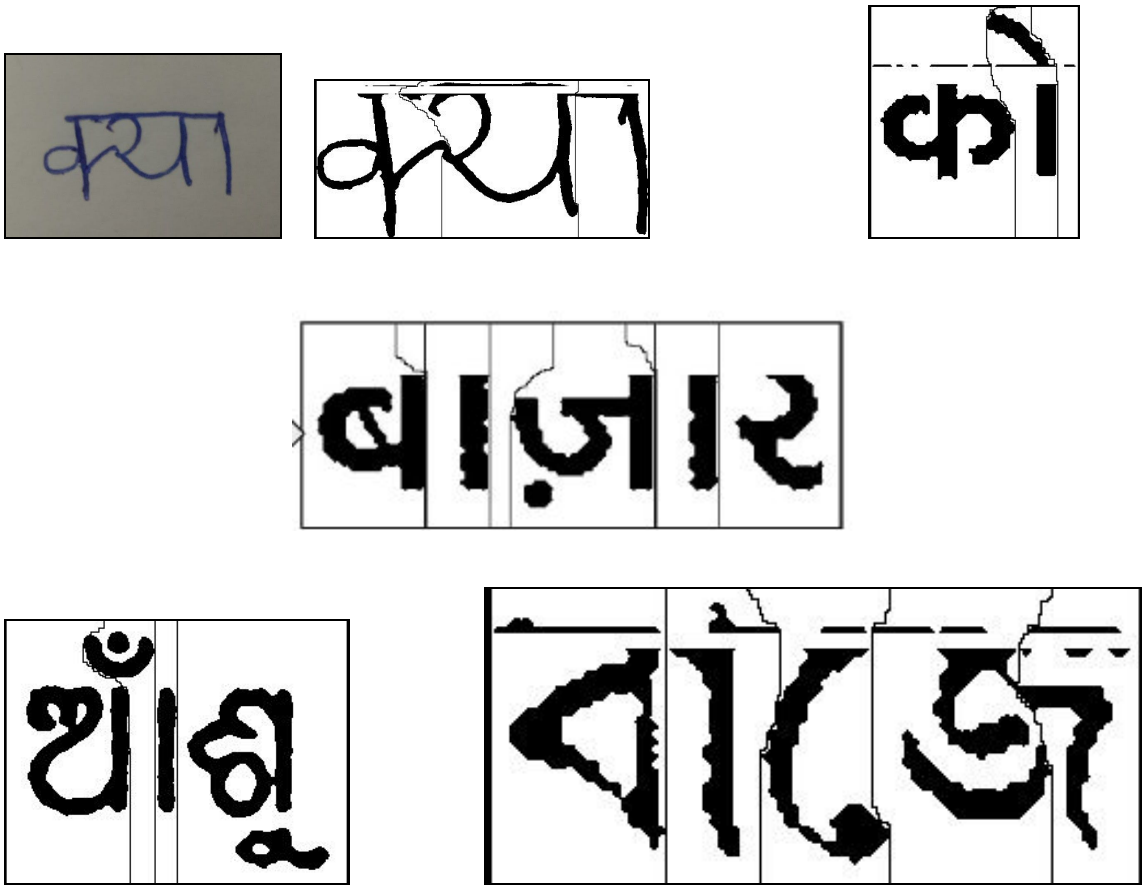


With randomisation and with minor weight to white edges

The last result is the best among all the above results, since it has perfectly smooth segmentation curves with minimum horizontal deviation, without bias towards any particular direction.

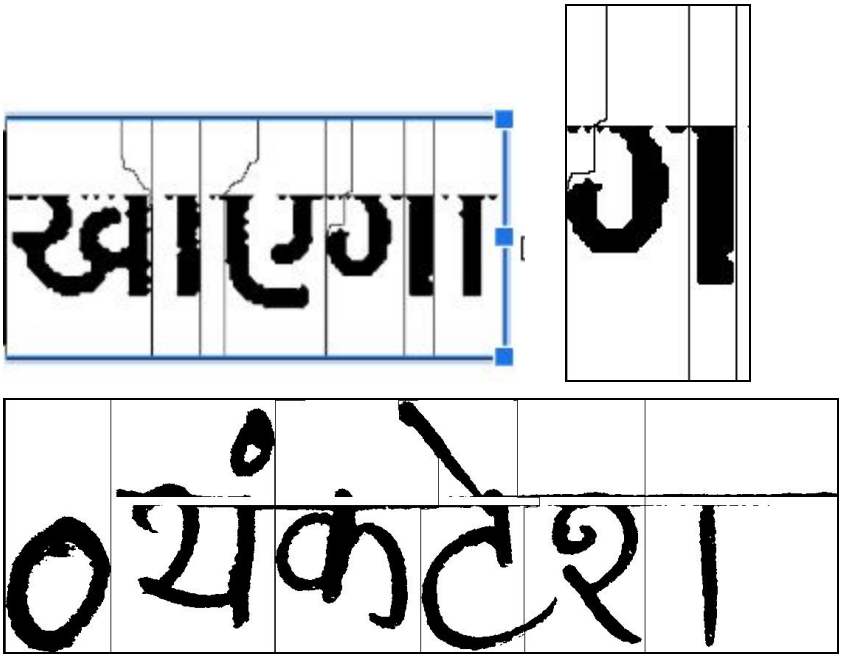
Results and Observations :

This method works on non-slanted Indian handwritings as well as standard fonts of Indian languages. We observed that it segments characters with an accuracy of 80%-85% although it fails on letters like 'ग' and 'श'.



Results

As stated above, our method fails on letters like 'ग' and 'श'. Here are the corresponding results :

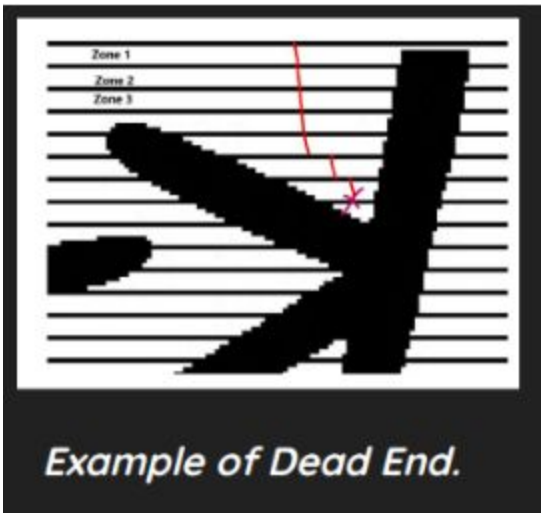


Examples of Failures

2. Dynamic Programming based Zoning Method

A basic zoning method involves creating horizontal zones and then segment the text by joining nearby vertical pixel bars with no black pixels. Although this method is easy to implement, it is very slow in its naive implementation since checking for each next optimal bar takes exponential time.

But since many of the pixel bars ultimately reach the dead end, we could mark the dead end regions with the help of dynamic programming and only trace the pixel bars which never reach the dead end.



Here are some important terms that would be used multiple times hereafter:

1. Pixel bar : A vertical group of pixels of width one pixel over a horizontal zone.
2. Reachable : A pixel bar is said to be reachable if it could possibly be a part of a segment separating characters.(In other words, that pixel bar never reaches a dead end.)

Basic Idea :

Our ultimate objective is to trace the segments of pixel bars which never reach a dead end and thus separate characters from each other. So we have to obtain all the reachable pixel bars.
In order to get all the reachable pixel bars, we use a dynamic programming approach which is explained below in detail.

Implementation Steps:

1. Remove header by a well-known horizontal projection profile method.
2. Divide the image into horizontal zones across its height. The number of optimal zones may vary depending on the image resolution and other factors but for practical purposes, we have used the number of zones = 80 for best results. Then take vertical projections across all zones.
3. Maintain 2 two-dimensional arrays isReachable[number of zones][width], a boolean array which tells if a corresponding pixel bar is reachable and dp[number of zones][width], which stores the number of reachable pixel bars in the next zone, from the corresponding pixel bar.
4. Initially, all the white pixel bars (i.e. vertical projection=0) in the topmost zone are assumed to be reachable and rest non-reachable.
5. If any pixel bar (i, j) is reachable, all the white pixels in the next zone in range (j-offset, j+offset) and white pixels consecutively connected to them are marked reachable. (offset = 10 gives best results.) The number of white pixels in the next zone (i+1) marked reachable this way are stored in dp[i][j].
6. If we get dp[i][j]=0, for some pixel bar (i, j), it implies dead end so we mark it unreachable and recursively backtrack in the upper zone through the arrival path.
7. During backtracking at pixel bar (i, j), we decrement dp[i][j] by 1 since the pixel bar previously marked reachable turned out to be a dead end.
8. After this procedure, we successfully get all the reachable pixel bars. Now we can easily trace paths through those and finalise the ones with the minimum horizontal deviation.

Input image after marking unreachable pixel bars

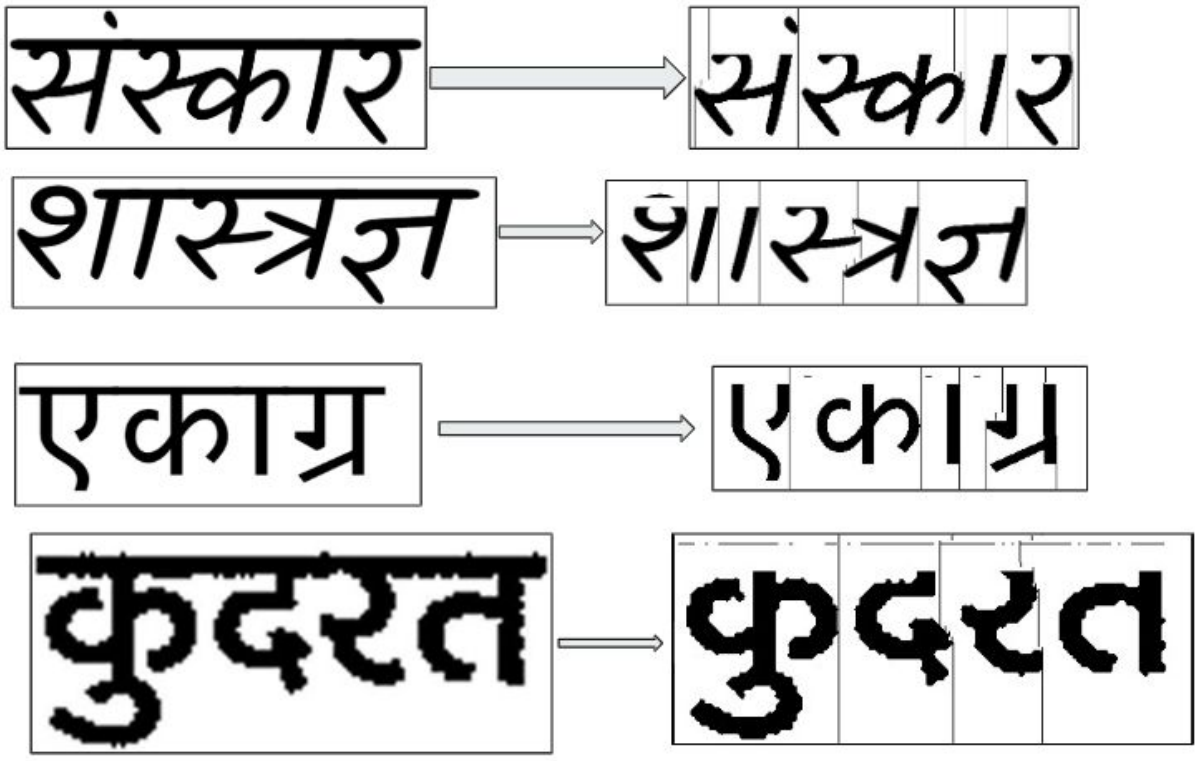
Complexity Analysis :

Let N= Number of horizontal zones, W=Width of image,
H=Height of image, H_z=Height of a zone, K=offset.
In this algorithm, we iterate over the width of all zones and at every pixel bar (i, j), we iterate horizontally on the zone (i + 1) in the offset range to find reachable pixels and compute dp[i][j].
From the view of complexity analysis, the same holds true for backtracking as well.
Let's assume this algorithm requires O(f(H, W)) computations.
Now, in first sight, f(H, W) = N * W * 2 * K =(H / H_z) * W * 2 * K = H * W * (2 * K/H_z).
Let C = 2*K/H_z.
For best results, we had used K=10 and for an image with standard height (in the range 128 to 200), we used N=80, which gives H_z ~ 2.5 so C < 10.
Thus, we have f(H, W) = H * W * C such that C < 10.
So, effectively, this gives a time complexity of O(H * W).

Time Complexity : O(H * W)
Space Complexity : O(H * W)

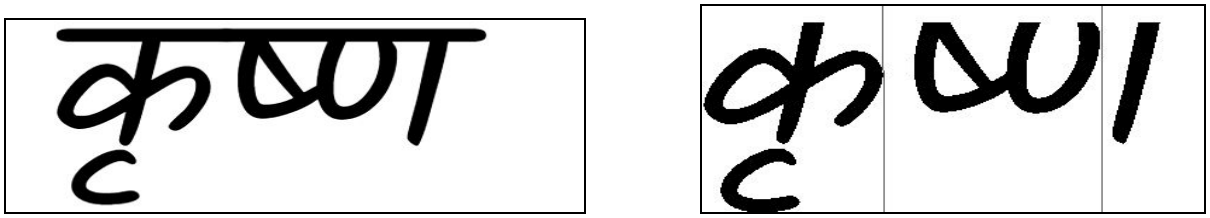
Results and Observations :

This method works on non-slanted to moderately-slanted text and successfully segments single and loosely connected mixed letters like 'स्ज' and 'क्य'.



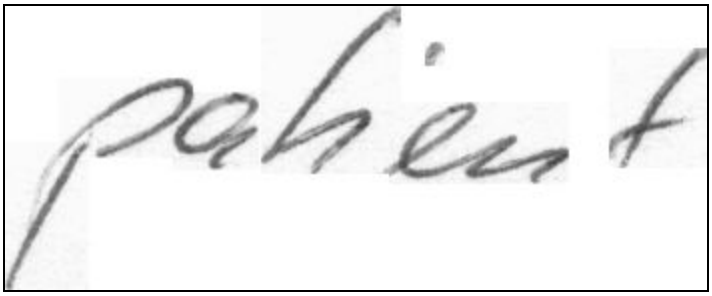
Results

But this method does not work on strongly connected mixed letters since it considers the joining ligatures as dead ends . The accuracy of this method for letter segmentation is approximately 75%.



3. Non-uniform Slant Correction :

For segmentation of cursive English text, we faced the main problem with the slanted text. The traditional method of determining uniform slant angle for the entire word and then rotating it accordingly doesn't work because every letter could have different slant angles. So, we implemented a **non-uniform slant correction algorithm which calculates the slant angles for every column.**

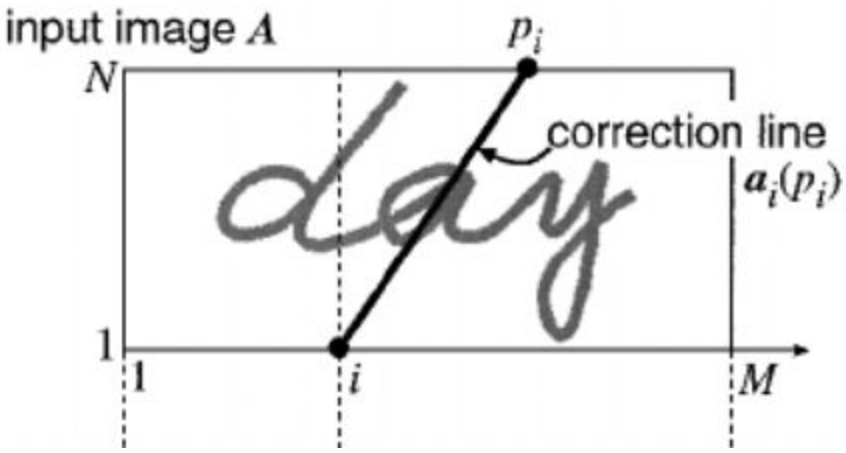


Example of a non-uniformly slanted word

Reduction to a Dynamic Programming Problem:

Let's consider an image of height N and width M.

For every point (1, i) on the bottommost line, we have to find the corresponding slant angle leading to (M, p_i). The slant joining (1, i) to (N, p_i) is called a correction line. Our objective is to find the set of points p_1, p_2, …, p_M ,



Now, we need to fix some upper bound over the slant angle so that we have :

$|p_i - i| <= W$

For best results, we fixed maximum slant angle at $\tan^{-1}(2)$ so W = 2 * N.

For smoothing the sequence of correction lines and avoiding any abrupt changes, it has to be subjected to the continuity constraints so we have : $0 <= p_i - p_{i-1} <= 2$

In order to measure how good any correction line is, we define a function $f(p_i , p_{i-1})$:

$f(p_i , p_{i-1}) = \omega * s_i(p_i) + (1 - \omega) * \gamma_i(p_i , p_{i-1})$.

It grades the correction line based on 2 factors :

1. s_i : This is the length of the longest continuous stroke across the correction line. The idea is that **continuous long slanted strokes show the slant angle more clearly** (just like the long stroke of 'd' in fig. above) than other strokes. For suppressing the effects of short vertical strokes, we set $s_i(p_i)$ to 0 if the length of the longest continuous stroke < N/3.
2. γ_i : This function is used to evaluate the smoothness between the local slant of two consecutive positions i and (i - 1). It is defined as:

$$\gamma_i(p_i \mid p_{i-1}) = \begin{cases} 0 & \text{if } p_i = p_{i-1} + 1 \\ -\Gamma_i(p_i) & \text{otherwise,} \end{cases}$$

where $\Gamma_i(p_i)$ is the number of black pixels on the correction line. The idea is, we **maximise the function $\gamma_i(p_i , p_{i-1})$. if local slants of neighbouring positions are the same** i.e. $p_i = p_{i-1} + 1$ and at ligatures and blanks, its value is very less since there'd be less black pixels on the correction line in that case which in turn allows quick changes in slant angles at ligatures and blanks. (Note : For i=1, we set $\gamma_i(p_i , p_{i-1})$ to 0.)
The experimentally optimal value of ω was found to be 0.75.

This is how we have successfully boiled down a slant correction problem to a maximisation problem. Our goal is to maximise :

$$F(p_1, \ldots, p_i, \ldots, p_M) = \sum_{i=1}^{M} f_i(p_i \mid p_{i-1}),$$

Evidently, the computation of p_i requires the value of p_{i-1} and not any previous values like p_{i-2}, p_{i-3} etc. so the process has a **Markov property.** The maximisation problems with Markov property are solvable by dynamic programming (DP). Thus, the slant correction problem has been reduced to a DP problem.

Solution by DP :

Here is the pseudo-code of the dynamic programming approach used to tackle this maximisation problem :

```
// Initializing DP

for all p1 in range [1-W, 1+W]:
    g[1][p1] = w * s(p1)

// Building DP recursion

for i from 2 to M :
  for all pi in range [i-W, i+W]:
    g[i][pi]=0
    for prev_pi in range[pi-2, pi]:
        g[i][pi] = max(g[i][pi],g[i-1][prev_pi]+f(pi,prev_pi))
    b[pi]=prev_pi which gives maximum value of g[i][pi]

// Backtracking

for pM in range [M-W, M+W]:
    p[M]=argmax(g[M][pM])
for i from M to 2 :
  p[i-1]=b[p[i-1]]
```

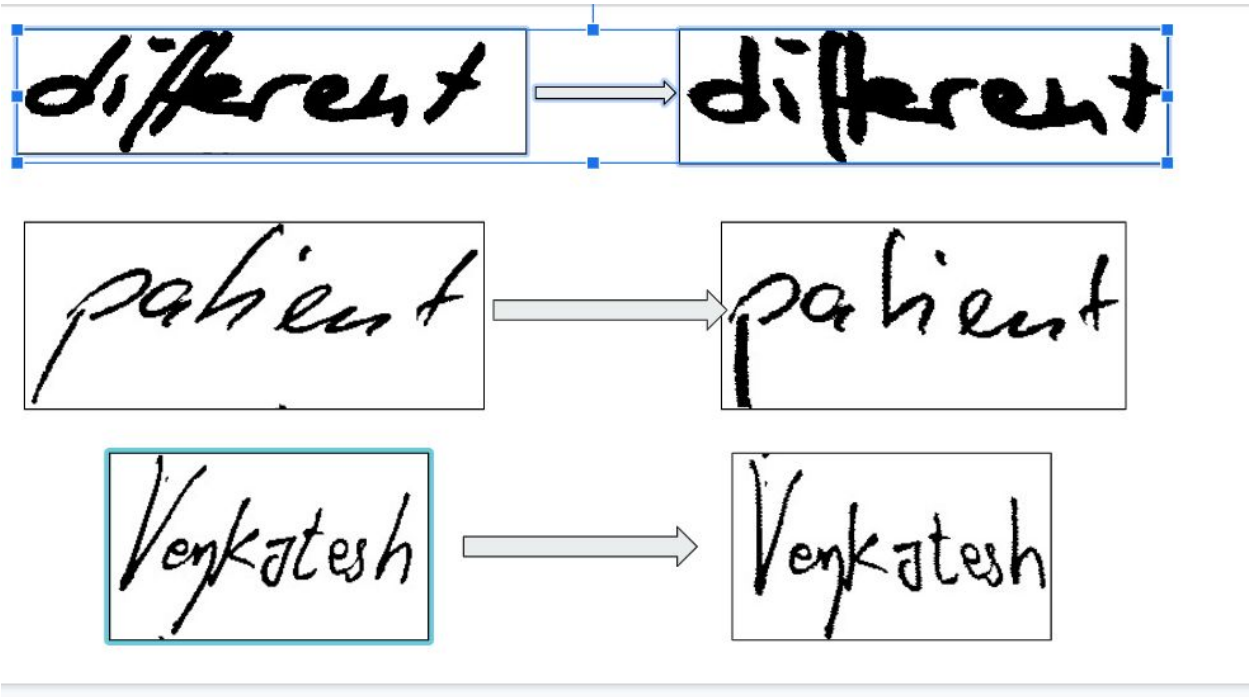
Here, g[i][p_i] is the maximum cumulated value of $f(p_k, p_{k-1})$ till k=i.
This pseudo-code consists of mainly 3 things:

1. Initialising DP : Here, we initialise the first row of dp with the corresponding value of $f(p_i) = \omega * s_i(p_i)$ at i=1.
2. Building recursion: Here, for every i from 1 to M, we compute the value of g[i][p_i] where p_i ranges from [i-W, i+W], with the appropriate recursion given above thus making optimal choice of p_{i-1}, which we store in b[p_i].
3. Backtracking : We find p_M which maximises the value of g[M][p_M] and then backtrack for previous p values taking help from b[p_i].

Complexity Analysis :

The calculation of $f(p_i , p_{i-1})$ requires O(N) computations and DP recursion has O(M*W) computations so evidently, the time complexity of this approach is **O(M * N * W)**, whereas the space complexity is O(M * W).

Results :



This method works perfectly on almost all the slanted words at angles less than $\tan^{-1}(2)$ although the rare failure is found on slanted ‘X’.

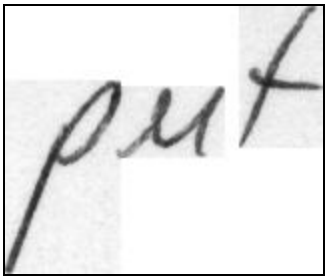
4. Local Minima Heuristic on Skeletonised English Text :

This is a heuristic approach to segment the cursive English handwriting. The basic idea is to find the local minimum columns which could be potential segmentation columns and then finalise the actual segmentation columns by appropriate merging of neighbouring columns.

If we observe closely, there are 2 categories of English characters:

- Closed Characters: These characters contain a loop or semi-loop and are closed. E.g. ‘a’, ‘b’, ‘c’, ‘d’, ‘p’ etc.
- Open Characters: They don’t have a loop or semi-loop. E.g. ‘u’, ‘v’, ‘m’, ‘n’, etc.

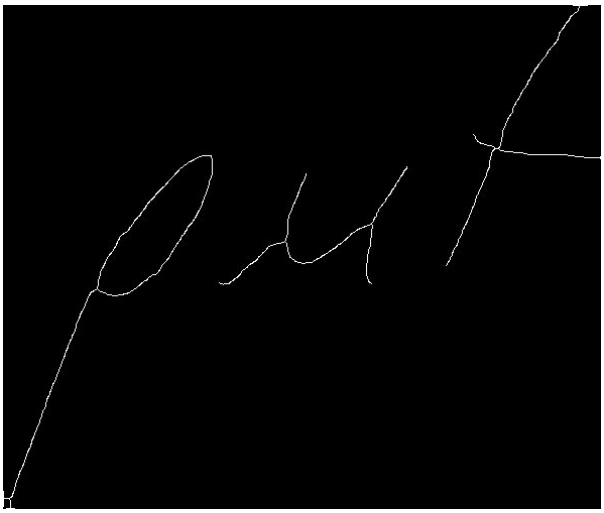
In the case of open characters in cursive handwriting , it’s extremely difficult to differentiate them from ligatures with heuristic methods because of the cursive nature of handwriting. Although we have tried this local minima based heuristic approach which gives decent results.



Input Image

Implementation Steps:

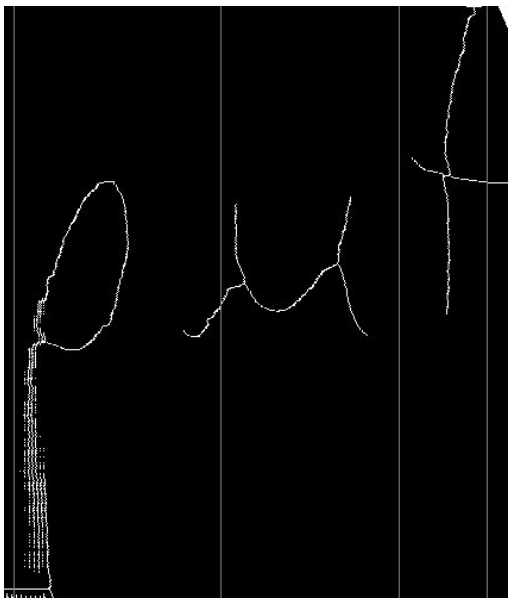
1. **Thinning** : Different handwritings have different thicknesses and strokes so in order to bring a uniformity in the inputs, we perform thinning of image. For this we use the popular Zhang-Suen Thinning Algorithm, which keeps on deleting pixels on contours thick strokes, until a single pixel width skeleton is created.



Thinned Image

2. **Deslanting :** We apply the slant correction algorithm explained before in the report.
3. Find the vertical projection of all columns and list the columns whose vertical projection is 0 (indicating blank space between 2 letters) or 1(indicates ligature between 2 characters or also in letters like ‘h’ and ‘n’). These are our potential segmentation columns (PSCs). Since we have thinned the input image, now ligatures add only 1 white pixel to vertical projection.
4. **Overcoming over-segmentation :** At this point, we face the problem of over-segmentation of characters since the PSCs occur in clusters. In order to overcome this problem, we set a distance threshold D and merge all the PSCs into a single

segmentation column (SC) which are at a distance < D pixels from each other. The resultant SC is situated at the mean position of all the component PSCs. For an image of height 128 pixels, the experimentally optimal value of D was found to be 7 pixels, which is less than the width of the thinnest letters like ‘i’ and ‘l’.



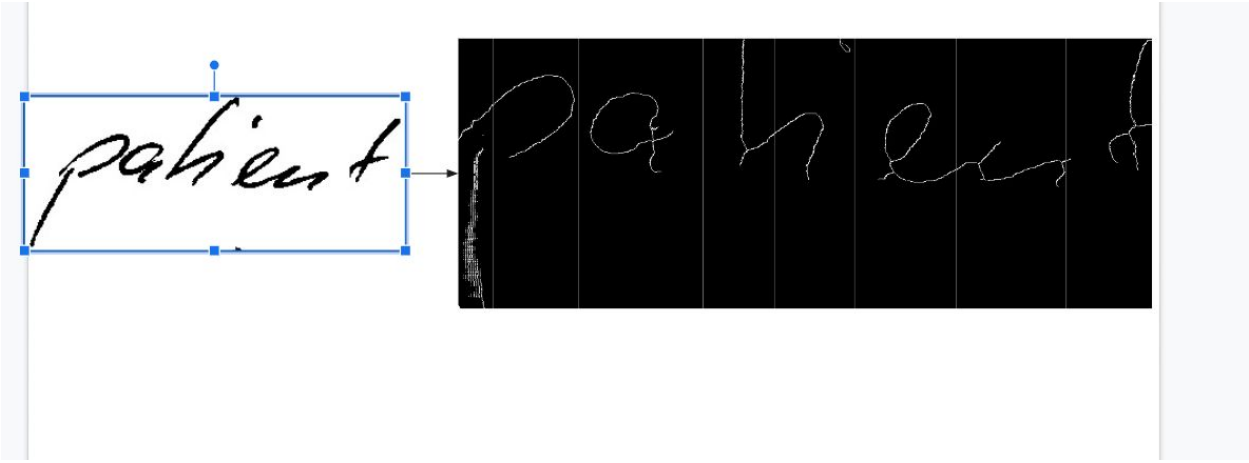
Final Segmented Image

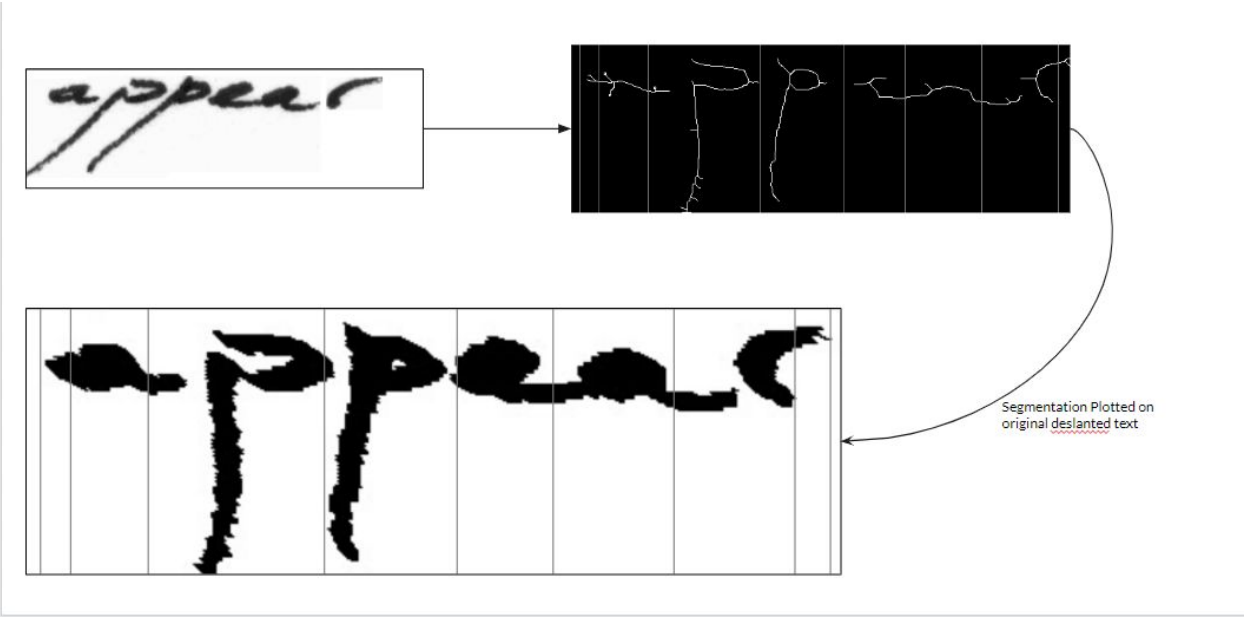
Complexity Analysis :

The most expensive operation among all these operations is deslanting so the complexity of this approach is the same as the complexity of slant correction algorithm which we analyzed in the previous section of this report.

Results and Observations:

This method is viable on slanted cursive and non-cursive English text. It works perfectly on closed characters.





But some inaccuracies were found in segmentation of open characters in cursive handwriting.

It was observed that even after setting an optimal distance threshold to overcome over-segmentation, characters like ‘m’ were found to be over-segmented in some words.
The accuracy of character segmentation of this method was found to be lying around 65%-75%.

Conclusion :

Various heuristic based methods for segmentation of cursive English and Devanagari alike scripts have been proposed. We achieved accuracies comparable to already published methods for heuristic based segmentation.

Towards the end and conclusion of our project we realized that we can do only so much using Heuristic and we have to make use of Machine Learning models in order to achieve better results than this.
Although the above fact stands strong, we still feel we can speed up the segmentation process with a tradeoff of accuracy by using the above proposed, highly time efficient methods. This is especially true for Handwritten Devanagari Scripts as the proposed algorithm for its segmentation is novel , optimised and has high accuracy.
Further, these techniques can be improved by taking clues from specific users' handwriting and adjusting the constants accordingly, although this will make the algorithm offline.

Acknowledgements :

We wish to express our sincere gratitude to Dr. Gaurav Harit for guiding us through the entire project and a special thanks for suggesting the basic idea of zoning method.

References :

1. Otsu's method : Original Paper
2. Dijkstra's Algorithm for Shortest Path : Wikipedia
3. Non-uniform Slant Correction Algorithm : Original Paper
4. Local Minima Heuristic Approach : Original Paper