# Introduction

The performance of popular Optical Character Recognition (OCR) engines such as Tesseract can be greatly improved by pre-processing of the image. When texts exist in an image alongside with other objects, OCR engines often return random characters and carriage returns. To combat this problem ITSegmenter.js detect and isolate text regions in an image.

To access the pixel data, the image is drawn on a canvas element, the image data can then be accessed by the getImageData method. From there an Unsharp Masking is applied to the image data to sharpen it. This improves the results of the next step, feature detection. After sharpening the image, Features from Accelerated Segment Test (FAST) is used to find the corners in the image. Then the corners coordinates stored in an array are used to construct a k-d tree. From there, the Density-based Spatial Clustering of Applications with Noise (DBSCAN) algorithm is applied to corners, which groups them into clusters. Once the clusters are formed, bounding boxes can be drawn based on the maximum and minimum x/y values of each cluster.

# ITSegmenter.js

ITSegmenter.js aims to detect and isolate text regions in an image.

textSegment is the main function to call, it has 1 required parameter and 9 optional parameters.

The simplest usage would be to just specify the image path and use the default value for the optional parameters.

textSegment(imgPath);

Using default optional parameters the original image remains unaffected, and the coordinates of the bounding rectangle for text area will be stored in outputRects asynchronously.

textSegment(imgPath, fThreshhold, eps, minPts, dia, amt, drawRects, splitRects, convertToImage, canvasId)

fThreshhold: Fast Threshhold; Default:100, higher = less corners

eps: Maximum distance between two points to be considered neighbours; Default:15

minPts: Minimum number of points required to form a cluster; Default:5

amt: Scalar of Unsharp Mask

drawRects: Option to draw bounding boxes on the image; Default:0

splitRects: Option to split the text segments into individual images; Default:0

convertToImage: Option to convert canvas to image

canvasId: Option to segment the image on a specific canvas

# Unsharp Masking

Unsharp Masking is an image sharpening technique, which takes in an image and blurs it, invert the blurred image and add it back to the original image.

Blurring an image reduces the high frequency components, thus the inverted blurred image contain only the high frequency component. Adding that back to the original image amplifies the high frequency components of the image.

This is desirable because text on an image are generally high frequency. To apply Unsharp Masking to an image, call the sharpen function with the parameters being: canvas context of the image, width of the canvas, height of the canvas, diameter of the blur, and scalar for the Unsharp Masking effect.

sharpen(ctx, w, h, dia, amt);

## Gaussian Blur

The key process here is the Gaussian blur, which is a convolution operation with the Gaussian kernel. It is very computationally expensive to convolve with a Gaussian kernel.

Thus an approximate Gaussian blur by passing image through 3 box blur is used. It requires 3 parameters: the source buffer, width, height, and diameter of blur.

var blurred = gaussBlur(srcBuff, w, h, dia) ;

First the ideal kernel size of each box blur is calculated, then the image buffer is passed through 3 box blur. The sigma parameter correspond to the diameter of blur and n is the number of passes.

boxesForGauss(sigma, n)

Box blur is a simple averaging of surrounding k-cells for each pixel, where k is the kernel size. Furthermore box blur is separable; convolving the image horizontally then vertically yields the same result as convolving with the entire kernel.

boxBlur(srcBuff, w, h, kernelWidth) ;

# FAST

FAST (Features from Accelerated Segment Test) is a corner detection method.

It examines the 16 surrounding pixels of a pixel to determine whether or not that pixel is a corner. If a set of N contiguous pixels in the 16 pixels are all brighter than the pixel plus a threshold or darker than the pixel minus a threshold then the pixel is classified as a corner.

ITSegmenter.js uses the implementation of FAST from tracking.js. (https://trackingjs.com/)

var corners = Fast.findCorners(pixels, width, height);

# K-D Tree

K-D tree is a binary space partitioning tree, where points are partitioned 2 parts at each node based on alternating dimensions. It is a very efficient data structure for nearest neighbour and range search problems.

This is desirable because the time complexity of DBSCAN is governed by the number of range search it invokes.

## Construction

In this implementation, the K-D tree is constructed by recursively populating each node’s children. The position of the node for partitioning is selected by using Quick Select.

By selecting the median as the position of the node for partitioning, this method results in a balanced tree, which is essential for fast operations on the tree.

To create a K-D tree, initialize a new instance of kdTree with a points array storing the coordinates. Where the points array is in this format: [[x1, y1,], [x2, y2], …].

var tree = new kdTree(points);

tree.nodes;

tree.rootNode;

Once the tree is created the structure can be accessed via tree.nodes or tree.rootNode.

tree.nodes is an array that stores all the individual nodes of the tree and tree.rootNode stores the root node of the tree which contains all other nodes in its decendants.

The node class contain 4 properties: position, leftChild, rightChild, and depth. Position is the coordinate of the node, leftChild and rightChild are children node of this node, and depth is how deep the node is on the tree.

The descendants of a node can be accessed via the getDescendants method. It takes in 2 parameters, the node to be traversed and the descendants array to store the output.

var descendants = [];

getDescendants(node, descendants);

## Range Query

The range query implemented here is a circular range query, it is very similar to K-NN search and also orthogonal range query.

Circular range query returns all points that are within radius r from point [x,y].

var Neighbours = tree.rangeSearch(x, y, r);

rangeSearch traverse down the tree recursively visiting nodes with region that either intersects the query circle or is fully contained in the query circle.

If the node’s region intersects with the query circle, the square distance between node.position and the query point is compared to rSquare. If it is less than rSquare the point is within the query region.

If a node’s region is completely contained in the query circle then the node and all its decendants are within the query region.

# DBSCAN

DBSCAN (Density-based Spatial Clustering of Applications with Noise) is a data clustering algorithm. DBSCAN is very versatile, it can find clusters of arbitrary shape and even internal clusters.   
It is used for clustering group of text together in order to draw bounding boxes around them or crop them out.

It takes in an array of points and the parameter eps and minPts and output clusters of points.

var clusters = DBSCAN(arr, eps, minPts);

Points form a cluster if they are close together. Every point is the input array is visited, and its neighbouring points is found using RangeQuery. Pt is the query point, eps is the maximum distance between two points to be considered neighbours and index is the K-D tree structure.

var Neighbours = RangeQuery(Pt, eps, index);

RangeQuery calls the rangeSearch method from the kdTree class and returns all neighbouring points within radius r of point Pt.