K-means and DB-SCAN in JavaScript

Robin Smit - 4043561

January 13, 2019

Abstract

In this report, we will present an implementation of the clustering algorithms K-means and DB-SCAN using JavaScript as the programming language. The data sets that these algorithms will operate on are images, Each data point (pixel) will have a color value in RGBA-space, as well as a position in $\mathbb{N}$.

Initially, we will use K-means as a method of grouping similar colors together. Next, each of these k clusters will be fed as input to the DB-SCAN algorithm in order to decide whether groups are not only of similar color, but possibly split them using positional density.

Using this technique we will see that features in simple images can easily be grouped into seperate clusters. More complex images, however, this identification of features is not quite as good.

1 Introduction

The goal of this project is to identify and cluster together pixels of an image that represent the same feature. We will attempt to do so by implementing the clustering algorithms K-means and DB-SCAN. Using the pixels of images as data sets K-means will create clusters of similar color values. DB-SCAN will then further seperate these clusters based on pixel position.

This implementation was written in JavaScript.

All code written is included with this report. The most relevant files are Kmeans.js, DBSCAN.js and main.js.

2 Computational complexity

2.1 K-means

The main loop for the k-means algorithm is as follows:

```
// Initialize centroids
...

do {
    // Reset clusters ~ O(k)
    for (var i = 0; i < k; i++)
        clusters[i] = [];

    // Assign points to a cluster ~ O(n * k * d)
    for (var i = 0; i < X.length; i++) {
        var x = this.filter(X[i]); //Selects the color value
           of a pixel ~ O(1)
        var j = getNearestCentroidIndex(x, centroids,
           distance); // ~ O(k * d)
        clusters[j].push(X[i]);
    }

    // Update centroids ~ O(k * (n/k) * d)
    var _centroids = map(clusters, S => mean.call(this, S));

    // Calculate centroid movement ~ O(k * d)
    var movement = map(centroids, function(c, j) {
        return distance(c, _centroids[j]); // ~ O(d)
    });

    centroids = _centroids;
} while (any(movement, d => d !== 0)); // ~ O(k)
```

In short, it will continually iterate the following steps, until the centroids no longer move w.r.t. their last positions.

1. Assign all points to the cluster of it's nearest centroid
2. Recalculate all centroids as the mean of their cluster

Important factors in calculating the complexity are the number of pixels in a data set ($n = |X| = width * height$), as well as the parameter k and the number of iterations i required to converge to a clustering of the image. The dimension of the RGBA color space that our pixels occupy is the constant $d = 4$.

Thus, the time complexity for this implementation is $O(i * n * k * d)$.

2.2 DB-SCAN

The following is the main loop loop of the DB-SCAN algorithm:

```
for (var i = 0, j = 0; i < this.X.length; i++) {
    var p = this.X[i];
    if (p.label !== undefined)
        continue;

    var neighbours = getNeighbours.call(this, this.filter(p),
        eps, this.distance);
    if (neighbours.length < minPts)
        p.type = Labels.NOISE;
    else {
        p.type  = Labels.CORE;
        p.label = j++;
        for (var k = 0; k < neighbours.length; k++) {
            var q = neighbours[k];
            if (q.type === Labels.NOISE) {
                q.type  = Labels.EDGE;
                q.label = p.label;
            }
            else if (q.type === undefined) {
                q.label = p.label;
                var N = getNeighbours.call(this, this.filter(
                   q), eps, this.distance);
                if (N.length >= minPts) {
                    for (var n = 0; n < N.length; n++)
                        if (!neighbours.includes(N[n]))
                            neighbours.push(N[n]);
                }
            }
        }
    }
}
```

In this case, the data set $X = S_j$ represents one of the k clusters that are produced in the previous k-means step. Let n be the size of the original image and $m_j = |S_j|, (j = 1..k)$ the sizes of its color-based clusters. Every data point is now a 2D pixel position ($d_{pos} = 2$).

Note that, even though the code snippet displays nested loops, the getNeighbours function is called only once for each point in X.

Since the getNeighbours functions plays such a prominent role in this algorithm, it's worth the effort to create a version that is more efficient than calculating the distance from a given point to *every* other point in X.

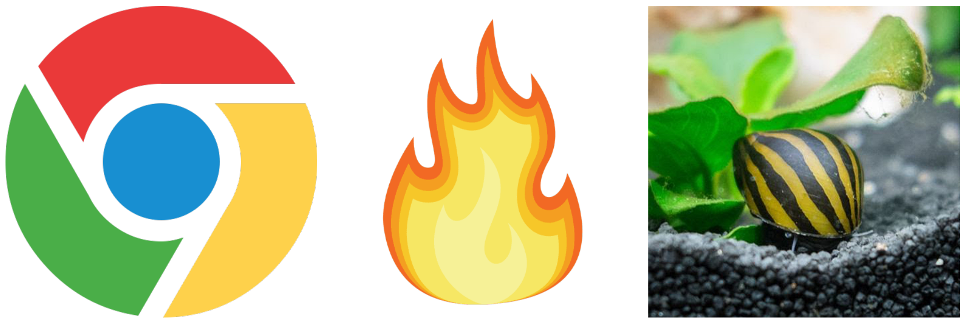
One way to do this, is using the the knowledge that we are dealing with pixel positions. The following implementation starts considering points close to the given position (fx) and expands outward. As long as $\epsilon = r$ is relatively small (i.e. does not cover large portions of X for many pixels), this gives a decent improvement over naively iterating over all elements in X.

```
/* Because we know we are dealing with pixel positions,
 * we only have to check distances within a given radius
 * from the center.
 */
function getNeighbours(fx, r, distance) {
    var neighbours = [],
        check_positions = [];
    for (var dy = 1; dy <= r; dy++) {
        for (var dx = 0; distance(fx, [fx[0] + dx, fx[1] + dy
            ]) <= r; dx++)
            check_positions.push(
                // 4 rotational symmetries
                [fx[0] + dx, fx[1] + dy],
                [fx[0] - dy, fx[1] + dx],
                [fx[0] - dx, fx[1] - dy],
                [fx[0] + dy, fx[1] - dx]
            );
    }

    forEach(check_positions, function([x, y]) {
        var index = x + width * y;
        if (0 <= x && x < width &&
            0 <= y && y < height && mask[index]) //mask was
                previously computed in O(m_j) time
              neighbours.push(X[index]);
    });

    return neighbours;
}
```

The cost for computing the variable mask is $O(m_j)$, therefore the time complexity for getNeighbours is $O(m_j * \epsilon^2 * d)$. The total time complexity for running DB-SCAN on cluster $j \in \{1, ..., k\}$ is $O(m_j^2 * \epsilon^2 * d)$

3 Results

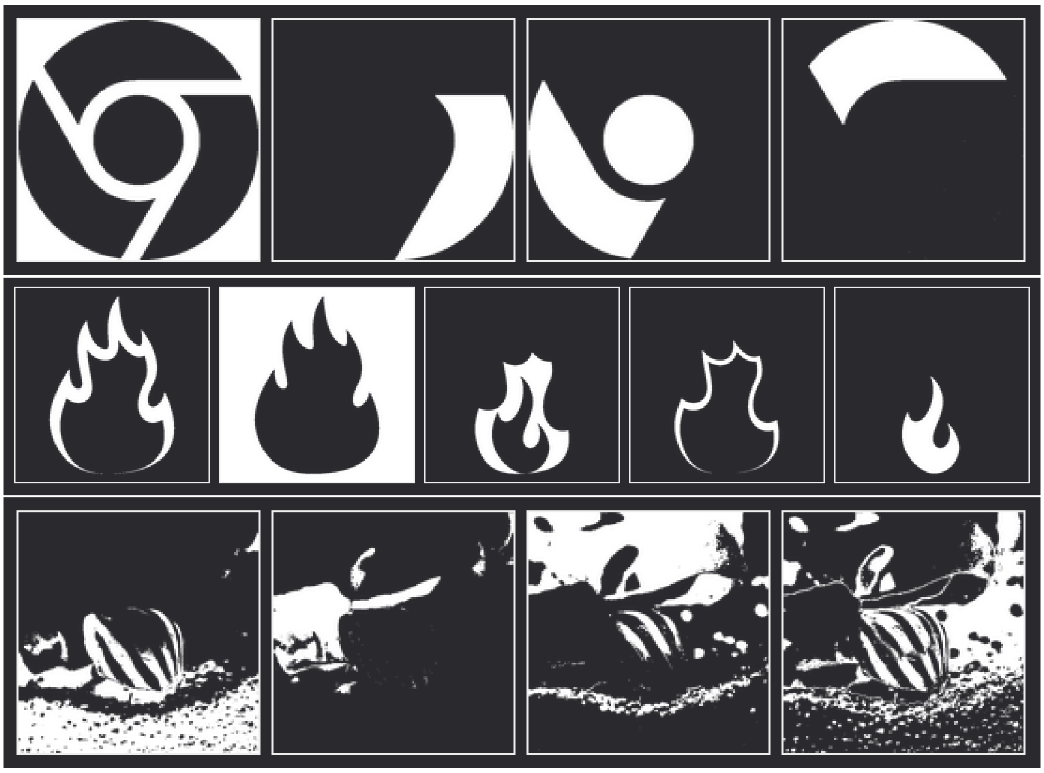
We will run the algorithms for these examples:



The pixels in the clusters that result from the algorithm will be displayed white against a dark background. This way, pixels with full transparency are visible as well.

3.1 K-means

Running K-means for $k = 4$, $k = 5$ and $k = 4$ respectively, gives us the initial clusters below. Note that the k-means algorithm considers the color values of pixels in the data set only.

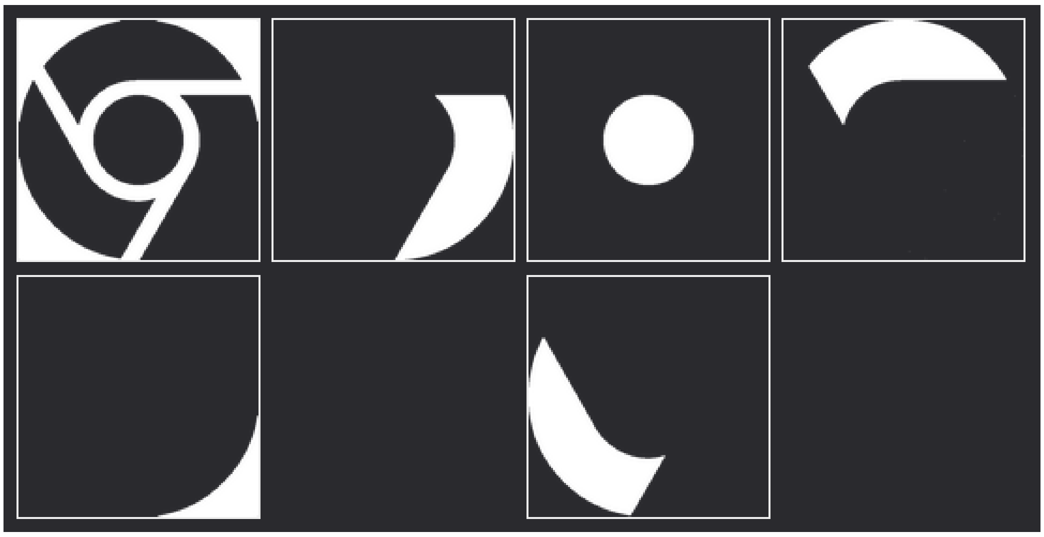


3.2 DB-SCAN

Example 1

In the first example, note that $k = 4$ is just a bit too low when accounting for the transparent background color of the icon. After applying k-means, the green and blue areas of the images are clustered together. While increasing k to 5 does usually give the 'correct' (intuitive) segmentation, this is not a guarantee.

These are the 6 final clusters after running DB-SCAN ($\epsilon = 1, minPts = 4$) on the first image:

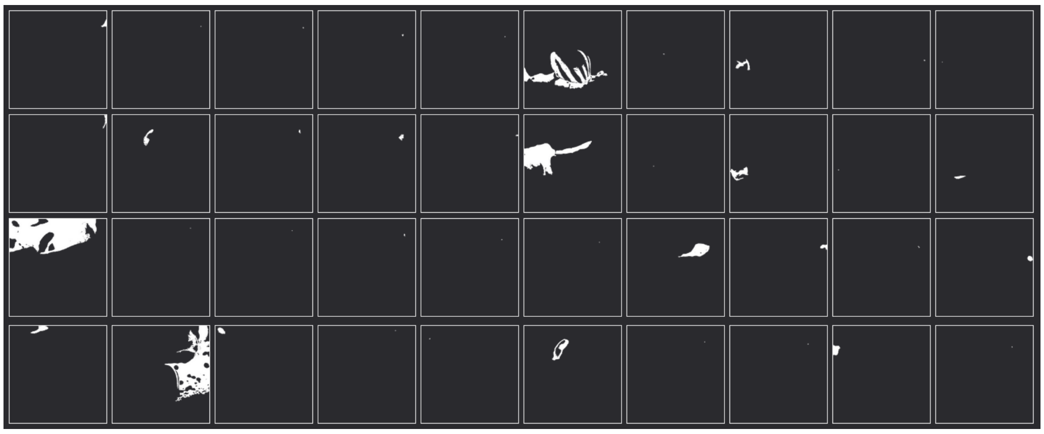


Example 2

In the flame icon example, the colors are much more similar to each other. Still k-means does a good job identifying the main clusters. Running DB-SCAN with parameters $\epsilon = 1, minPts = 4$ on each of these clusters labels a few disconnected pixels at the 'bottom' of the clusters as noise, but the overall result remains largely the same.

Example 3

From the 4 clusters above, DB-SCAN produces the 40 clusters below, based on pixel proximity to others ($\epsilon = 2, \text{minPts} = 8, \text{distance} = \textit{manhattan}$). We now have quite some clusters that contain only a few pixels, but note that the larger clusters do (roughly) represent main features in the original image.



4 Conclusion & Future direction

4.1 Expectations vs. results

Expectations were that using this combination of clustering algorithms would be most effective in clustering together features of 'simple' images. That is, images with few or no gradients and color differences that appear quite obvious to the human eye (spaced apart in RGBA). As can be seen from Example 1 in the Results section, this is indeed the case: even when similar colors of seperate features are initially grouped together, the second step of this method (DB-SCAN) will seperate them.

For more complex images, however, the results could definitely be improved. Later on in this section, we will give some ideas for further development of this technique of using K-means and DB-SCAN in conjunction for image feature detection.

4.2 Limitations

One limitation of the current implementation is the fact that it runs in a web-browser. There are undoubtedly better alternatives when it comes to efficiency. For instance, our JavaScript-implementation typically takes approximately 60 seconds to do repeated clusterings of just a 512×512-image.

4.3 Future direction

There are multiple things that could be done to improve the efficiency of the implementation or aid in reaching better results for identifying the main features of images. Some future steps that may be done are:

1. Experiment with the order of these algorithms. Would executing DB-SCAN before K-means improve the identification of features in an image?
2. Automate the selection of parameters (k for K-means, ϵ and minPts for DB-SCAN).
3. For DB-SCAN on images, implement a spatial indexing structure. This may even further improve the performance of the getNeighbours method.