# Image Labeling

Néstor Campos Gestal - 1494758

Pedro García Suárez - 1494603

Samer Bujana Escalona - 1490885

# 1 Introduction

# INTRODUCTION: GOALS

Labeling clothes:

- Color (K-means)
- Shape (K-NN)

# Introduction (II): Algorithms

**K-MEANS**

Not supervised.

Find patterns (colors).

Iterative:

- Initialize Centroids.
- Get Distances.
- Classes.
- New Centroids.
- Stops when solution is found.

Heuristics: *within class distance, fisher coefficient...*

Color assignment.

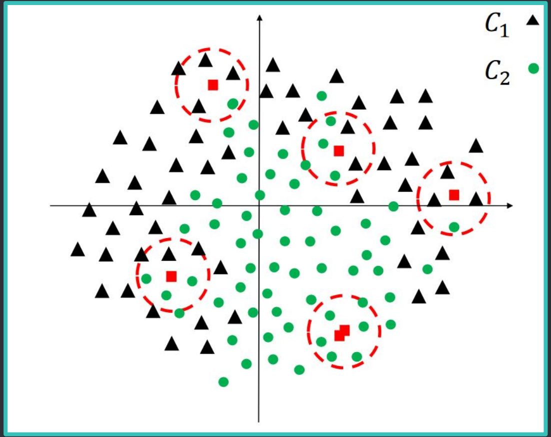
**K-NN**

Supervised.

Train and Test data comparison.

Nearest neighbors labels.

Class finding from labels.



## 2 Qualitative Analysis

# Qualitative analysis

We dispose of 3 functions to analyze the quality of our algorithm, all of them are based on the same functionality and they share part of the name: retrieval.

- Retrieval_by_color.
- Retrieval_by_shape.
- Retrieval_combined.

(blue, jeans)

```
def Retrieval_combined(list_img, list_img_color, list_img_shape, color_query, shape_query):
    returner = []

    for index, (img_color, img_shape) in enumerate(zip(list_img_color, list_img_shape)):
        if color_query in img_color and shape_query in img_shape:
            returner.append(list_img[index])

    return returner
```

# 3 Quantitative Analysis

# Quantitative Analysis

- Analyze how well our algorithm works quantitatively.
- Quantitative → numbers
- Functions:

```
Kmean_statistics(obj_kmeans, Kmax)
Get_shape_accuracy(knn_labels, test_class_labels)
Get_color_accuracy(kmeans_colors,test_color_labels)
```

```
Kmean_statistics(obj_kmeans, Kmax)
```

- obj_kmeans: object of Kmeans class
- Kmax: max value of K

- It give us information about the Kmeans algorithm

## Kmean_statistics(obj_kmeans, Kmax)

- obj_kmeans: object of Kmeans class
- Kmax: max value of K

Main:

```
elem_kmeans = []
elem_colors_kmeans = []

for img in test_imgs:
    elem_kmeans.append(km.KMeans(img))
    elem_kmeans[-1].find_bestK_improved(4, 'WCD')
    elem_colors_kmeans.append(km.get_colors(elem_kmeans[-1].centroids))

Kmean_statistics(km.KMeans(test_imgs[0]), 6)
```

Function:

```
def Kmean_statistics(obj_kmeans, Kmax):
    for k in range(2, Kmax):
        obj_kmeans.K = k
        obj_kmeans.fit()
        obj_kmeans.heuristic = obj_kmeans.heuristic_kmeans('WCD')
        visualize_k_means(obj_kmeans, [80, 60, 3])
```

# Kmean_statistics(obj_kmeans, Kmax)

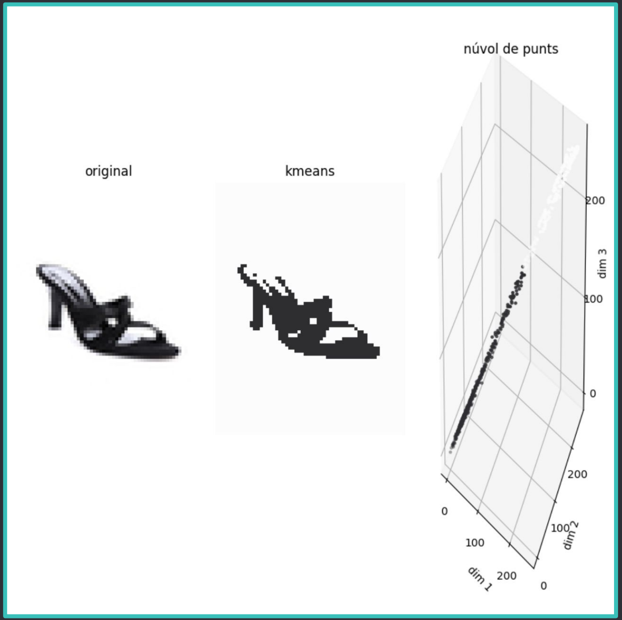
```python
elem_kmeans = []
elem_colors_kmeans = []

for img in test_imgs:
    elem_kmeans.append(km.KMeans(img))
    elem_kmeans[-1].find_bestK_improved(4, 'WCD')
    elem_colors_kmeans.append(km.get_colors(elem_kmeans[-1].centroids))

Kmean_statistics(km.KMeans(test_imgs[0]), 6)
```

Main

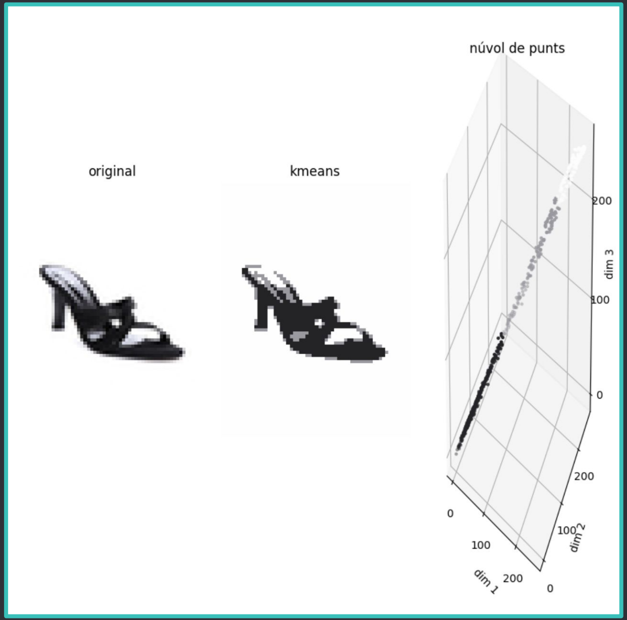
```python
def Kmean_statistics(obj_kmeans, Kmax):
    for k in range(2, Kmax):
        obj_kmeans.K = k
        obj_kmeans.fit()
        obj_kmeans.heuristic = obj_kmeans.heuristic_kmeans('WCD')
        visualize_k_means(obj_kmeans, [80, 60, 3])
```

Function

# Kmean_statistics(obj_kmeans, Kmax)

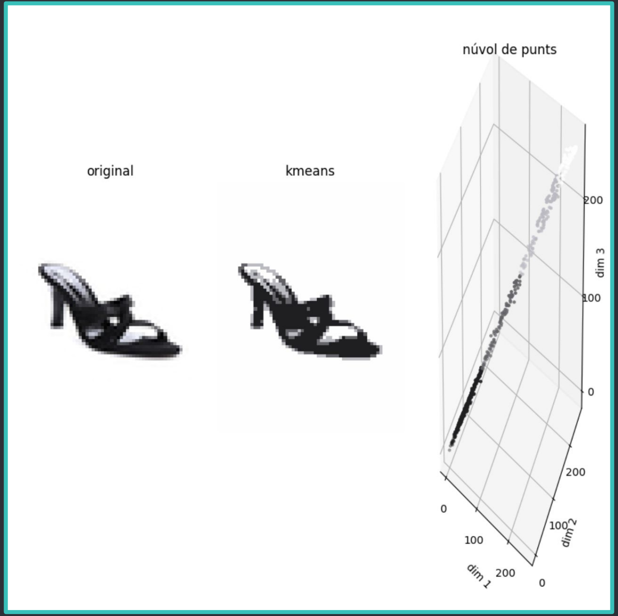


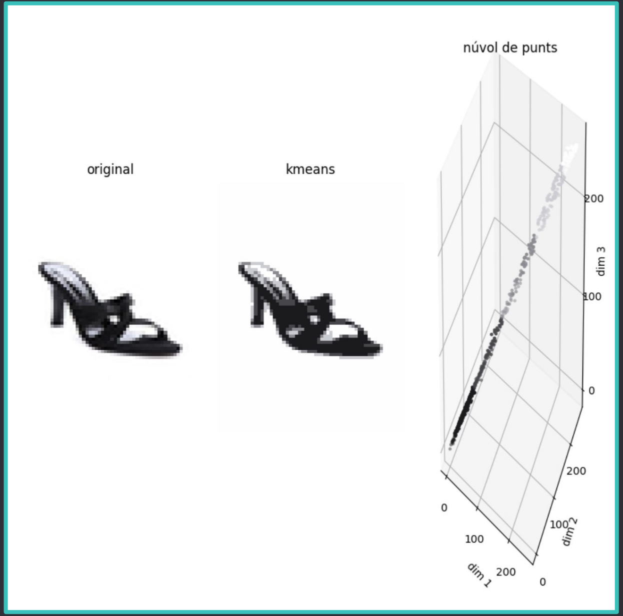

K: 2
WCD: 678.38832
Iterations: 5
Time: 0.41697s




K: 3
WCD: 257.54503
Iterations: 17
Time: 0.3942368s

## Kmean_statistics(obj_kmeans, Kmax)

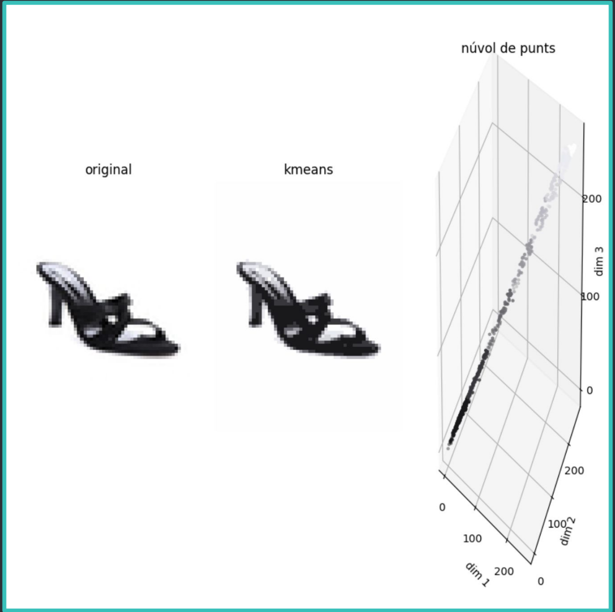


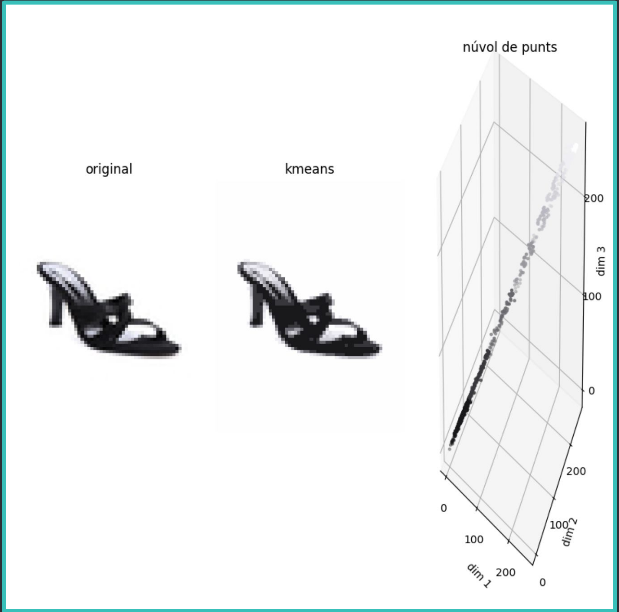




K: 4
WCD: 138.692
Iterations: 43
Time: 0.41008s

K: 5
WCD: 85.0973134
Iterations: 78
Time: 0.3970969s

# Kmean_statistics(obj_kmeans, Kmax)




K: 8
WCD: 42.09232
Iterations: 211
Time: 0.42467s

K: 9
WCD: 40.78192
Iterations: 272
Time: 0.3970969s

↑K ↑Colors ↑Iterations

↓WCD

## `Get_shape_accuracy(elem_knn,test_class_labels)`

- elem_knn: object of KNN class
- test_class_labels: Ground-Truth

- It measures how accurate is the KNN algorithm
- It gives us a percentage of how similar our KNN result is to the Ground-Truth results.

## Get_shape_accuracy(elem_knn,test_class_labels)

- elem_knn: object of KNN class
- test_class_labels: Ground-Truth

Main:

```
elem_knn = KNN.KNN(train_imgs, train_class_labels)
porcentaje_shape = Get_shape_accuracy(elem_knn.predict(test_imgs, 4), test_class_labels)
print(porcentaje_shape)
```

Function:

```
def Get_shape_accuracy(knn_labels, test_class_labels):
    return (np.sum(np.array(sorted(knn_labels)) == np.array(sorted(test_class_labels))) / len(test_class_labels)) * 100
```

## `Get_color_accuracy(elem_colors,test_color_labels)`

- elem_colors: object of Kmeans class
- test_color_labels: Ground-Truth

- It measures how accurate is the Kmeans algorithm
- It gives us a percentage of the correct labels of the Kmeans.

## Get_color_accuracy(elem_colors,test_color_labels)

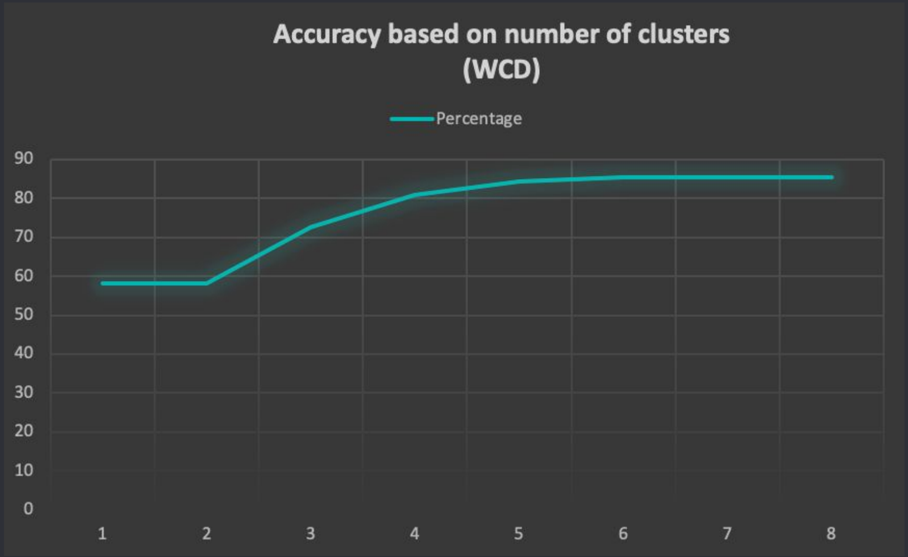
- elem_colors: object of Kmeans class
- test_color_labels: Ground-Truth

Function:

```
def Get_color_accuracy(kmeans_colors, test_color_labels):
    count = 0

    for test, ground_truth in zip((kmeans_colors), (test_color_labels)):
        test = set(test)
        if test == ground_truth:
            count += 1
        else:
            count_aux = 0
            for test_color in test:
                if test_color in ground_truth:
                    count_aux += 1

            count += count_aux / len(ground_truth)

    return (count / len(test_color_labels)) * 100
```

## Get_color_accuracy(elem_colors,test_color_labels)

- elem_colors: object of Kmeans class
- test_color_labels: Ground-Truth



# 4 Improvements and modifications

# Different Heuristics for BestK

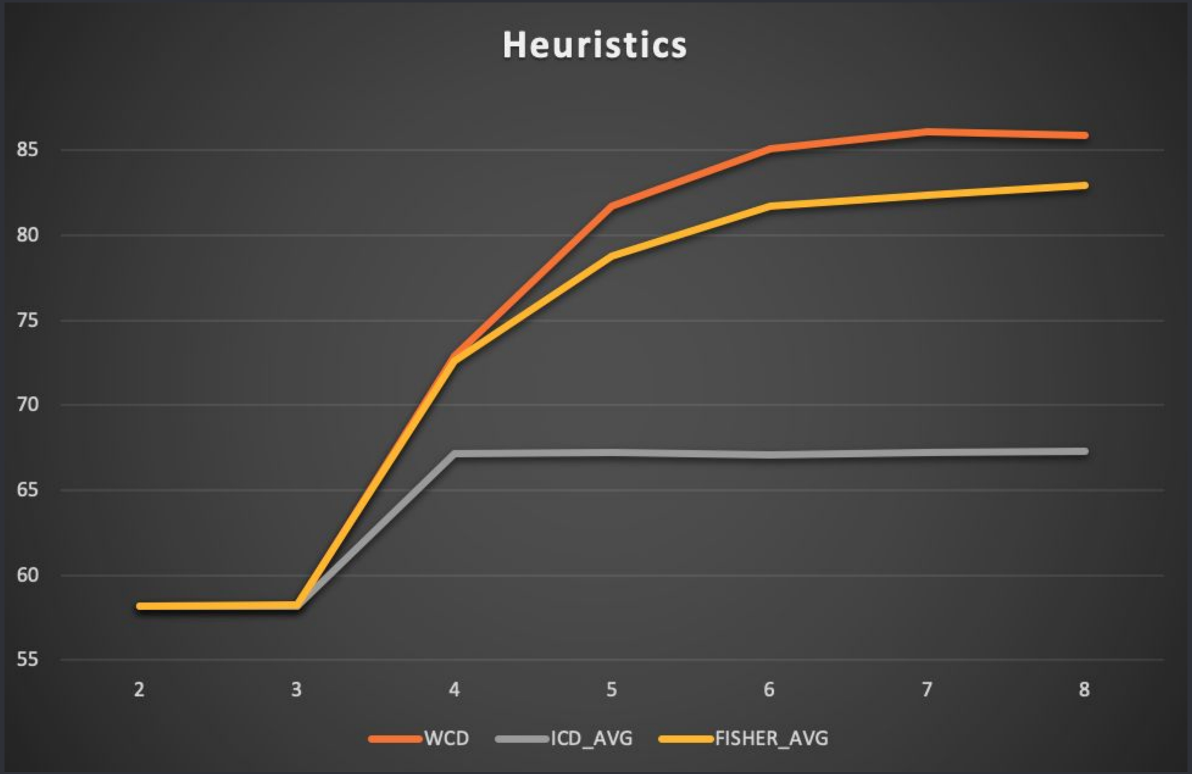
## heuristic_kmeans(self,type)

- To use the different heuristics in an easier way

Function:

```python
def heuristic_kmeans(self, type):

    if type == 'WCD':  # Within Class Distance
        return self.withinClassDistance()

    elif type == 'ICD':  # Inter Class Distance (MINIMUM VALUE)
        dist_centroids = distance(self.centroids, self.centroids)
        return np.min(dist_centroids[np.nonzero(dist_centroids)])

    elif type == 'ICD_AVG':  # Inter Class Distance (AVERAGE VALUE)
        dist_centroids = distance(self.centroids, self.centroids)
        return np.mean(dist_centroids[np.nonzero(dist_centroids)])

    elif type == 'FISHER':  # WCD/ICD
        dist_centroids = distance(self.centroids, self.centroids)
        return self.withinClassDistance() / np.min(dist_centroids[np.nonzero(dist_centroids)])

    elif type == 'FISHER_AVG':  # WCD/ICD_AVG
        dist_centroids = distance(self.centroids, self.centroids)
        return self.withinClassDistance() / np.mean(dist_centroids[np.nonzero(dist_centroids)])
```

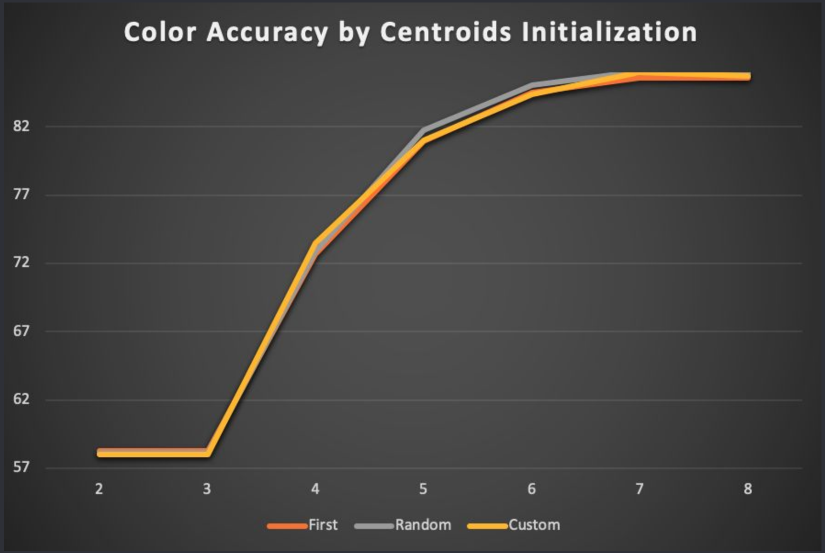
# Comparison between all heuristic methods



## Initializations of Kmeans

# Initial Centroids

- Random comes out as best of the three



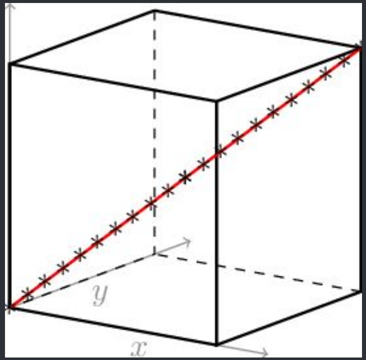

| K | First | Random | Custom |
|---|---|---|---|
| 2 | 58,27262 | 58,16 | 58,018 |
| 3 | 58,27262 | 58,16 | 58,018 |
| 4 | 72,6479 | 72,86 | 73,4704 |
| 5 | 80,942 | 81,74 | 80,9949 |
| 6 | 84,501 | 85,08 | 84,412 |
| 7 | 85,6071 | 86,07 | 85,9988 |
| 8 | 85,58754 | 85,91 | 85,705053 |

# Initial Centroids (II)

- Custom first idea:

  Hypercube diagonal:

  - Find K equidistant points in the matrix diagonal
  - Translate to vector X with 4800 positions



# New custom

Modification of the random initialization.

```
elif self.options['km_init'].lower() == 'custom':
    awesomeness_factor = 10
    centroid_selector = []

    while n_k < (self.K * awesomeness_factor):
        np.random.seed()
        n_aleatorio = np.random.randint(0, len(self.X))

        if self.X[n_aleatorio].tolist() not in lista_puntos:
            centroid_selector.append(self.X[n_aleatorio])
            lista_puntos.append(self.X[n_aleatorio].tolist())
            n_k += 1

    centroid_selector = np.array(centroid_selector)
    dist_centroids = distance(centroid_selector, centroid_selector)
    dist_centroids = np.array([np.mean(dist_centroids[i][np.nonzero(dist_centroids[i])])
                               for i in range(len(dist_centroids))])
    max_args = np.argsort(dist_centroids)[-self.K:]
    self.centroids = centroid_selector[max_args]
```

# Ideas of improvement of the new custom

The previous function takes the centroids with more distance between them, we can translate that to a larger interclass distance.

The problem is that those centroids may be too similar.

***How to improve this?***

We could change the centroids we take, instead of those with the largest mean, we could take the ones in the center or the smallest.

Analyzing the colors of those random centroids and take their supposed RGB values and take those are more different colors. We can do that using the function given to us in the utils.py file.

For example we don’t want two centroids with these values:

Centroid = [255, 255, 255]

Centroid_two = [250, 252, 245]

```python
elif self.options['km_init'].lower() == 'hyper':
    dict_colors = {'Red': [255, 0, 0], 'Orange': [255, 127, 80], 'Brown': [255, 228, 196],
                   'Yellow': [255, 255, 0], 'Green': [0, 128, 0], 'Blue': [0, 0, 200],
                   'Purple': [255, 0, 0], 'Pink': [255, 20, 147], 'Black': [0, 0, 0],
                   'Grey': [192, 192, 192], 'White': [255, 255, 255]}
    centroid_selector = []

    for fila in self.X:
        centroid = [utils.colors[element] for element in
                    np.argmax(utils.get_color_prob(np.array([fila])), axis=1)]
        if n_k < self.K and centroid not in lista_puntos:
            centroid_selector.append(dict_colors[centroid[0]])
            lista_puntos.append(centroid)
            n_k += 1

        if n_k >= self.K:
            break

    self.centroids = np.array(centroid_selector)
```

# Ideas of improvement of the new custom

This new version is called hyper, trying to be the hypercube’s diagonal. Is a fusion of ideas between the hypercube and the new custom that is the modified random initialization.

It iterates all over the image’s pixels searching for the number of different colors in it.

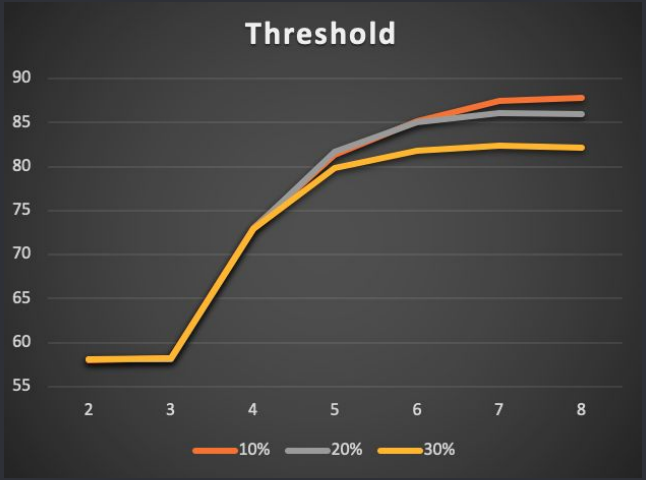
***Problems***

We have problems initializing the centroids because the shape of some matrix, and we are looking how to solve it.

# Find_BestK Different Threshold Results

# Threshold

Accuracy after changing the threshold for WCD with Kmax =7:




| K | 10% | 20% | 30% |
|---|---|---|---|
| 2 | 58,018 | 58,16 | 58,1355 |
| 3 | 58,273 | 58,16 | 58,27262 |
| 4 | 73,0592 | 72,86 | 72,90247 |
| 5 | 81,28476 | 81,74 | 79,85899 |
| 6 | 85,14689 | 85,08 | 81,80768 |
| 7 | 87,418723 | 86,07 | 82,34626 |
| 8 | 87,85938 | 85,91 | 82,18958 |

# 5 Conclusions

# Conclusions

Learning pattern recognition with KMEANS and KNN

Working with numpy in python to improve efficiency

Applications:

- Computer Vision
  - Autopilot in vehicles
  - Image reverse search
  - Text recognition in images