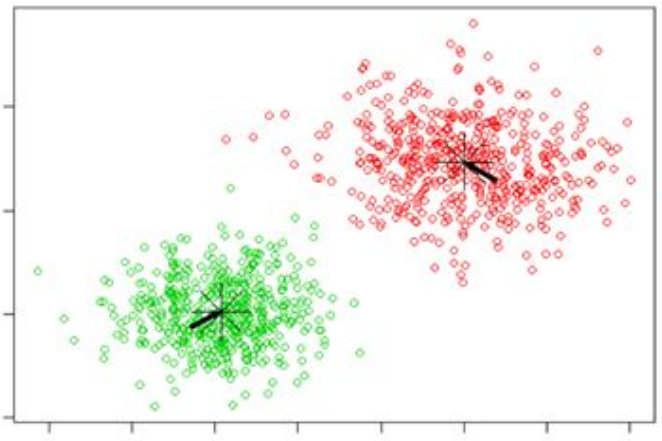
LAB PROJECT 2: IMAGE LABELING

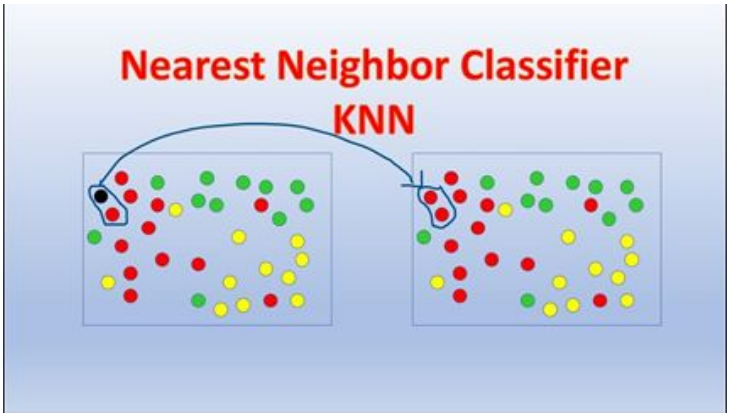
Intel·ligència Artificial

2019-2020

Departament de Ciències de la Computació

Universitat Autònoma de Barcelona





Samer Bujana Escalona – 1490885 Néstor Campos Gestal – 1494758 Pedro García Suárez – 1494603

1. Introduction

The goal of this project was the labelling of images based on their color and shape using a pair of algorithms: K-means and K-NN respectively. We will have a set of images to which we will be able to assign one in 8 labels for the kind of clothing and one or more between 11 labels for their colors.

Following, we will talk with more detail about each of the algorithms seen in the theory lessons that implement the two parts of this project:

1.1 K-means

It is defined as a **not supervised** classifying method. This means we don't possess a dataset correctly labeled from which the algorithm can learn to classify. In this case, the K-means utility resides in its ability to find patterns in the pixels of the images, that is to say, finding the different colors that predominate in them.

First, it is established a predefined number of centroids and each one of them is assigned to an specific pixel in the image the distances between every pixel and these centroids are calculated to then classify each pixel by its nearest centroid. To continue, new centroids are calculated based on the center point of these groups that we just created and the distances are calculated again, creating a loop that will not stop until it converges, which means that the new centroids are the same as the old ones, thus, finding a solution.

To decide what number of centroids do we use, we will call it k, we can run the algorithm many times increasing this variable and seeing how it improves the results. Finding the final and adequate result depends on our understanding of a good enough solution, which we can define as the iteration of the algorithm where an important value that talks about its efficiency doesn't improve above certain level. In this project, we will use the within class distance as this value by default but later on we will talk about other possibilities.

Once we get the final centroids, the only thing left is to identify them with a color. To do this there is a function that assigns to a centroid the color that a human is more probable to identify it with when he or she sees it.

1.2 K-NN

This algorithm is a **supervised** classifying method. This means that it will need a data set already labeled to learn from so it can classify new elements.

The algorithm compares the new data yet to be labeled and compares it to all the training data. Looking at its K nearest neighbor points labels, the ones more repeated are selected and their class is assigned to the new data.

2. Implemented analysis methods

To analyze how well our program works, we need to be able to evaluate it by its accuracy and by its efficiency. The optimal solution would be that code that is a 100% accurate and also as fast as possible.

For these analysis we have implemented two kinds of methods:

2.1. Qualitative analysis

We dispose of 3 functions to check the quality of our algorithms:

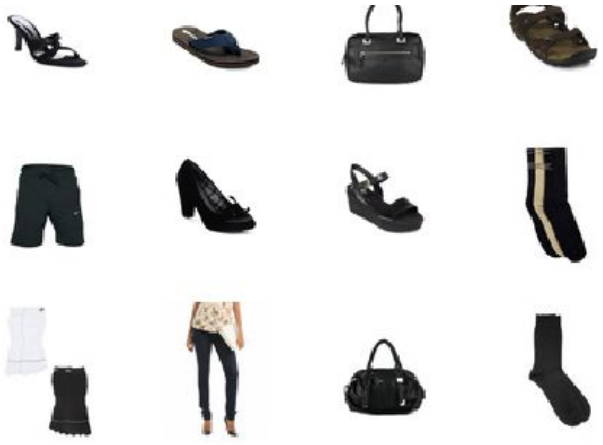
`Retrieval_by_color()`, `Retrieval_by_shape()` and `Retrieval_combined()`.

All of them are based on the same functionality: running the program with an specific query and different parameters and look at the results to see variations in these executions.

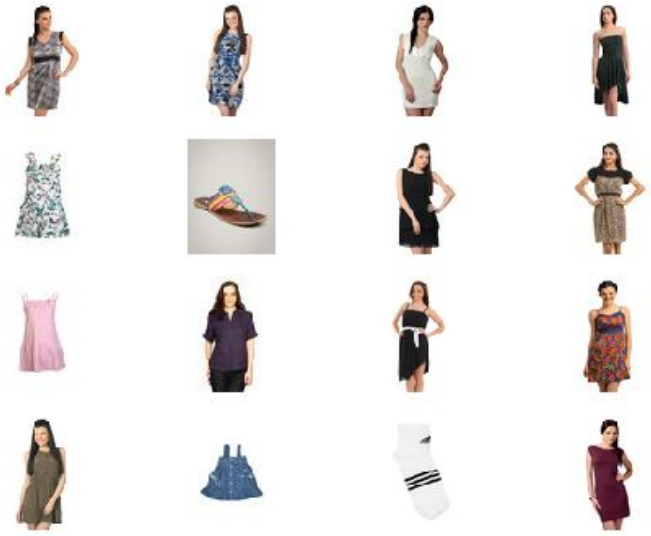
These functions receive a list of images, their labels after running our K-means or KNN algorithms, and the query made with the colors or shapes we wanted to find. They all return the images from the list that have the matching colors and shapes with the query.

The use of these results is pretty simple: to check if the algorithm actually works.

`Retrieval_by_color()`, asking for black items (`init = 'random'`):



`Retrieval_by_shape()`, asking for dresses (`k = 4`):



`Retrieval_combined()`, asking for black dresses:

Even though there are some mistakes, we can affirm that our algorithms work correctly, or at least, do what they are supposed to. It's important to notice that these results are achieved with low values for K in both algorithms and we see they are already pretty good.

Following, we will analyze more exhaustively how modifying the different parameters we obtain faster, slower, more accurate or less accurate versions of the program.

2.2. Quantitative analysis

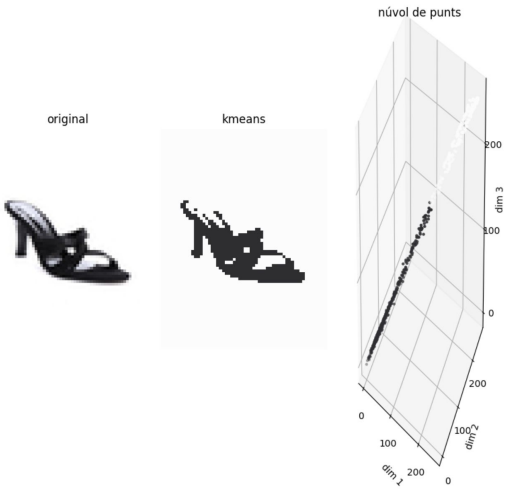
The principal reason of this analysis is to analyze how well our algorithm work quantitatively. We dispose of this 3 functions which will give us information with numbers of our algorithms and they will indicate us how accurate our algorithms are.

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

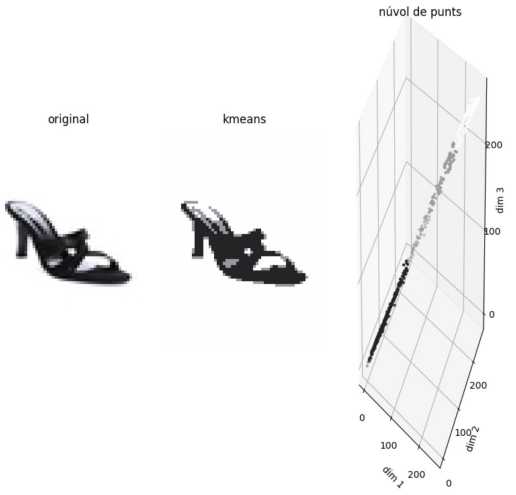
Kmean_statistics

This function gives us information about the K-means algorithm and how it works. It's very easy to see with the information that this function provides us how better is to have a higher K and how important this is.

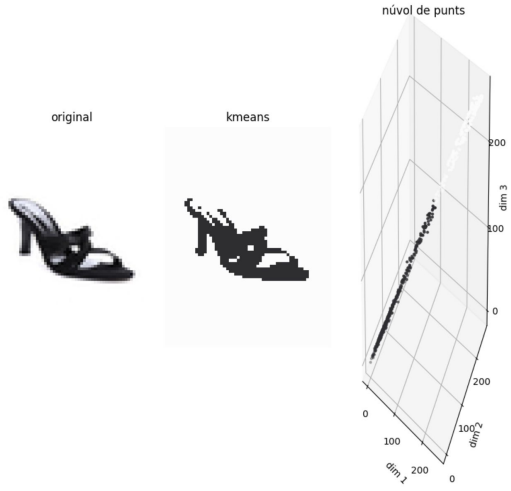
We can see here one example, where if we increase the K, we have more detail on the picture and this makes the function Kmeans do more interactions. However, we also can see that the WCD decreases and this is good because it means that the classes are compact.



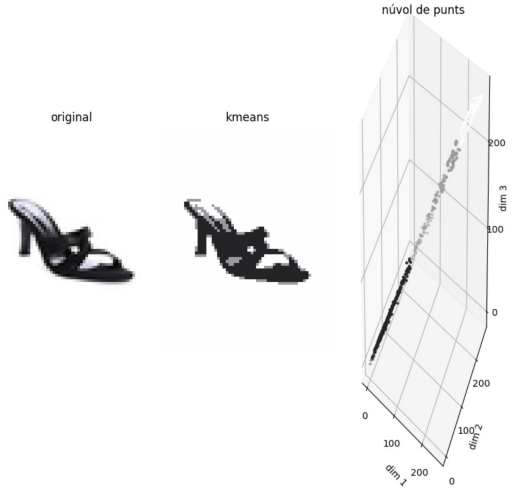
K: 2
WCD: 678.38832
Iterations: 5
Time: 0.41697s



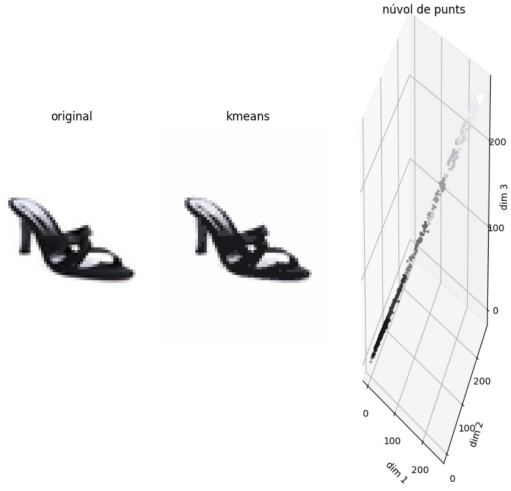
K: 3
WCD: 257.54503
Iterations: 17
Time: 0.3942368s



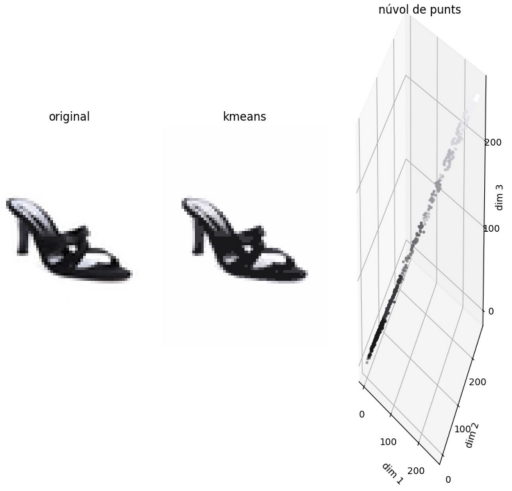
K: 4
WCD: 138.692
Iterations: 43
Time: 0.41008s



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



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

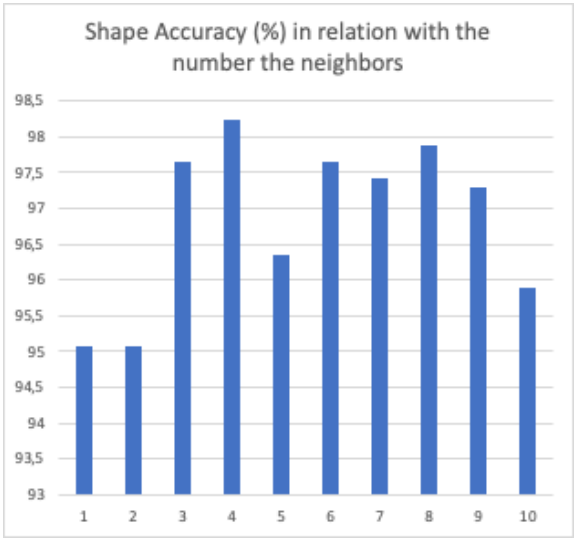


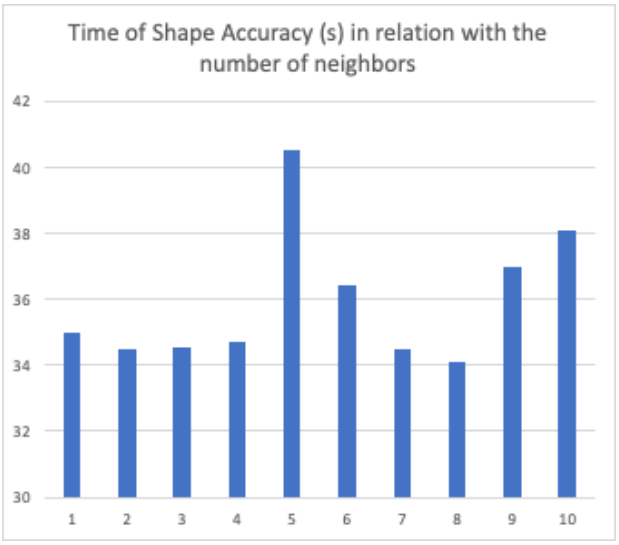
K: 9
WCD: 40.78192
Iterations: 272
Time: 0.3970969s

We can see as we increase the K, the number of iterations is increasing and the details of the picture is increasing as well. However we can see that the value of the within class distance is decreasing.

Get_shape_accuracy

This function gives us a percentage that will tell us how accurate the shape of our labels is.



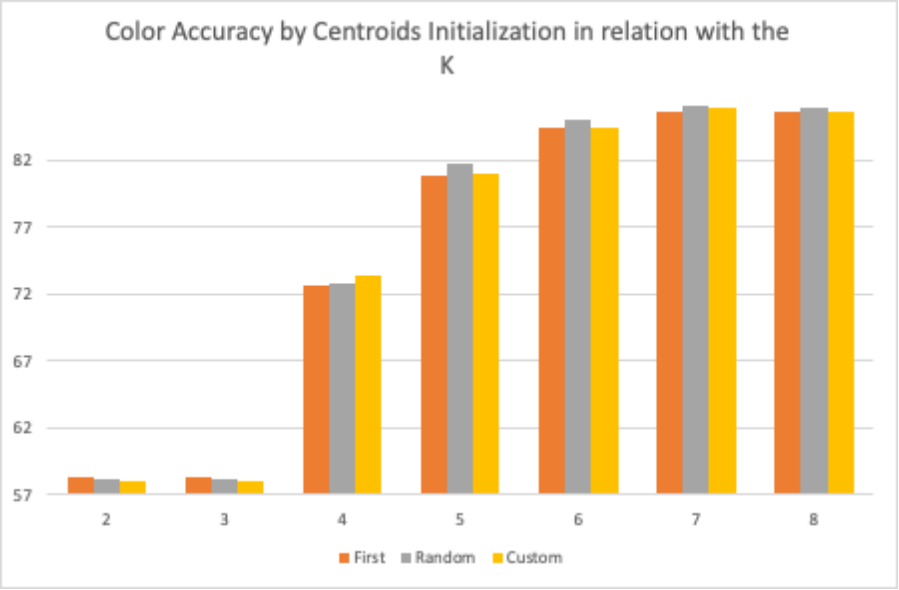


In the graphs above we can see some simulations we run for the different number of neighbors in the KNN algorithm.

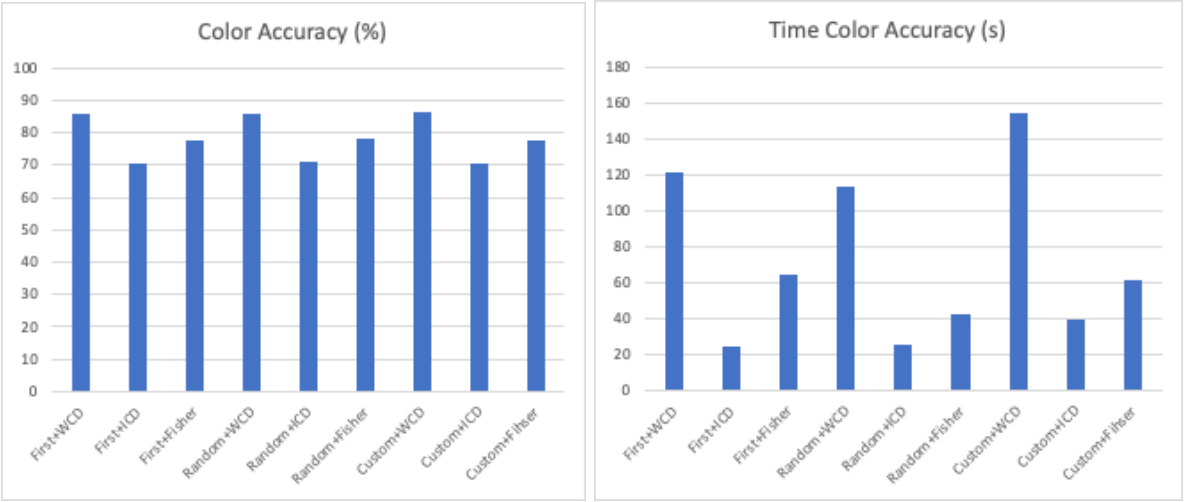
We can see that the time almost was constant in all the simulations and we obtain the maximum accuracy for a number of neighbors of 4.

Get_color_accuracy

This functions give us a percentage of how accurate the color of our labels are.

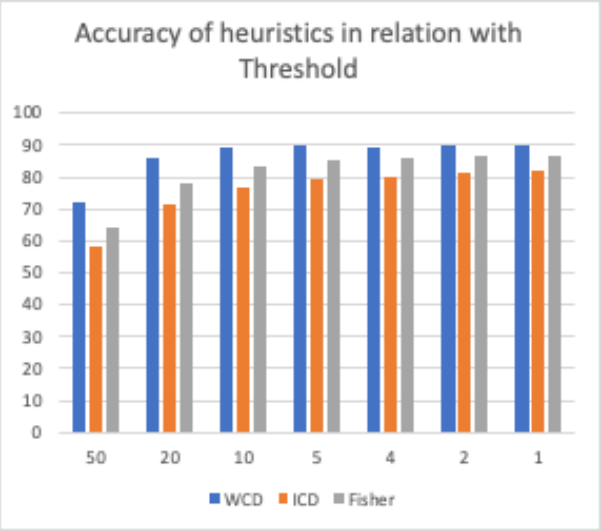


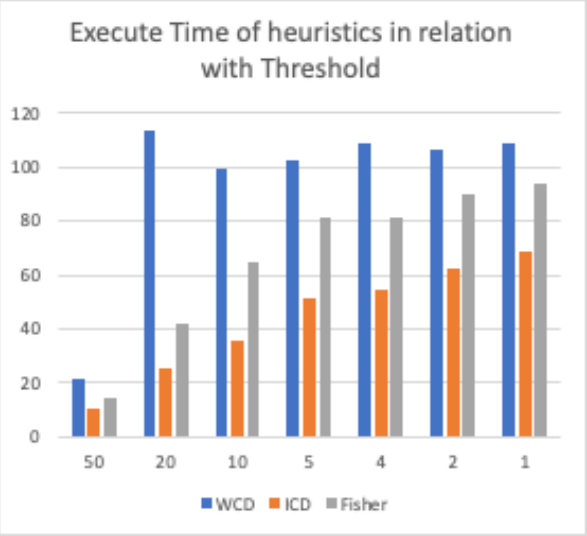
In this graph, we can see the different simulation for different types of initializations of the centroids of the classes. We can see that we get the best results with the higher K.



In this two graphs, we run simulations for the different type of heuristics and initialization of centroids. We can see we get the best results with the WCD for the three types of initializations of centroids. Also, the WCD simulations take more time.

Note: in this simulations we use K=9, Threshold=20.





In this simulations, we run the different heuristics with the initializations of centroids “random” for different threshold values.

We can see that to the higher the threshold gets, the performance is worst, we get worse accuracy.

3. Algorithm modifications

For the different algorithms used in the project we thought and implemented different modifications trying to improve the results or the speed to reach to them.

Due to the quantity of code and the actual results given by the standard versions, we decided to focus more on modifying K-means algorithm to improve it.

3.1. Heuristics

We started with the heuristics, we created differents versions of the initial heuristic to add more options, hoping that one of them would be better, that would be Fisher Coefficient.

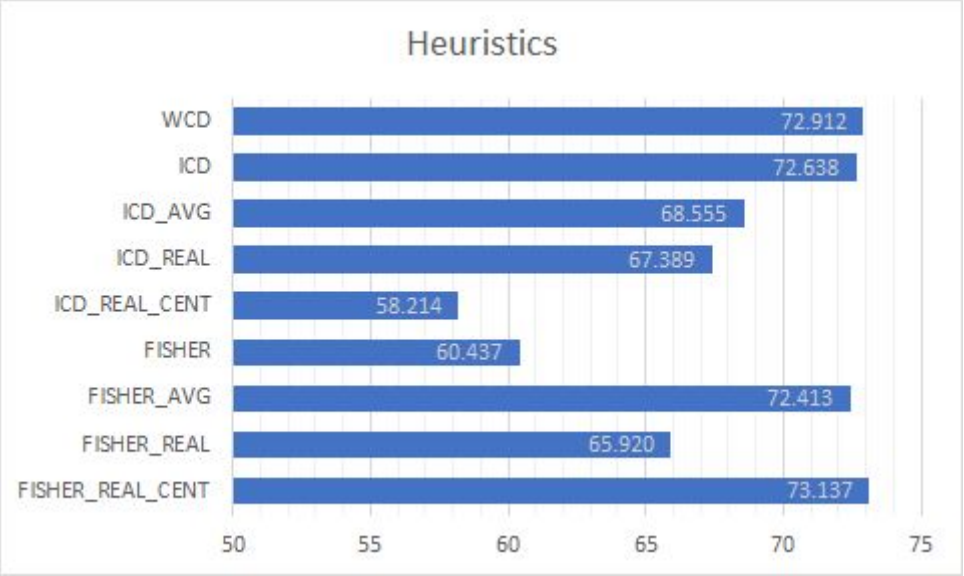
We added four different heuristics, *interclass distance* and the *Fisher coefficient*, the thing is that we made a simpler version of the *interclass distance* that was based on the distance between the centroids instead of the distance between the pixels of each class. That is an approximation to the result of the correct formula but much faster, so we are giving up a bit of precision in exchange for speed. For that, we could take the minimum distance between centroids or the average. The last one gives better results. So, we have two *Fisher coefficient*, one using the minimum, and the other using the average.

To clarify things, we are going to list and briefly explain each one of the final heuristics that we have created and implemented, we are also going to add the accuracy when we work with random centroid initialization, K = 4 and the threshold is 20%:

- WCD: *Within Class Distance*, is the average distance between the pixels of the same cluster and the centroid.
- ICD: *Inter Class Distance*, minimum squared distance between centroids of different classes.
- ICD_AVG: average of squared distances between centroids of different classes.

- ICD_REAL: average of squared distances between all pixels of different classes.
- ICD_REAL_CENT: average of squared distances between all pixels and the centroids of different classes.

The different *Fisher coefficient* are made with the different versions of the *inter class distance*.



It is important to know that with a higher number of centroids it would be more accurate.

We can see that the results are pretty similar, but the most accurates are WCD and FISHER_REAL_CENT, in matters of time, in the computer that they have been executed, they all last between 40 and 60 seconds, except for ICD_REAL and FISHER_REAL, with an approximate of 1650 seconds. These two are not the most precise, so knowing how much time they last, they would not be considered for use.

For testing we reversed the parameters while we were calculating the *Fisher coefficients*, turning them into *inter class distance / within class distance*, and the results were all worse except for FISHER, that improved a 7% respect the result of the graphic above.

3.2. Centroid initialization

To initialize the centroids we have a main method: *first*, t takes the first K colors that finds in the image. At the moment of creating this method, we created an alternative: *random*, it takes K random but different pixels as centroids.

Then, we tried to create a new initialization algorithm that was more effective, that is the *custom* initialization, its base concept is the same than the *random*. All centroids taken will be selected from a list with random centroids. There is a factor that limits the saved centroids. For example, we will keep in an array K · factor centroids, all of them without repeating themselves. From that array we will take the K centroids that meet the condition of having the largest distance between them and other centroids. The idea of this algorithm is to take the best random centroids as possible taking a good *inter class distance* from the beginning.

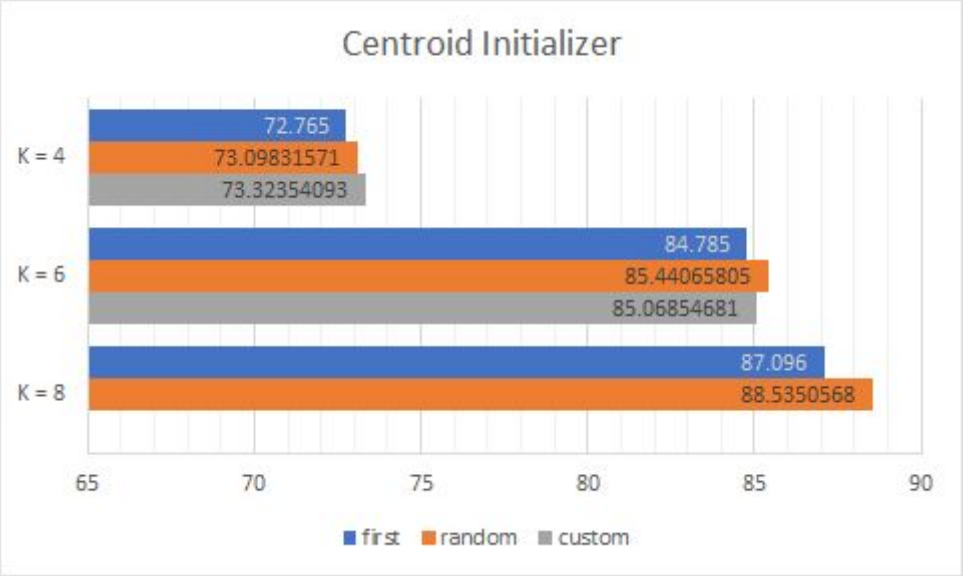
A new try of an algorithm is the one we called *hyper*, because the main idea was trying to obtain the centroids placed in the cube's hyperdiagonal. The algorithm is a mix of the concepts of the cube's hyperdiagonal with the *first* working method. We established a dictionary with the eleven main colors and which was their supposed RGB value. We would iterate over all the pixels, take which was the name color with more probability to be for that pixel, and check if we already have that color on the list. At the end we would take this array and check which were the corresponding RGB values for the name colors in the dictionary previously established, and those would be our centroids.

The last one is called *order*, the idea is to have an ordered array with the RGB values of the color, and take the K % 11 values as centroids. The colors are ordered in a subjective way that simulates the probability of this color and its relevance in the image. For example, we consider that in every image black is more probable and relevant than blue, so black appears first, and then does blue.

```
nombre_color = ['White', 'Black', 'Blue', 'Grey', 'Green',
'Red', 'Orange', 'Brown', 'Purple', 'Pink', 'Yellow']
```

We had problems with this two last methods because of the shape of the matrix, and we solve it changing the way to calculate the *get_centroids* function. These changes do not affect to the result. Then we found more problems, like jumping into an exception without executing even one single line, and we did not know how to solve them.

This study is made with K = 4, K = 6 and the FISHER_REAL_CENT heuristic, that was the one with best results at the previous test.



The execution of *custom* with K = 8 is missing due the time that takes to calculate the results, going over 30 minutes.

As we can see, despite our efforts to create more complex methods of centroid initialization, the *random*, just as simple as it is, is the one that gives the best results, even though the *custom* ones are close.

Analyzing all these results we should choose the best parameters to take the most reliable tests, that would be with: *random* initialization, and then, counting the similar results, we would have to think what heuristic we do prefer, WCD or FISHER_REAL_CENT.

4. Conclusion

In conclusion, we would like to go over what we have learnt with this project, mention some possible applications for this knowledge and also give our personal opinion.

First of all, we must highlight the programming language, Python. For us three, this was by far the biggest project we have made in this language so we needed our time to get used to it and also get the maximum performance out of it. We have discovered how powerful Python can be mainly by using the library Numpy, which improved enormously the performance of our code and its simplicity in some cases. We know that these improvements were caused mainly because of the ease with which Numpy can operate with arrays and vectors, avoiding looping over these data structures.

Before this project the idea of AI was mainly abstract in our minds, but getting to see how it works after coding it with our own hands made it real and very satisfying. We have learnt the basis of how pattern recognition algorithms like K-Means and KNN work, which gave us an insight of the huge world of AI programming.

This kind of algorithms could probably be used in problems that require computer vision power to solve them. We can see many real life examples of this technology, such as an image reverse search in Google, or programs that are able to read handwritten text in photos by recognizing the patterns of the letters, similar to what we do here with the shape of the clothing items. A very popular field where we see the use of similar technology is the autopilot mode in some automobile companies like Tesla Motors where the software of the car, using many cameras, needs to recognize from street signs to pedestrians crossing the streets.

Overall, we are left with a positive feeling about this project. We can say we have learnt but also enjoyed along the way. We all agree the organization made by the professors was good and made it easy for us to understand and follow what we were doing, apart from helping us in some cases with specific guidance.