# Practice 2: Labelling

## Part 1: K-means and color

## Artificial Intelligence Course

Computer Science Department
Universitat Autònoma de Barcelona

March 21, 2024

## 1 Introduction

In this practice, we will solve a simple image tagging problem. Given a set of images in a clothing catalog, we will develop algorithms that allow you to learn how to automatically label images by type and color. To simplify the practice, we will use a very small set of labels: 8 types of clothes and 11 basic colors. The system we will develop will run on a given image, and will return a label for a type of cloth and one or more labels for colors. We can see a general scheme in Figure 1. To do this we will implement the following algorithms seen at the theory class:

1. K-means (`k_means`) Unsupervised classification method used to find the predominant colors in an image.
2. K-NN (`k_nn`) Supervised classification method that will be used to assign clothing type labels.

In this first part of the practice we will deal with the K-means algorithm.

## 2 Required files

To do the project you will need to download the following folders and files:

1. `images`: Folder containing the image sets we will use. In this folder you will find:
   (a) `gt.json`: A file that contains information about the class of the images of Train.


Figure 1: Global goal of the project

(b) `Train`: Folder with the set of images that we can use as a learning set. Of all of them (train set) we have information on which class belongs to the file `Class_labels.cvs`.

(c) `Test`: Directory with the set of images that we want to tag and of which we have no information, this will be the set of experiments (test set).

2. `test`: Folder containing the set of files necessary to be able to perform the tests requested in the testing scripts (you should not use them in your scripts, the test functions load them automatically in the setUp).

3. `utils.py`: Contains a number of functions needed to convert color images to other spaces, mainly converting them to grey-level `rgb2gray()` and getting the 11 basic colors `get_color_test()`.

4. `Kmeans.py`: File where you will program the functions needed to implement K-means to extract the predominant colors.

5. `TestCases_kmeans.py`: A file you can use to check whether the functions you are programming in the `Kmeans.py` file give the expected result.

6. `KNN.py`: File where you will program the functions needed to implement KNN to label the clothe name (second part of the practice).

7. `TestCases_knn.py`: A file you can use to check whether the functions you are programming in the `Kmeans.py` file give the expected result (third part of the practice).

8. `my_labeling.py`: File where you will combine the two tagging methods and your enhancements to get the final tagging of the images (third part of the practice).

9. `utils_data.py`: Contains a series of functions you can use for the visualisation of results (third part of the practice).

# 3 Preparation

In this practice we will work with matrices for images and also with centroids that use iterative algorithms. To implement these algorithms efficiently, we recommend that you have a good command of the `Numpy` library to simplify your functions.

At the following links you can find some basic Numpy exercises that can help you understand how this library works:

- NumPy Basic (41 exercises with solution): https://www.w3resource.com/python-exercises/numpy/basic/index.php
- NumPy arrays (192 exercises with solution]): https://www.w3resource.com/python-exercises/numpy/index-array.php
- NumPy Sorting and Searching (8 exercises with solution): https://www.w3resource.com/python-exercises/numpy/python-numpy-sorting-and-searching.php
- Other exercises: https://www.w3resource.com/python-exercises/numpy/index.php

At the end of the tutorials you should be able to do the following operations with matrix `numpy`:

- Resize an array
- Calculate the average of the values of an array
- Perform operations between arrays and vectors (e.g., subtract one vector from each row in an array)
- Perform the above two operations only with certain rows of an array.

Figure 2 gives an example on how, using the `Numpy` library we can speed up the execution of an algorithm. There, you will find two versions of a code that determine the pixels that belong to each cluster for 1000 80×60 pixel images, in 6 clusters. The first version uses python lists and the second uses arrays with numpy, the first version turns out to be about 620 times more expensive than the numpy version. In this practice you will have to do a lot more operations than in this code, so it is essential that you do it this way if you want your code to be efficient.

# 4 What do we have to program?

In this first part of the practice, you will program the `K-means` classification algorithm to find the predominant colors in each image. We have divided all the tasks you need to do into the following 4 groups:

1. Functions for initializing `K-means` (section 4.1).
2. Functions required to implement `K-means` (section 4.2).

3. Function that implements `K-means` (section 4.3).

4. Functions that find the best k to apply `K-Means` to find the predominant colors (section 4.4).

5. Function that will convert the predominant colors into color name tags (section 4.5).

## 4.1 Initialization function

`_init_options`: Allows you to specify operating parameters of the `K-menas`. The entry is a dictionary that can contain the following keys with their possible values:

```
km_init:  ['first', 'random', 'custom'],
tolerance:  float,
maxiter:  int,
fitting:  ['WCD', ...].
```

If necessary, you can create as many parameters as you need.

`_init_X`: A function that receives a matrix of points X and does several things: (a) ensures that the values are of the `float` type; (b) where applicable, convert the data to a matrix of only 2 $N \times D$ dimensions; if not, and the input is an image of dimensions F $\times$ C $\times$ 3[1] then it will transform it and return the pixels of the same image, but in an array of dimensions $N \times 3$ and finally save this matrix to the variable X of the `K-Means` object, `self.X`.

*** Hint: You can use the reshape function of the NumPy library to resize the input image.

`_init_centroid`: Function that initializes the variables of the classes `centroids`, and `old_centroids`. These two variables will have a size of $K \times D$ where K is the number of centroid we have passed to the `K-Means` class and $D$ is the number of channels. It then assigns values to the centroids based on the initialization option we have chosen.

The 'first' option assigns to the centroid the first $K$ points in the `X` image that are different from each other. By default, you will need to program the 'first' option. The 'random' option will choose random centroids, so that they are not repeated. The 'custom' option may follow any other centroid initial selection policy that you consider.

** Hint: Points distributed on the diagonal of the data hypercube?

## 4.2 Functions for the `K-means`

In order to implement `K-means`, we will need a number of functions to update the clusters in each iteration of the algorithm. In Figure 3 we recall the pseudocode

[1]Here we assume that N is the number of pixels in an image that has $F$ rows and $C$ columns, so $N = F \cdot C$. The dimension of the feature space is given by $D$ which in color images is 3.

```
################################################
# Code determining the pixels that belong to each
# of the six clusters for 1000 80x60 images.
################################################

import time
import random import numpy as np K=6
N = 80*60 Iteracions = 1000

# Version with python lists.
############################
list = random.choices(range(K), k=N)
t0 = time.time()
grups = {}
for iteration in range(Iterations):
    grups = {}
    for k in range(K):
        grups[k] = [True if elem == k else False for elem
in list]
t1 = time.time()-t0
# Version with numpy matrix
############################
vector_np = np.random.randint(K, N)
t0 = time.time()
for iteration in range(Iterations):
    grups = {}
    for k in range(K):
        grups[k] = vector_np==k
t2 = time.time()-t0

print(f'time to obtain the {K} groups for {N} pixels,
{Iterations} times.')
print(f'        WITHOUT numpy:\t{t1} seconds')
print(f'        WITH numpy:\t\t{t2} seconds')
print(f'WITH is {t1/t2} times faster that WITHOUT numpy')
```

The execution gives:

```
        WITHOUT numpy: 4.341734886169434 seconds
        WITH numpy: 0.006999969482421875 seconds
WITH is 620.2505449591281 times faster than WITHOUT
numpy.
```

Figure 2: Example on the advantage of using the library `numpy`.

of this algorithm as we have seen in the theory classes. Here are the 4 necessary functions:

`distance` : Function that takes as input the image `X` ($N \times D$) and the centroid `C` ($K \times D$), and calculates the Euclidean distance between each point in the image with each centroid, and returns it as a matrix of dimension $N \times K$.

`get_labels` : Function that for each point in the `X` image, assigns which is the nearest centroid and saves it to the variable of the `KMeans` class: `self.labels`. Therefore this variable will be a vector of length the number of points of `X`. It contains values between 0 and $K - 1$.

`get_centroids` : Function that passes the value of the variable centroids to `old_centroids`, and calculates the new centroids. To do this, you need to calculate the center of all `X` points related to a centroid. And this must be done for each of the centroid.

`converges` : Function that checks if centroid and `old_centroids` are the same. If so, it means that `K-Means` has reached a solution, so the function will return True, if not False. To speed up the response time, you can use a certain tolerance for the difference between centroids and `old_centroids`, defined in the initialization options and / or the maximum number of iterations.

**Preliminary definitions:**

*X: Set of points that make up the training set.*

$$X = \{\vec{x}^i : \vec{x}^i = (x_1^i \quad \cdots \quad x_d^i)\ \forall i : 1..n\}$$

*K: Number of classes in which we want to divide the space*

$C_i$ *: Set of points of the i class*

$CI_i^t$ *: Centre of inertia of class i at time t.*

$d(\vec{x}, \vec{y})$ *: distance between two points.*

**Algorithm**

**Function** k-means(X,K)

1. Choose randomly K points of X to initialize

$$CI_i^0 = random(X^j), \forall i : 1 \ldots k, \forall j : 1 \ldots n$$

2. **Repeat**
   1. **For** ( each class $C_i, \forall i : 1 \ldots k$ ) do
      1. $C_i = \{x \in X : \forall\, j \neq i, d(x, CI_i^t) \leq d(x, CI_j^t)\}$
      2. Calculate the new centre:

$$CI_i^{t+1} = \left( \sum_{j=1}^{\#C_i} x_1^j \Big/ \#C_i \,, \quad \cdots \quad , \sum_{j=1}^{\#C_i} x_d^j \Big/ \#C_i \right)$$

   2. **EndFor**
   3. t++;

3. **Until** $(CI_i^t = CI_i^{t+1}; \forall\, i : 1..k)$

4. **Return** $(\{CI_i^t\}_{i:1..k})$

**EndFunction**

Figure 3: Pseudocode for the K-means algorithm

## 4.3 K-means

Now that we have programmed all the necessary functions of the `KMeans` class we can program the fit function that will group the points of the image in K

clusters.

`fit`: Function executed by the `Kmeans` algorithm and iterates over the steps shown in Figure 3, which are:

1. For each point in the picture, find the nearest centroid.
2. Calculate new centroids using the `get_centroids` function
3. Increase the number of iterations by 1
4. Check if it converges, in case of not doing so go back to the first step.

## 4.4 Functions to get the ideal K

Once we have programmed the k-means function we need to automate the parameter k that we will pass to it, and that will represent the number of predominant colors of each image. Therefore, we need to find the value of K that best fits the color distribution of each image. To do this we will use the intra-class distance for different K's, and we will keep that K from which the result of the intra-class distance is stabilized and almost does not decrease. We will do this with the following two functions which will be two functions of the Kmeans class:

`withinClassDistance` : Function belonging to the KMeans class, which calculates its intra-class or within-class-distance (WCD) distance and updates the WCD field with its value calculated as follows:

$$WCD = \frac{1}{N} \sum_{\mathbf{x} \in \mathbf{X}} dist(\mathbf{x}, \mathbf{C_x})^2 = \frac{1}{N} \sum_{\mathbf{x} \in \mathbf{X}} (\mathbf{x} - \mathbf{C_x})(\mathbf{x} - \mathbf{C_x})^T \qquad (1)$$

where $\mathbf{C_x}$ is the vector representing the cluster to which the point $\mathbf{x}$ belongs.

`find_bestK` : Function that belongs to the KMeans class and takes as input a number, max_K, which indicates the maximum k to be analyzed. Runs the Kmeans algorithm for every k from 2 to max_K on the data on which the KMeans object was initialized. For each k, calculate the WCD value, which should decrease as k increases. We will then calculate the percentage decrease in WCD for each k, as follows:

$$\%DEC_k = 100 \frac{WCD_k}{WCD_{k-1}} \qquad (2)$$

Where $WCD_k$ is the WCD on the result of the Kmeans with number of classes $k$. We will keep the $k$ from a high decrease to a stabilized decrease. So when $100 - \%DEC_k + 1$ is smaller than a threshold (for example 20%) then we will take these $k$ as the ideal a $k$ and it will be updated in the K field. If no one is found it will put max_K. You can better adjust this threshold globally.