Práctica 3.
Caracterización y clasificación de texturas.

31508811-1 Martiñón Luna Jonathan José.
31506842-1 Ortega Ibarra Jaime Jesus.
41800471-9 Tapia López José de Jesús.

I. Objetivo.

El alumno

- Desarrollará métodos de caracterización de texturas.
- Aprenderá a utilizar clasificadores como *K-NN*, *K-Means* o máquinas de soporte vectorial.

II. Ambiente

La práctica fue desarrollada en *"Jupyter Notebook"*, haciendo uso de Python 3.

III. Introducción.

La textura es una característica que nos permite identificar secciones u objetos dentro de imágenes. Ahora bien, para obtener una buena descripción de nuestras texturas, podemos hacer uso de diversas "*herramientas*":

- Energía
  La energía normalizada de una sub-imagen formada por N coeficientes se define como:
  $$E_n = \frac{1}{N}\sum_{j,k}\left[D_n(b_j, b_k)\right]^2$$
- Histograma
  El histograma de una imagen digital con L niveles de gris en el rango [0, L-1] es una función discreta de la forma:
  $$h(r_k) = \frac{n_k}{N};$$
  Con:
  k = 0,1,...,L-1
  N = Total de Pixeles
  n_k = Pixeles en ese nivel de gris
  r_k = k-esimo nivel de gris
- Matriz de Co-ocurrencia
  Nos ayudará a obtener también la matriz de probabilidad, esta matriz es es calculada en ciertas direcciones y con cierto valor de pixeles, se encargará de ver el cambio entre ellos, es decir, si de 0 pasamos a 1, nos mantenemos, etc. Se suman los valores por su transpuesta (Dirección contraria) y dividimos sobre el total de la suma de todos los valores (Normalizamos)
- Matriz de probabilidad
  Es calculada a partir de la matriz anterior y nos otorgará las probabilidades dentro de cada pixel, está dada por la fórmula:
  $$P_{i,j} = \frac{v_{i,j}}{\sum_{i,j} V_{i,j}}$$
- Contraste:
  $$\sum_{i,j=0}^{n-1} P_{i,j}(i-j)^2$$
- Homogeneidad:
  $$\sum_{i,j=0}^{n-1} \frac{P_{i,j}}{1+(i-j)^2}$$
- Disimilitud:
  $$\sum_{i,j=0}^{n-1} P_{i,j}|i-j|$$

K-NN

También conocido como algoritmo de vecinos más cercanos. Es un algoritmo supervisado que nos ayudará a agrupar mediante '*K*' vecinos más próximos, es decir, podemos seleccionar la cantidad de elementos en un grupo. Ocupa bastante espacio de memoria, pero es ideal para pequeños datos.

Nota:
En python se puede hacer uso de ella con la librería sklearn: '*from sklearn.neighbors import KNeighborsClassifier*'

K-means

Dicho algoritmo es no supervisado; trabajará con '*K*' centroides, a diferencia del anterior, donde seleccionabamos la cantidad de elementos en nuestros grupos, aquí buscaremos tener '*K*' grupos. Se comienza con centroides aleatorios y se toma distancia con los puntos se decide que las menores correspondan a cierto grupo, finalmente de los puntos se calcula el promedio en (X,Y) y se obtienen nuevos centroides; termina en el momento en que se cumplan ciertas iteraciones, no haya cambio de centroides o hayamos conseguido cierto margen de error.

Nota:
En python se puede hacer uso de ella con la librería sklearn: '*from sklearn.cluster import KMeans*'

LDA Análisis Discriminante Lineal.

Se trata de un método de clasificación para variables cualitativas donde se conozcan las probabilidades '*a priori*' de al menos 2 grupos y nueva observaciones a clasificar de acuerdo con sus características, se utiliza el teorema de Bayes.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Para determinar la probabilidad de que pertenezca a una de las clases, se clasificará dentro de la que tenga mayor probabilidad. Es una alternativa a regresión logística en cuanto a que la variable cualitativa presente más de 2 niveles, a diferencia de regresión, no presenta inestabilidades dentro de los parámetros con clases bien separadas. Eso sí, al tener valores de 2 niveles LDA y Regresión presentan resultados similares. Básicamente consta de 5 pasos:

1) A partir de nuestros grupos conocidos calcular las probabilidades *a priori*.
2) Determinar si la varianza o matriz de covarianzas es homogénea para cada grupo.
3) Estimar las probabilidades condicionales.
4) Calcular el resultado de la discriminante.
5) Nos apoyamos en *Cros-validation* para estimar calsificaciones erróneas.

Nota:
En python se puede hacer uso de ella con la librería sklearn: '*from sklearn.discriminant_analysis import LinearDiscriminantAnalysis*'

SVM Máquinas de soporte vectorial.

Sirven para tratar con límites no lineales Trabaja con *kernels*, los cuales son enfoques más eficientes utilizados para calcular la similitud entre 2 observaciones en una nuevo espacio dimensional. Tenemos 3 tipos de kernel.

1) Lineal
   Usa el método de correlación de Pearson
$$K(x_i, x_i') = \sum_{j=1}^{P} x_{i,j}, x_{i,j}'$$
2) Polinómico
   Con grado d (d>1), permite una decisión más 'flexible'
$$K(x_i, x_i') = \left(1 + \sum_{j=1}^{P} x_{i,j}, x_{i,j}'\right)^d$$
3) Radial
   Sirve para datos locales, digamos, sólo serviría para aquellas observaciones parecidas a nuestro conjunto de entrenamiento.
$$K(x_i, x_i') = exp\left(-\gamma \sum_{j=1}^{P} (x_{i,j}, x_{i,j}')^2\right)$$
   Donde γ es una constante positiva que representa la flexibilidad del SMV, entre más grande más flexible.

Nota:
En python se puede hacer uso de ella con la librería sklearn: '*from sklearn import svm*'

IV. Desarrollo.

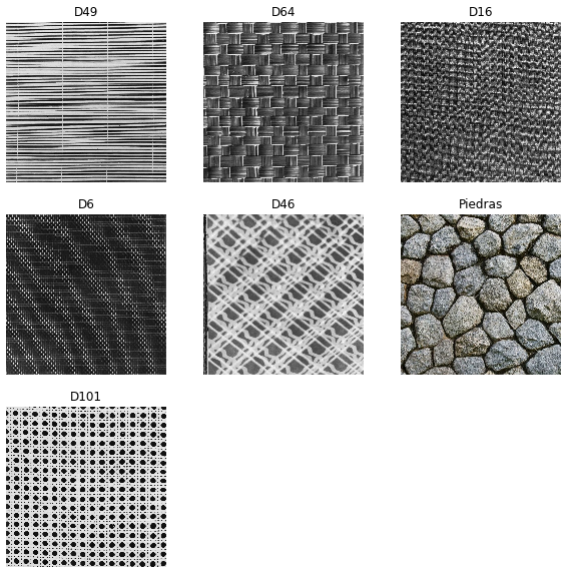
Parte A)

1) A partir de un conjunto de texturas que se le proporciona al alumno generar un sistema de "image retrieval" mediante un proceso de reconocimiento de patrones Usar de 4 a 6 imágenes texturas de la base de datos Brodatz para el proceso. A cada imagen subdivirla en "n" ventanas cuadradas (escoger las dimensiones adecuadas de acuerdo a las texturas que utilce) y guardar 1 o 2 de las mismas para el proceso de prueba siendo el resto para el proceso de entrenamiento. Obtener información característica de las imágenes a partir del proceso de extracción de características que entrega la matriz de Haralick o GLCM (gray level coocurrence matrix). Generar al menos 2 matrices de Haralick con los parámetros de distancia y ángulo. De la matriz de Haralick obtener entropía, energía u otra información; con estos datos generar un vector de características que se utilizará tanto para el entrenamiento como para la prueba.

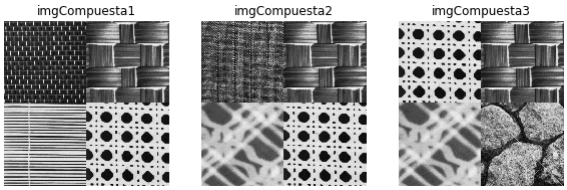
   Primero, importamos las bibliotecas que usaremos:

```
from skimage import io
from skimage import color
import matplotlib.pyplot as plt
import numpy as np
from skimage.segmentation import slic
from skimage.segmentation import mark_boundaries
from skimage.util import img_as_float
from skimage.exposure import rescale_intensity
from sklearn.model_selection import
    train_test_split
from sklearn.neighbors import KNeighborsClas
    sifier
from sklearn import cluster
from skimage.feature import greycomatrix,
    greycoprops
from sklearn.metrics import clas
    sification_report
from os import listdir
import matplotlib.image as mpimg
from PIL import Image
from math import ceil
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import
    train_test_split
```

Para este caso elegimos 7 imagenes de texturas:



Esto debido a que eran las que se presentaban en nuestro conjunto de imágenes compuestas:



La segunda imágen de piedras no aparecía en ninguna, por lo que consideramos irrelevante el trabajar con ella, simplemente sería información no necesaria a nuestro modelo.

Definimos una función para leer imágenes y convertirlas directamente a formato blanco y negro.

```
def Lee_imagen(ruta): # Leemos la imagen y la convertimos a blanco y negro
    imagen = Image.open(ruta).convert('L')
    imagen = np.asarray(imagen)
    return imagen
```

Posterior a ello convertimos nuestras imágenes a usar

```
imagenes = []
for img in rutas_images:
    imagenes.append(Lee_imagen(img))
```

Para poder saber que estábamos seleccionando ventanas adecuadas, decidimos generar una función que nos ayudase a comparar la ventana generada con la imagen original:

```
def Visualiza_ventana(ventana,original,name): # Veremos la ventana y compararemos con la original
    print(f"Imagen: {name}")
    plt.figure(figsize=(7, 7))

    plt.subplot(1, 2, 1)
    plt.imshow(ventana,cmap='gray')
    plt.title('Ventana')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.imshow(original,cmap='gray')
    plt.title('Original')
    plt.axis('off')


    plt.show()

```

Con todo eso, generar una ventana era un proceso repetitivo, motivo por el cual decidimos que lo mejor sería generar otra función que realizara dicho cometido, asegurándose de que nuestros datos se ajustasen a los requerimientos que la función encargada de generar la matriz GLCM requería.

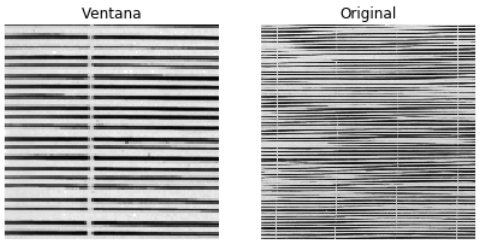
```
def Genera_ventana(imagen,S_window,Ren,Col): # Recibe los pixeles y el tama o de la ventana

    #Recibe la posic n inicial a generar la ventana, digamos Ren = 1, Col = 1, a partir
    # De ah  se contaran los renglones y columnas de acuerdo al tama o de ventana deseada

    Ventana = np.array([])


    flag = True

    for i in range(Ren,Ren+S_window):

        aux = np.array([]) # Se resetea cada que se cambia de posici n

        for j in range(Col,Col+S_window): # Guardamos todos los datos

            aux = np.append(aux, [imagen[i][j]], axis=0)
            # Vamos a multiplicar por 10 y redondear, esto es por que
            # M s adelante nuestra funci n GLCM trabajar  con enteros

        if flag == True: # En caso de que sea el primer elemento, se agrega
            # Al rengl n 0
            Ventana = np.append(Ventana,aux,axis = 0)
            Ventana = np.array([Ventana]) # Duplicamos la dimensi n para ir a adiendo filas
            flag = False

        else: # En caso que ya haya datos, simplemente lo agregamos como otro rengl n

            aux = np.array([aux]) # Duplicamos la dimensi n

            Ventana = np.append(Ventana,aux,axis = 0)


    return Ventana.astype(int)
```

Una vez que teníamos lo anterior, pudimos visualizar nuestras ventanas, trabajar con ellas y modificarlas de tal forma que se adecuaran lo más posible a nuestras necesidades:

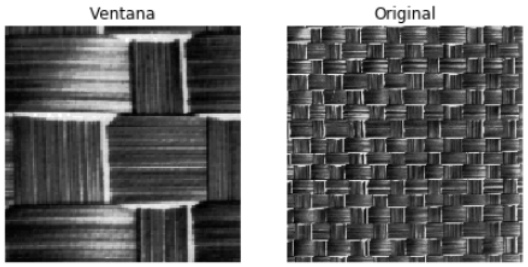
```
original = imagenes[0]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,79,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'D49')

original = imagenes[1]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,79,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'D64')

original = imagenes[2]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,121,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'D16')

imagenes[3] = imagenes[3]
original = imagenes[3]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,79,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'D6')

imagenes[4] = imagenes[4][20:,20:]
original = imagenes[4]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,79,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'D46')

original = imagenes[5]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,79,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'Piedras')

original = imagenes[6]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,79,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'D101')

```

Los resultados obtenidos fueron

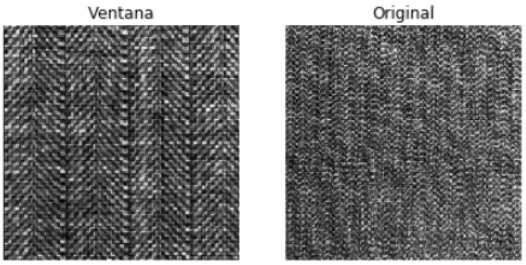
```
Tamaño de la original: (640, 640)
Tamaño de la Ventana: (121, 121)
Imagen: D49
```



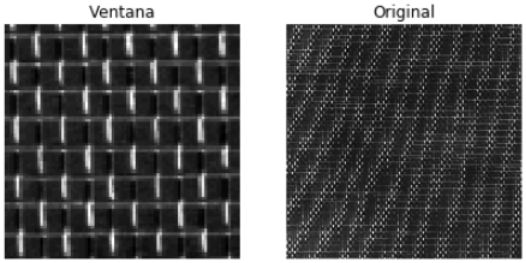

```
Tamaño de la original: (640, 640)
Tamaño de la Ventana: (121, 121)
Imagen: D64
```



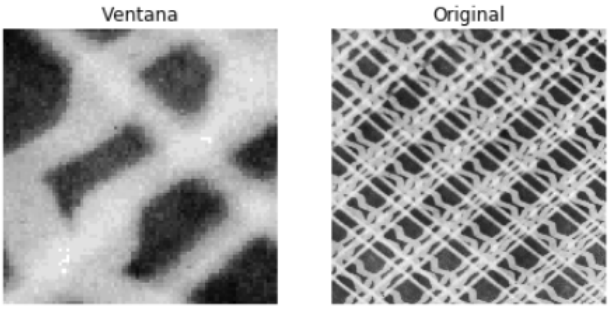

```
Tamaño de la original: (640, 640)
Tamaño de la Ventana: (121, 121)
Imagen: D16
```



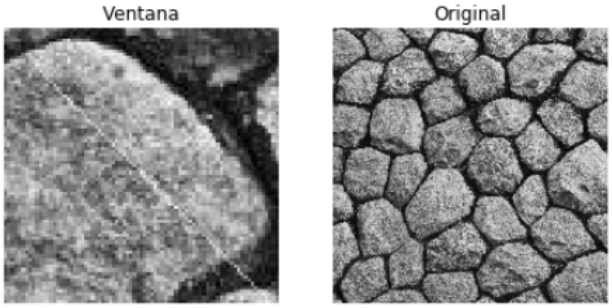

```
Tamaño de la original: (640, 640)
Tamaño de la Ventana: (79, 79)
Imagen: D6
```



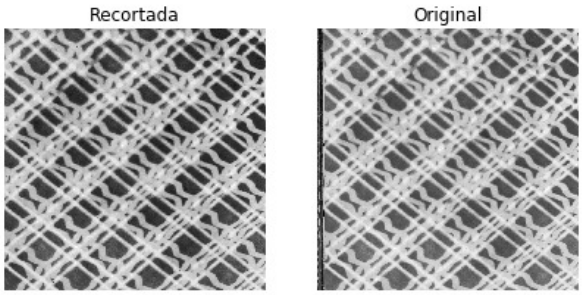

```
Tamaño de la original: (620, 620)
Tamaño de la Ventana: (79, 79)
Imagen: D46
```



```
Tamaño de la original: (641, 641)
Tamaño de la Ventana: (79, 79)
Imagen: Piedras
```

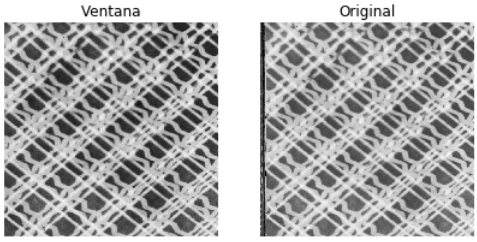



```
Tamaño de la original: (640, 640)
Tamaño al recortar: (620, 620)
Imagen: D46
```



Donde la única imagen que sufrió cambios fue 'D46', pues presentaba una línea negra a la izquierda, cosa que evidentemente no era parte de la textura y pensamos que podría llegar a ocasionarnos problemas de cálculos, por eso se decidió ajustarla.

```
Tamaño de la original: (640, 640)
Tamaño de la nueva: (620, 620)
Imagen: Recortada
```



Una vez que elegimos el tamaño de la ventana óptimo, decidimos proceder a generar todas las ventanas posibles para cada imagen, en un principio nos movimos pixel por pixel, sin embargo era altamente costoso y no contamos con la infraestructura para realizar dicha función, motivo por el cual, decidimos movernos de ventana en ventana, haciendo esto más sencillo, pero aún así obteniendo un buen número de muestras

```
#Almacenamos el nombre da cada imagen
name_imgs = ['D49','D64','D16','D6','D46','Piedras','D101']

#Almacenamos el tama o de cada ventana
#Todas fueron el mismo
S_w = 79
```

```
#Definimos la funci n encargada de generar todas
#Las ventanas posibles en una imagen
def Almacena_ventana(tama o,imagen):
    almacen = []

    tam = imagen.shape[0] # l mite de nuestra imagen

    i = 0
    while i <= tam-tama o:
        j = 0
        while j <= tam-tama o:
            almacen.append(Genera_ventana(imagen,tama o,i,j))
            j+=tama o
        i+=tama o

    return np.array(almacen)

#Finalmente, creamos nuestras ventanas
Ventanas = []
for imagen in imagenes:
    Ventanas.append(Almacena_ventana(S_w,imagen))

#Mostramos las ventanas que contamos para
#Cada imagen
print("-----------------------------------------")
for i,imagen in enumerate(Ventanas):
    print(f"Contamos con {imagen.shape[0]} ventanas para la imagen {name_imgs[i]}")
print("-----------------------------------------")
```

```
Contamos con 64 ventanas para la imagen D49
Contamos con 64 ventanas para la imagen D64
Contamos con 64 ventanas para la imagen D16
Contamos con 64 ventanas para la imagen D6
Contamos con 49 ventanas para la imagen D46
Contamos con 64 ventanas para la imagen Piedras
Contamos con 64 ventanas para la imagen D101
```

Ahora bien, llegados a este punto necesitabamos obtener nuestras matrices GLCM y de ahí nuestros vectores, motivo por el cual ocupamos 2 funciones: una que devolvía las características a partir de una matriz GLCM y otra que creaba la matriz y llamaba a la que obtenía las características, así:

```
def Caract_GLCM(GLCM):


    vec =np.array([greycoprops(GLCM,'contrast')[0][0]])


    aux = np.array([greycoprops(GLCM,'dissimilarity')[0][0]])
    vec = np.append(vec,aux,axis=0)

    aux =np.array([greycoprops(GLCM,'homogeneity')[0][0]])
    vec = np.append(vec,aux,axis=0)

    aux = np.array([greycoprops(GLCM,'ASM')[0][0]])
    vec = np.append(vec,aux,axis=0)
```

```
    aux = np.array([greycoprops(GLCM,'energy')[0][0]])
    vec = np.append(vec,aux,axis=0)

    aux =np.array([greycoprops(GLCM,'correlation')[0][0]])
    vec = np.append(vec,aux,axis=0)

    return vec

def Obten_Carac_GLCM(Ventanas,Max_Gray_level,distancia,direcciones):

    '''

    parametros

    Ventanas       ==>   Son las ventanas totales a calcular su GLCM (Puede ser 1)   (LIST)
    Max_Gray_level ==>   Maximo valor de Gris, se coloca en levels                   (INT)
    Distancia      ==>   Distancia entre pixeles a evaluar                           (INT)
    Direcciones    ==>   Las direcciones a evaluar en funcion de pi                  (LIST)

    Direcciones frecuentes:

    Norte - sur:   np.pi/2
    Sur - Norte:   3*np.pi/2
    Este - Oeste:  0
    Oeste - Este:  np.pi

    '''


    flag = True

    for ventana in Ventanas:
        # Evaluaremos cada una de las ventanas
        GLCM = greycomatrix(ventana,
                            [distancia],
                            direcciones,
                            levels=Max_Gray_level,
                            normed=True,
                            symmetric=True)

        if flag == False:

            aux = np.array([Caract_GLCM(GLCM)])
    #Obtenemos caracter stica
            Vect_car_Vent = np.append(Vect_car_Vent,aux,axis = 0) # Lo agregamos

        else: # Primer Dato

            Vect_car_Vent = np.array([Caract_GLCM(GLCM)])
            flag = False

    return Vect_car_Vent

```

Fueron funciones separadas por si en algún momento contábamos ya con la matriz GLCM y sólo deseábamos obtener sus características. Si nos fijamos bien, la primera función devuelve **todas** las características, esto fue a debido a que en un inicio no sabíamos bien cuáles íbamos a usar y nos parecía más sencillo ir discriminando y seleccionando, que estar modificando codigo interno y regenerando valores.

Como sabemos, podemos elegir la dirección de nuestra matriz, motivo por el cual, mostramos las imágenes y de acuerdo con un análisis, decidimos si debía ser horizontal, vertical o incluso diagonal.

```
# Teniendo en cuenta que la imagen es:
plt.imshow(Ventanas[0][0],cmap='gray')
plt.axis('off')
plt.show()
#Lo mejor ser a de norte a sur
#Creamos una lista que almacenar  los vectores por
#Imagen
Vect_caract = []
Vect_caract.append(Obten_Carac_GLCM(Ventanas[0],
                                    256,
                                    7,
                                    [np.pi/2, 3*np.pi/2]))
# Teniendo en cuenta que la imagen es:
plt.imshow(Ventanas[1][0],cmap='gray')
plt.axis('off')
plt.show()
#Podriamos probar en diagonal -> np.pi/4,5 * np.pi/4
Vect_caract.append(Obten_Carac_GLCM(Ventanas[1],
                                    256,
                                    7,
                                    [np.pi/4,5*np.pi/4]))

# Teniendo en cuenta que la imagen es:
plt.imshow(Ventanas[2][0],cmap='gray')
plt.axis('off')
plt.show()
#Podr amos probar en horizontal
Vect_caract.append(Obten_Carac_GLCM(Ventanas[2],
                                    256,
                                    7,
                                    [0,np.pi]))

# Teniendo en cuenta que la imagen es:
plt.imshow(Ventanas[3][0],cmap='gray')
plt.axis('off')
plt.show()
#Podr amos probar en horizontal o vertical, probaremos horizontal por las luces
Vect_caract.append(Obten_Carac_GLCM(Ventanas[3],
                                    256,
                                    7,
                                    [0,np.pi])) # Buscando disimilitud

# Teniendo en cuenta que la imagen es:
plt.imshow(Ventanas[4][0],cmap='gray')
plt.axis('off')
plt.show()
#Podr amos probar en horizontal o vertical, probaremos horizontal
Vect_caract.append(Obten_Carac_GLCM(Ventanas[4],
                                    256,
                                    7,
                                    [0,np.pi]))

# Teniendo en cuenta que la imagen es:
plt.imshow(Ventanas[5][0],cmap='gray')
plt.axis('off')
plt.show()
#Podr amos probar en horizontal o vertical, probaremos vertical
```

```
Vect_caract.append(Obten_Carac_GLCM(Ventanas[5],
                                    256,
                                    7,
                                    [np.pi/2, 3*np
    .pi/2]))

# Teniendo en cuenta que la imagen es:
plt.imshow(Ventanas[6][0],cmap='gray')
plt.axis('off')
plt.show()
#Podr amos probar diagonal
Vect_caract.append(Obten_Carac_GLCM(Ventanas[6],
                                    256,
                                    7,
                                    [np.pi/4,5*np.
    pi/4]))

#Verificamos contar con 7 imagenes
len(Vect_caract)
7
```

2) Generar un clasificador mediante las funciones de Python para probar sus vectores de datos obtenidos.

   a) Utilizar al menos dos clasificadores, K-NN, Bayes o SVM (máquinas de soporte vectorial), de manera que pueda comparar sus resultados.

   Para dicho apartado hemos utilizado la librería de $scikit-learn$, pues nos proporciona ambos algoritmos, tanto $Navie\ Bayes$, como $K-Nearest-Neighbor$, los cuales se ponen a entrenamiento asignando datos de entrenamiento X_{train} y y_{train}, los cuales podemos obtener con ayuda de $train\_test\_split$, proporcionado de igual manera por $scikit-learn$, antes de realizar dicho proceso, juntamos dentro de un DataFrame todas nuestros vectores característicos de la siguiente manera:

```
contraste = []
disimilitud = []
homogeneidad =[]
ASM = []
energia = []
Correlacion = []
Y = []
# Vamos a extraer los datos
for i,imagen in enumerate(Vect_caract):
    for vector in imagen:
        contraste.append(vector[0])
        disimilitud.append(vector[1])
        homogeneidad.append(vector[2])
        ASM.append(vector[3])
        energia.append(vector[4])
        Correlacion.append(vector[5])
        Y.append(i+1)
```

Donde $Vect\ caract$ contendrá todos nuestros vectores de las imágenes seleccionadas.

```
import pandas as pd
data = pd.DataFrame({'Contraste':contraste, '
    Disimilitud':disimilitud, 'Homogeneidad':
    homogeneidad, 'ASM':ASM, 'Energia':
    energia, 'Correlacion':Correlacion, '
    Textura':Y})
data
```

Dando como resultado el siguiente DataFrame:

	Contraste	Disimilitud	Homogeneidad	ASM	Energia	Correlacion	Textura
0	10384.186533	74.702356	0.076781	0.003800	0.061642	0.113456	1
1	11259.399965	78.211498	0.095497	0.003147	0.056097	0.149427	1
2	11208.073312	79.771273	0.077533	0.002855	0.053430	0.126490	1
3	12127.325070	84.143636	0.055149	0.002141	0.046276	0.078069	1
4	12322.782349	84.419831	0.051000	0.001924	0.043863	0.080528	1
...	...	...	...	...	...	...	...
364	11208.971147	66.433163	0.239443	0.028918	0.170053	0.267458	6
365	11385.506026	66.579803	0.205638	0.022022	0.148399	0.292136	6
366	11602.647370	67.916180	0.198248	0.016241	0.127440	0.285282	6
367	10718.653031	64.403944	0.159305	0.012724	0.112801	0.311058	6
368	11314.833820	66.773192	0.158163	0.012890	0.113533	0.244771	6

Notemos que en la columna $Texturas$ se encuentran las etiquetas que hemos asignado a nuestras características, las cuales hacen referencia a las distintas imágenes procesadas desde un inicio. Una vez obtenidos nuestros datos, procedemos a dividirlos para entrenar nuestros algoritmos tal como se muestra en el siguiente código para el caso de $K-Nearest-Neighbors$:

```
X = data[['Contraste','Disimilitud','
    Homogeneidad','ASM','Energia','
    Correlacion']]
Y = data[['Textura']]
from sklearn.neighbors import KNeighborsClas
    sifier
from sklearn.model_selection import
    train_test_split
knn = KNeighborsClassifier(n_neighbors=12,
    metric='euclidean')
X_train, X_test, y_train, y_test =
    train_test_split(X, Y, test_size = 0.30,)
knn.fit(X_train, y_train)
predict = knn.predict(X_test)
predict
```

Dando como salida los siguientes valores.

```
array([6, 6, 2, 3, 6, 2, 3, 6, 6, 1, 1, 1, 2,
       2, 1, 6, 3, 6, 5, 2, 3, 4, 1, 3, 3, 3,
       4, 6, 5, 6, 1, 6, 1, 3, 2, 6, 5, 1, 4, 4,
       4, 2, 1, 3, 3, 4, 5, 3, 6, 3, 4, 1, 3,
       3, 6, 6, 3, 2, 3, 6, 4, 1, 3, 6, 2, 4, 2,
       1, 2, 6, 2, 6, 3, 1, 6, 4, 1, 5, 3, 2,
       2, 6, 2, 3, 3, 2, 4, 6, 4, 2, 3, 4, 3, 6,
       2, 2, 6, 2, 6, 2, 5, 5, 4, 6, 2, 1, 3,
       5, 3, 6, 5])
```

Y para el caso de Naive Bayes con el siguiente código:

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)
y_pred = gnb.predict(X_test)
y_pred
```

Dando el siguiente resultado:

```
array([6, 6, 3, 3, 6, 5, 2, 1, 6, 1, 1, 1, 2,
       2, 1, 6, 2, 6, 2, 2, 3, 4, 1, 3, 3, 2,
       4, 1, 2, 6, 1, 1, 1, 3, 2, 1, 5, 1, 4, 5,
       4, 3, 1, 3, 3, 4, 5, 3, 6, 3, 4, 1, 3,
       3, 6, 1, 5, 5, 3, 6, 2, 1, 3, 6, 2, 2, 2,
       6, 5, 1, 5, 6, 3, 1, 6, 4, 1, 4, 3, 5,
       5, 1, 5, 3, 3, 5, 4, 6, 2, 2, 3, 4, 3, 6,
       5, 2, 6, 3, 6, 5, 5, 5, 4, 6, 5, 1, 3,
       2, 2, 6, 2])
```

b) Para cada clasificador escoger y explicar cuál será el método de validación que utilice. Para este punto de la práctica hemos decidido obtener un reporte de la clasificación, la cual en automático nos indica cual es la puntuación que obtiene cada uno de los métodos. Para el caso de KNN:

```
print(classification_report(y_test, predict))
```

Obteniendo los siguientes resultados:

	precision	recall	f1-score	support
1	0.73	0.61	0.67	18
2	0.32	0.39	0.35	18
3	0.76	0.83	0.79	23
4	0.71	0.83	0.77	12
5	0.56	0.29	0.38	17
6	0.73	0.83	0.78	23
accuracy			0.64	111
macro avg	0.64	0.63	0.62	111
weighted avg	0.64	0.64	0.63	111

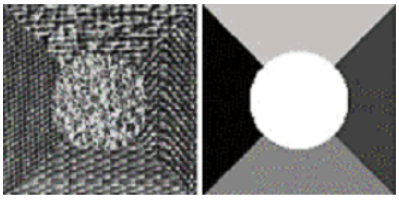
Para el caso de Naive Bayes:

```
print(classification_report(y_test, y_pred))
```

Obteniendo los siguientes resultados:

	precision	recall	f1-score	support
1	0.86	1.00	0.92	18
2	0.89	0.94	0.92	18
3	0.96	0.96	0.96	23
4	1.00	0.92	0.96	12
5	0.94	0.94	0.94	17
6	1.00	0.87	0.93	23
accuracy			0.94	111
macro avg	0.94	0.94	0.94	111
weighted avg	0.94	0.94	0.94	111

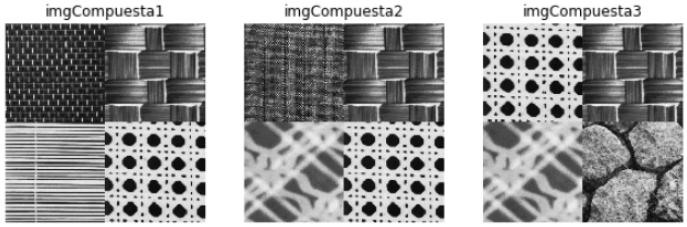
c) Una vez que ya realizó este entrenamiento para al menos 4 texturas, evaluar sobre la imagen compuesta de varias texturas que se le proporciona. Y clasificarla obteniendo la máscara de segmentación similar como se muestra en la siguiente figura



El objetivo de la práctica es realizar una evaluación de sus clasificadores intra-clase e inter-clase (entre clases).

- Despliegue sus resultados, cuando considere cambios en las pruebas y/o parámetros que valieron la pena.
- Realice varias pruebas variando sus estadísticos o parámetros de descripción de características (features), así como el clasificador.
- Explicar por qué obtuvo los resultados y qué pasaría si se varia algún parámetro utilizado en el método.

Para este apartado, inicialmente identificamos las imágenes compuestas, las cuales son las siguientes:



Una vez identificadas las imágenes, procedemos a obtener las ventanas de la siguiente manera:
Imagen 1:

```
original = ImgComp[0]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,41,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'imgCompuesta1')
```

```
Tamaño de la original: (320, 320)
Tamaño de la Ventana: (41, 41)
Imagen: imgCompuesta1
```

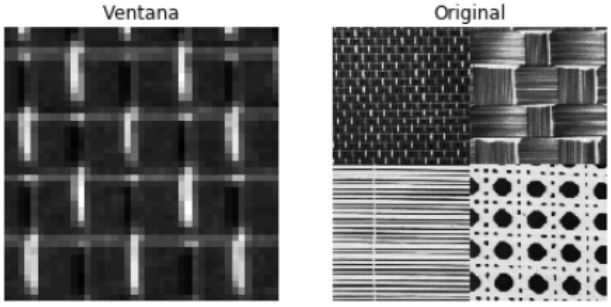


Imagen 2:

```
original = ImgComp[1]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,41,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'imgCompuesta2')
```

```
Tamaño de la original: (320, 320)
Tamaño de la Ventana: (41, 41)
Imagen: imgCompuesta2
```

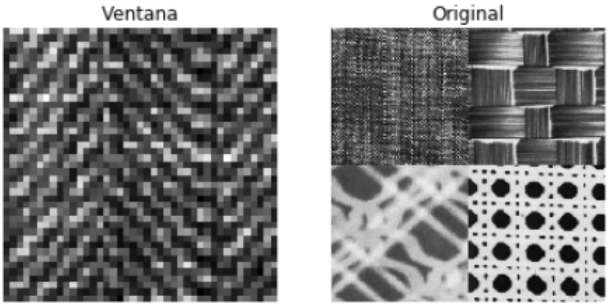
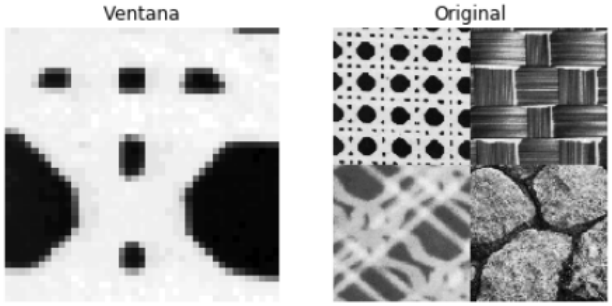


Imagen 3:

```
original = ImgComp[2]
print("Tamanio de la original:",original.shape)
ventana = Genera_ventana(original,41,0,0)
print("Tamanio de la Ventana:",ventana.shape)
Visualiza_ventana(ventana,original,'imgCompuesta3')
```

```
Tamaño de la original: (320, 320)
Tamaño de la Ventana: (41, 41)
Imagen: imgCompuesta3
```



Ya que las obtuvimos, procedemos a almacenarlas y obtener los vectores característicos.

```
Ventanas_Comp = []
for imagen in ImgComp:
    Ventanas_Comp.append(Almacena_ventana(41,imagen)
    )
print("-----------------------------------")
for i,imagen in enumerate(Ventanas_Comp):
    print(f"Contamos con {imagen.shape[0]} ventanas
    para la imagen {names_comp[i]}")
print("-----------------------------------")
```

Dando los siguientes resultados:

```
-----------------------------------
Contamos con 49 ventanas para la imagen
    imgCompuesta1
Contamos con 49 ventanas para la imagen
    imgCompuesta2
Contamos con 49 ventanas para la imagen
    imgCompuesta3
-----------------------------------
```

En este momento comenzamos a obtener los vectores característicos:

```
ImgComp[0].shape
Vect_caract_Comp = []
Vect_caract_Comp.append(Obten_Carac_GLCM(Ventanas
    _Comp[0], 256, 7,[np.pi/4,5*np.pi/4]))
Vect_caract_Comp.append(Obten_Carac_GLCM(Ventanas
    _Comp[1], 256, 7, [np.pi/4,5*np.pi/4]))
Vect_caract_Comp.append(Obten_Carac_GLCM(Ventanas
    _Comp[2], 256, 7, [np.pi/4,5*np.pi/4]))
Vect_caract_Comp[0][0]
```

Observando el Vector de la ventana 1:

```
array([ 4.71480556e+03,  4.55478395e+01,  6.39639040
    e-02,  2.58213782e-03, 5.08147402e-02,
    -1.19434431e-01])
```

Obtenido esto, construimos nuestro DataFrame de características con la siguiente función:

```
def Genera_Data_Set(Ventana_con_vectores):

    contraste = []
    disimilitud = []
    homogeneidad =[]
    ASM = []
    energia = []
    Correlacion = []
    # Vamos a extraer los datos
    for vector in Ventana_con_vectores:
        contraste.append(vector[0])
        disimilitud.append(vector[1])
        homogeneidad.append(vector[2])
        ASM.append(vector[3])
        energia.append(vector[4])
        Correlacion.append(vector[5])

    data = pd.DataFrame({'Homogeneidad':homogeneidad
    ,
                         'ASM':ASM,
                         'Energia':energia,
                         'Correlacion':Correlacion})
    return data
datos_Comp = []
for vector in Vect_caract_Comp:
    datos_Comp.append(Genera_Data_Set(vector))
```

Dando como resultados los siguientes DataFrame:

```
datos_Comp[0].head()
```

	Homogeneidad	ASM	Energia	Correlacion
0	0.063964	0.002582	0.050815	-0.119434
1	0.067737	0.002461	0.049608	-0.118017
2	0.057650	0.002072	0.045518	-0.132420
3	0.080831	0.002422	0.049214	-0.097348
4	0.047688	0.000958	0.030956	0.484563

```
datos_Comp[1].head()
```

	Homogeneidad	ASM	Energia	Correlacion
0	0.037876	0.000731	0.027028	0.169150
1	0.034217	0.000639	0.025287	0.157760
2	0.031091	0.000686	0.026201	0.284907
3	0.035142	0.000658	0.025655	0.294820
4	0.047688	0.000958	0.030956	0.484563

```
datos_Comp[2].head()
```

	Homogeneidad	ASM	Energia	Correlacion
0	0.193654	0.019980	0.141349	0.266047
1	0.201033	0.017560	0.132514	0.277145
2	0.201623	0.018256	0.135115	0.291120
3	0.149387	0.013466	0.116045	0.336562
4	0.047688	0.000958	0.030956	0.484563

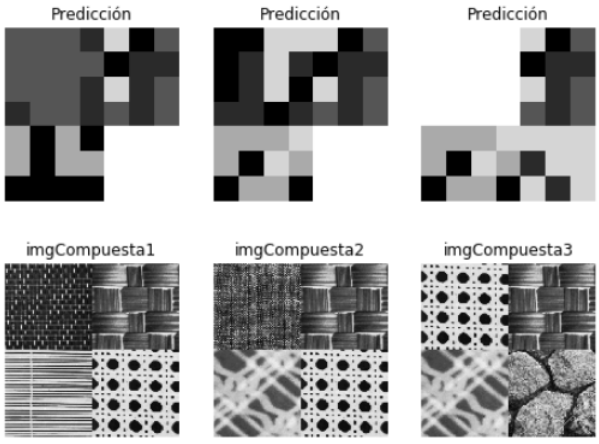
Una vez obtenidas nuestras características, volvemos a realizar nuestras predicciones. Predicciones con KNN:

```
Predict_Images_Comp = []
for dat in datos_Comp:
    Predict_Images_Comp.append(knn.predict(dat))
Predict_Images_Comp[0]
Predict_Images_Comp[1]
Predict_Images_Comp[2]
```

En nuestro caso para una mejor visualización, decidimos plotear dichos resultados de las predicciones para ver el comportamiento que mostraban.

```
def predict_to_image(data):
    size = int((len(data))**(1/2))
    imagen = np.array([data[0:size]])
    i=size
    while i <= len(data) - size:
        aux = np.array([data[i:i+size]])
        imagen = np.append(imagen,aux,axis = 0)
        i+=size
    return imagen
imagen_pred1 = []
for pred in Predict_Images_Comp:
    imagen_pred1.append(predict_to_image(pred))
plt.figure(figsize=(8,6))
for i in range(6):
    plt.subplot(2, 3, i+1)
    if i<3:
        plt.imshow(imagen_pred1[i],cmap='gray')
        plt.title('Prediccion')
        plt.axis('off')
    else:
        muestra_imagen(RutasComp[i-3], subplot=True)
plt.show()
```

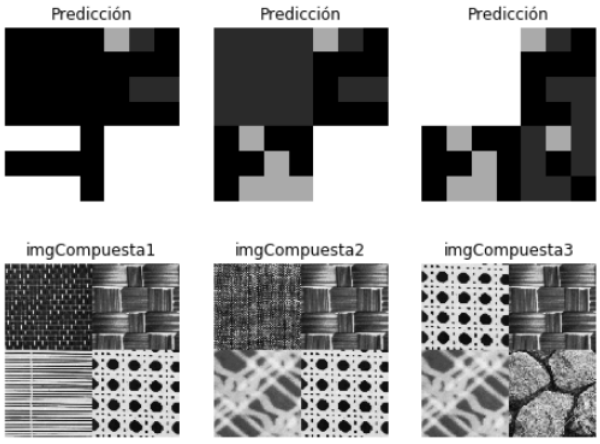
Y así obteniendo el siguiente resultado, mostramos la imagen compuesta y a su vez el ploteo de nuestras predicciones.



Predicciones con Naive Bayes: Realizamos las predicciones, utilziando el entrenamiento anteriormente realizado y plotemos utilizando las funciones creadas en el ejemplo de KNN.

```
Predict_Images_GB = []
for dat in datos_Comp:
    Predict_Images_GB.append(gnb.predict(dat))
imagen_GB = []
for pred in Predict_Images_GB:
    imagen_GB.append(predict_to_image(pred))
    plt.figure(figsize=(8,6))
for i in range(6):

    plt.subplot(2, 3, i+1)
    if i<3:
        plt.imshow(imagen_GB[i],cmap='gray')
        plt.title('Prediccion')
        plt.axis('off')
    else:
        muestra_imagen(RutasComp[i-3], subplot=True)

plt.show()
```

Dando los siguientes resultados:



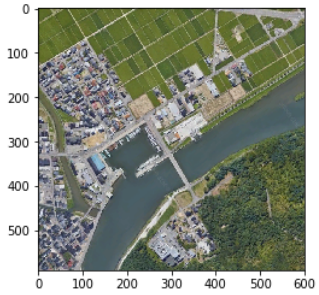
Como podemos observar nuestros algoritmos dieron buenos resultados.

Cabe mencionar que variamos los estadísticos de características, reduciendo la cantidad; pues de esta forma obteníamos una mejor clasificación. Es decir, no valía la pena utilizar todos los estadísticos de segundo orden, pues no mejoraba la clasificación, y es por eso que encontramos la óptima utilizando únicamente las características anteriormente mostradas.

Parte B)

1) Dada una imagen, como la mostrada en la Figura 2 de un satélite óptico, realizar una clasificación por texturas, utilizando superpixeles para la extracción de características estadisticas de 2º orden mediante GLCM. Usted tiene que seleccionar qué características le conviene utilizar para esta imagen.

   Para esta parte de la práctica, vamos a utilizar la siguiente imagen satelital que se nos proporcionó en la práctica anterior:



Dado que esta imagen se encuentra en $RGBA$ (En cuatro canales) realizamos la respectiva conversión, usando $scikit-image$, a RGB:

```
# Leemos la imagen satelital
imagen_satelital = io.imread('imagen_satelital.
    png')
# Convertimos la imagen rgba a rgb
imagen_satelital_rgb = color.rgba2rgb(
    imagen_satelital)
```

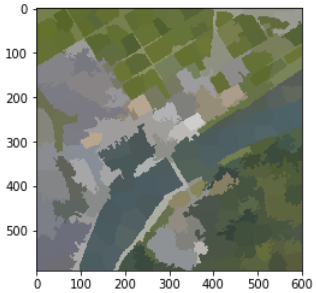
Esta conversión se hace, pues solamente es con este formato es con el que podemos obtener los superpíxeles. Entonces, para obtener estos últimos de la imagen satelital anterior; usamos el método $SLIC$ que vimos en clase, el cual ya está implementado en la biblioteca de $skimage.segmentation$. Cabe destacar que en este método podemos indicar la cantidad aproximada de superpíxeles que queremos conseguir:

```
# El parametro de compacidad intercambia
# similitud de color y proximidad,
# mientras que n\_segments elige el numero
#de centros para kmeans.
# Y el parametro final es sigma , que es el
# tamanio del nucleo gaussiano
# aplicado antes de la segmentacion.

segmentos_slic = slic(imagen_satelital_rgb,
    n_segments=250, compactness=10, sigma=1,
    convert2lab=True)

print(f"Numero de segmentos con SLIC: {len(np.
    unique(segmentos_slic))}")

fig, ax = plt.subplots(1, 1, figsize=(8, 6),
    sharex=True, sharey=True)

ax.imshow(mark_boundaries(imagen_satelital_rgb,
    segmentos_slic))
ax.set_title('SLIC')
ax.set_axis_off()

plt.tight_layout()
plt.show()
```

```
Número de segmentos con SLIC: 179
```

Sin embargo, con este método en Python podemos observar en la imagen anterior que únicamente podemos detectar el conjunto de píxeles que pertenecen a un superpíxel; esto es, no transforma dicho conjunto de superpíxeles de tal forma que todos esos pixeles que pertenecen a un superpíxel, tengan las mismas entradas en RGB. Por lo tanto, para que esto último se cumpliera, realizamos un promedio a los píxeles de cada superpixel para facilitarnos la detección y en este caso sí tendríamos lo que conocemos como superpixeles. La función que calcula lo antes mencionado se muestra a continuación; y además, enseñamos el resultado de obtener ahora sí los superpíxeles de la imagen satelital:

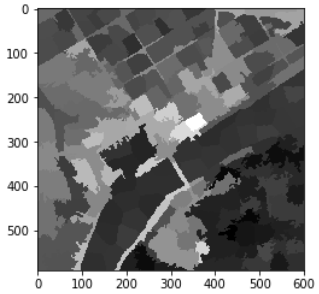
```
# Esta funcion calcula los superpixeles
def mean_image(image,label):
    im_rp = image.reshape((image.shape[0]*image.
    shape[1],image.shape[2]))
    sli_1d = np.reshape(label,-1)
    uni = np.unique(sli_1d)
    uu = np.zeros(im_rp.shape)
    for i in uni:
        loc=np.where(sli_1d==i)[0]
        #print(loc)
        mm=np.mean(im_rp[loc,:],axis=0)
        uu[loc,:]=mm
    img_superpixel = np.reshape(uu,[image.shape
    [0],image.shape[1],image.shape[2]]).astype('
    uint8')
    return img_superpixel

output = mean_image(imagen_satelital_rgb,
    segmentos_slic)

plt.imshow(output)
```



Ahora, basta convertir esta última imagen en escala de grises, ya que solamente de esta manera podemos calcular

las GLCM's para cada superpíxel:

```
output_grey = color.rgb2grey(output)
output_grey = np.uint8(rescale_intensity(
    output_grey, out_range=(0, 255)))
plt.imshow(output_grey, cmap = plt.get_cmap('
    gray'))
```



No obstante, como se habrá notado en las imágenes pasadas, los superpíxeles no tienen una forma rectangular; y este es un requerimiento fundamental para calcular las GLCM's de estos. Es por eso que implementamos las correspondientes funciones para obtener el índice y los valores de los píxeles que pertenecen a cada superpíxel. Posteriormente, a cada superpixel lo inscribimos dentro de un rectángulo sobre el cual le calculamos la GLCM. Esto implicó que necesitaramos una función donde le pasaramos la región del superpíxel y esta función regresara una región rectangular (no necesariamente cuadrada) dentro de ese superpixel. Para procurar que dicha región fuera lo más 'rectangular' posible, utilizamos el Teorema Fundamental de la Aritmética, el cual establece en pocas palabras que cualquier número entero lo podemos factorizar como un producto de números primos. Entonces, a cada superpíxel le almacenábamos todos los píxeles que le pertenecían a él. Luego, le aplicamos el Teorema Fundamental de la Aritmética a la cantidad de píxeles de cada superpixel, para poner colocar a dichos pixeles en un arreglo lo más 'rectangular' posible. Si por ejemplo, la factorización del número N, que equivale a la cantidad de píxeles que pertenecen a un determinado superpixel está dada por n factores, es decir, n números: $N = N_1 \cdot N_2 \cdot ... \cdot N_{n-1} \cdot N_n$ (con $N_1 \leq N_2 \leq ... \leq N_{n-1} \leq N_n$), la longitud de altura del nuevo arreglo rectángular del superpixel está dada por el producto de sus primeros $n-1$: $N_1 \cdot N_2 \cdot ... \cdot N_{n-1}$; mientras que la longitud del ancho estaría dada por el último factor, el cual es N_n

Después de realizar este procedimiento para los 179 superpíxeles, obtenemos de cada superpíxel (ya en su arreglo rectangular) su GLCM; y a esta última le calculamos cinco estadísticas de segundo orden. Estuvimos probando con varias estadísticas de segundo orden; pero para la clasificación con KNN y KMeans (que posteriormente mostraremos) utilizamos las siguientes: **disimilitud, contraste, homogeneidad, energía, correlación**; con las cuales obteníamos mejores resultados que con otras combinaciones de estadísticas de segundo orden.

Finalmente, a los superpíxeles de la imagen en escala de grises le agregamos las estadísticas de segundo orden antes mencionadas de su GLCM. Por lo cual, los superpíxeles ya no cuentan con una única entrada (Que era un número entre 0 y 255, porque la imagen está en escala de grises), sino que cuentan con 1+5=6 entradas; en la que cada entrada posterior a la primera señala la disimilitud; contraste, homogeneidad, energía y correlación respectivamente de su GLCM.

Notemos que podríamos también ponerle a cada píxel, las características de la GLCM al superpíxel que corresponde. Por ejemplo; supongamos que un superpíxel (en escala de grises) está compuesto de estos tres píxeles: [2,2,3,3,4,4], al obtener el superpíxel con la media, sabemos que estos píxeles se llenarían con la media de los seis elementos que conforman el superpíxel y los píxeles quedarían entonces [3,3,3,3,3,3]. Al calcular su GLCM al superpíxel (GLCM a este arreglo [2,2,3,3,4,4]) y al agregarle tres características a cada píxel; que supongamos resultan ser 0.5,5,0.7. Podríamos realizar las futuras clasificaciones con **KNN** y **KMeans** de las siguiente forma: La primera forma consiste agregarle esas características de la GLCM al superpixel que obtuvimos con la media de esos píxeles de tal forma que los pixeles se conviertan a: [[3,0.5,5,0.7],[3,0.5,5,0.7],[3,0.5,5,0.7],[3,0.5,5,0.7], [3,0.5,5,0.7],[3,0.5,5,0.7]], mientras que la segunda manera sería colocar esas características a los píxeles que pertenecen a ese superpíxel (de la imagen original (inicial)) (Y entonces no al superpixel) y de tal forma que quedan así: [[2,0.5,5,0.7],[2,0.5,5,0.7],[3,0.5,5,0.7],[3,0.5,5,0.7], [4,0.5,5,0.7],[4,0.5,5,0.7]]. Haremos las dos formas antes mencionadas, ya que solo cambiaría una parte del código, y para no tener reduncancias, vamos indicar con tres asteriscos y en mayúsculas, qué parte del código es la que cambia.

```
# Esta funcion nos permite obtener los
# indices de los pixeles que pertenecen
# a cada superpixel.
def sp_idx(sos superpixeles a , index, obtenemos
    l = True):
    u = np.unique(s)
    return [np.where(s == i) for i in u]

superpixel_list = sp_idx(segmentos_slic)
# Al hacer superpixel_list[0], obtenemos los
# indices de los pixeles que pertenecen
# al superpixel cero

superpixel      = [imagen_satelital_grey[idx]
    for idx in superpixel_list]
# Aqui obtenemos el valores de los
# pixeles en cada superpixel, por ejemplo,
# Al hacer superpixel[0], obtenemos
# el valores de los pixeles que
#pertenecen al superpixel numero cero

# Estas dos funciones nos permiten aplicar
# el Teorema Fundamental de la Aritmetica
def smallestdivisor(n):
    """Regresa el minimo divisor no trivial de n
    """
    d = 2  # to begin
    while n % d != 0:
```

```
        d = d+1
    return d

def factors(n):
    """Regresa la factorizacion en primos de n
    """
    if n == 1:
        return []  # empty list
    else:
        p = smallestdivisor(n)
    return [p] + factors(n/p)

# En esta parte es en la que convertimos
# los superpixeles a arreglos rectangulares

# Aplanamos todos los pixeles que
# pertenecen al superpixel

for i in range(len(superpixel_valores_orig)):
    superpixel_valores_orig[i] = np.array(
    superpixel_valores_orig[i]).flatten()
    # Convertimos cada superpixel en un arreglo
    cuadrangular
    superpixel_valores_orig[i] = np.reshape(
    superpixel_valores_orig[i],(int(np.prod(
    factors(len(superpixel_valores_orig[i]))
    [:-1])),-1))

# En esta parte calculamos la glcm a
# cada superpixel, y van a estar en orden

props = ['dissimilarity', 'contrast', '
    homogeneity', 'energy', 'correlation']

# lista que guarda las glcm de los
# superpixeles 'rectangulares'
glcm = []
for i in range(len(superpixel_valores_orig)):
        glcm.append(greycomatrix(
    superpixel_valores_orig[i], [1], [0], 256,
    symmetric=True, normed=True))

# estadisticas de segundo
# orden de cada glcm
lf=[]
for i in range(len(glcm)):
    for f in props:
        lf.append(greycoprops(glcm[i], f)[0,0])
lf = np.reshape(lf, (-1,5))
lf = list(lf)

cantidad_superpixeles = list(np.unique(
    segmentos_slic))

# En la siguiente parte del codigo es en
#la que le aniadimos a cada superpixel,las
# caracteristicas de segundo orden de su glcm
### PARA HACER LA CLASIFICACION DE LA
### PRIMERA FORMA HACEMOS LO SIGUIENTE:
output_grey_features = output_grey.copy()

### PARA HACER LA CLASIFICACION DE LA
### SEGUNDA FORMA, HARIAMOS LO SIGUIENTE:
output_grey_features = imagen_satelital_grey.
    copy()

# Indepentientemente de que forma se elige
# La siguiente parte del codigo se mantiene
    igual


output_grey_features = list(output_grey_features
    )

for i in range(len(output_grey_features)):
    output_grey_features[i] = list(np.reshape(
    output_grey_features[i], (len(
    output_grey_features[i]),1)))

for i in range(len(output_grey_features)):
    for j in range(len(output_grey_features[0]))
    :
        output_grey_features[i][j] = [list(
    output_grey_features[i][j])]

for k in range(len(cantidad_superpixeles)):
    for i in range(len(segmentos_slic)):
        for j in range(len(segmentos_slic[0])):
            if segmentos_slic[i][j] == k:
                output_grey_features[i][j].
    append(lf[k])

for i in range(len(output_grey_features)):
    for j in range(len(output_grey_features[0]))
    :
        output_grey_features[i][j] =
    output_grey_features[i][j][0] +
    output_grey_features[i][j][1]
```

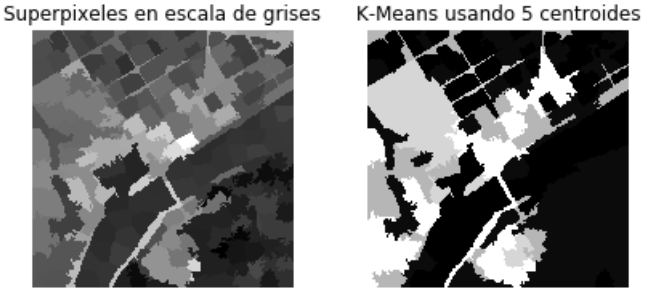
2) Comparar utilizando el algoritmo de clasificación supervisada K-NN y el de clasificación no supervisada KMeans.

   Luego del paso anterior, continuamos con la clasificación con **KMeans**.

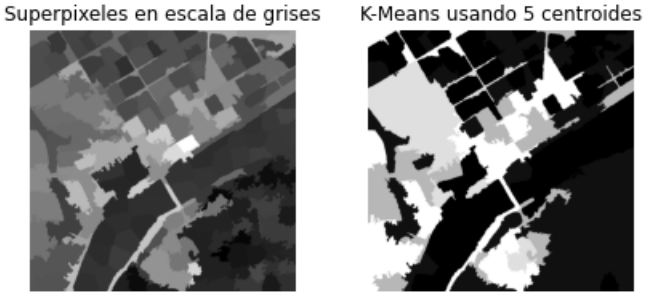
   Aunque en general hay 4 clases en la imagen satelital: agua, pasto, árboles y ciudad (casas), al algoritmo le indicamos que se tengan 5 centroides; ya que con 4 el algoritmo no detectaba adecuadamente la región de la imagen en la que hay árboles.

```
from sklearn import cluster

x, y, z = output_grey_features.shape

image_2d = output_grey_features.reshape(x*y, z)

kmeans_cluster = cluster.KMeans(n_clusters=5)

kmeans_cluster.fit(image_2d)

cluster_centers = kmeans_cluster.
    cluster_centers_

cluster_labels = kmeans_cluster.labels_

output_grey_features_kmeans = cluster_centers[
    cluster_labels].reshape(x, y, z)

# Graficamos por obvias razones la primera
    entrada de cada superpixel
fig, ax = plt.subplots(1, 2,sharex=True, sharey=
    True)

ax[0].imshow(output_grey,  cmap = plt.get_cmap('
    gray'))
ax[0].set_title('Superpixeles en escala de
    grises')
ax[0].set_axis_off()

ax[1].imshow(output_grey_features_kmeans[:,:,0],
     cmap = plt.get_cmap('gray'))
ax[1].set_title('K-Means usando 5 centroides')
ax[1].set_axis_off()

plt.tight_layout()
plt.show()
```

Usando la primera forma, el resultado es el siguiente:



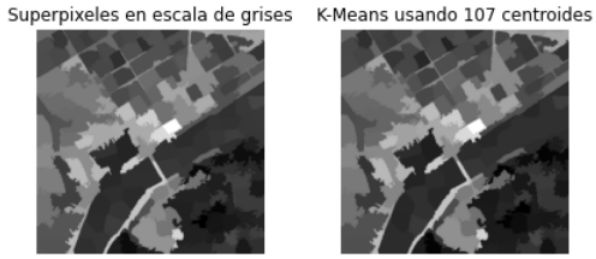
Usando la segunda forma, el resultado es el siguiente:



Sin embargo, si indicamos que el número de centroides sea igual al número de los valores diferentes que toman los superpíxeles, obtenemos la imagen que únicamente tiene los superpíxeles (y no las características de sus GLCM's):

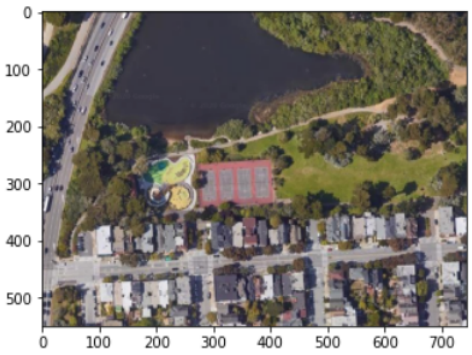
```
kmeans_cluster = cluster.KMeans(n_clusters=107)

kmeans_cluster.fit(image_2d)

cluster_centers = kmeans_cluster.
    cluster_centers_

cluster_labels = kmeans_cluster.labels_

output_grey_features_kmeans = cluster_centers[
    cluster_labels].reshape(x, y, z)

# Graficamos por obvias razones la primera
    entrada de cada superpixel
fig, ax = plt.subplots(1, 2,sharex=True, sharey=
    True)

ax[0].imshow(output_grey,  cmap = plt.get_cmap('
    gray'))
ax[0].set_title('Superpixeles en escala de
    grises')
ax[0].set_axis_off()

ax[1].imshow(output_grey_features_kmeans[:,:,0],
      cmap = plt.get_cmap('gray'))
ax[1].set_title('K-Means usando 107 centroides')
ax[1].set_axis_off()

plt.tight_layout()
plt.show()
```

Usando la primera forma, el resultado es el siguiente:



Usando la segunda forma, el resultado es el siguiente:

Por otro lado, para el algoritmo de **K-NN**, entrenamos y probamos el modelo utilizando la siguiente imagen, la cual la busqé mediante *Google Maps*. Para obtenerla, traté de encontrar un área que tuviera las mismas clases principales que la imagen original: agua, pasto, árboles y ciudad (casas):



Y entonces, a esta imagen le hacemos el mismo procesamiento que a la imagen satelital con la que inicialmente trabajamos, es decir, en resumen hacemos nuevamente lo siguiente:

1.- Convertimos la imagen RGBA a RGB.

2.- A la imagen le calculamos sus superpíxeles.

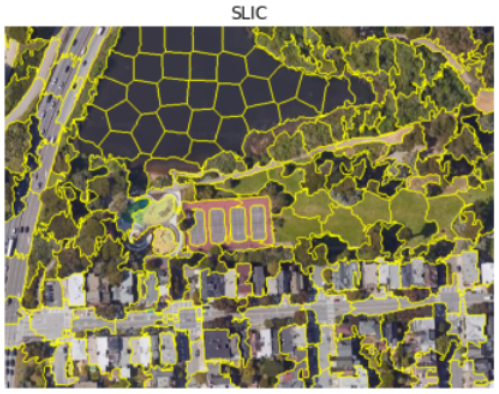
3.- Convertimos la imagen RGB, ya con los superpixeles, a escala de grises.

4.- A cada superpíxel le calculamos su GLMC y de esta última obtenemos las mismas estadísticas de segundo orden: disimilitud, contraste, homogeneidad, energía y correlación; y estas características las anexamos a cada superpíxel.

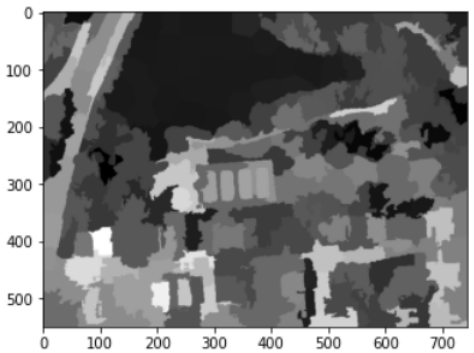
```
# Leemos la nueva imagen satelital
imagen_satelital_prueba = io.imread('
    imagen_satelital_prueba.png')
# Convertimos la imagen rgba a rgb
```

```
imagen_satelital_prueba_rgb = color.rgba2rgb(
    imagen_satelital_prueba)
imagen_satelital_prueba_rgb = np.uint8(
    rescale_intensity(
    imagen_satelital_prueba_rgb, out_range=(0,
    255)))

segmentos_slic_prueba = slic(
    imagen_satelital_prueba_rgb, n_segments=250,
     compactness=10, sigma=1, convert2lab=True)

print(f"Numero de segmentos con SLIC: {len(np.
    unique(segmentos_slic_prueba))}")

fig, ax = plt.subplots(1, 1, sharex=True, sharey
    =True)

ax.imshow(mark_boundaries(
    imagen_satelital_prueba_rgb,
    segmentos_slic_prueba))
ax.set_title('SLIC')
ax.set_axis_off()

plt.tight_layout()
plt.show()
```

```
Número de segmentos con SLIC: 187
```




```
output_prueba = mean_image(
    imagen_satelital_prueba_rgb,
    segmentos_slic_prueba)
output_prueba_grey = color.rgb2grey(
    output_prueba)
output_prueba_grey = np.uint8(rescale_intensity(
    output_prueba_grey, out_range=(0, 255)))
imagen_satelital_prueba_grey = np.uint8(
    rescale_intensity(color.rgb2gray(
    imagen_satelital_prueba_rgb), out_range=(0,
    255)))
plt.imshow(output_prueba_grey, cmap = plt.get_
    cmap('gray'))
```




```
superpixel_list = sp_idx(segmentos_slic_prueba)
superpixel      = [imagen_satelital_prueba_grey[
    idx] for idx in superpixel_list]

superpixel_valores_orig = superpixel.copy()

# Aplanamos todos los pixeles que pertenecen al
    superpixel

for i in range(len(superpixel_valores_orig)):
    superpixel_valores_orig[i] = np.array(
    superpixel_valores_orig[i]).flatten()
    # Convertimos cada superpixel en un arreglo
    cuadrangular
    superpixel_valores_orig[i] = np.reshape(
    superpixel_valores_orig[i],(int(np.prod(
    factors(len(superpixel_valores_orig[i]))
    [:-1])),-1))
# En esta parte calculamos la glcm a cada
    superpixel, y van a estar en orden

props = ['dissimilarity', 'contrast', '
    homogeneity', 'energy', 'correlation']

# left nearest neighbor
glcm = []

for i in range(len(superpixel_valores_orig)):
        glcm.append(greycomatrix(
    superpixel_valores_orig[i], [1], [0], 256,
    symmetric=True, normed=True))

lf=[]
for i in range(len(glcm)):
    for f in props:
        lf.append(greycoprops(glcm[i], f)[0,0])

lf = np.reshape(lf, (-1,5))
lf = list(lf)
for i in range(len(lf)):
    lf[i] = list(lf[i])
cantidad_superpixeles = list(np.unique(
    segmentos_slic_prueba))
### PARA HACER LA CLASIFICACION DE LA
### PRIMERA FORMA HACEMOS LO SIGUIENTE:
output_grey_prueba_features = output_prueba_grey
    .copy()
### PARA HACER LA CLASIFICACION DE LA
### SEGUNDA MANERA, HARIAMOS LO SIGUIENTE:
output_grey_prueba_features =
    imagen_satelital_prueba_grey.copy()

output_grey_prueba_features = list(
    output_grey_prueba_features)

for i in range(len(output_grey_prueba_features))
    :
    output_grey_prueba_features[i] = list(np.
    reshape(output_grey_prueba_features[i], (len
    (output_grey_prueba_features[i]),1)))

for i in range(len(output_grey_prueba_features))
    :
    for j in range(len(
    output_grey_prueba_features[0])):
        output_grey_prueba_features[i][j] = [
    list(output_grey_prueba_features[i][j])]
for k in range(len(cantidad_superpixeles)):
    for i in range(len(segmentos_slic_prueba)):
        for j in range(len(segmentos_slic_prueba
    [0])):
            if segmentos_slic_prueba[i][j] == k:
                output_grey_prueba_features[i][j
    ].append(lf[k])
for i in range(len(output_grey_prueba_features))
    :
    for j in range(len(
    output_grey_prueba_features[0])):
```

```
        output_grey_prueba_features[i][j] =
    output_grey_prueba_features[i][j][0] +
    output_grey_prueba_features[i][j][1]

output_grey_prueba_features = np.array(
    output_grey_prueba_features)
```

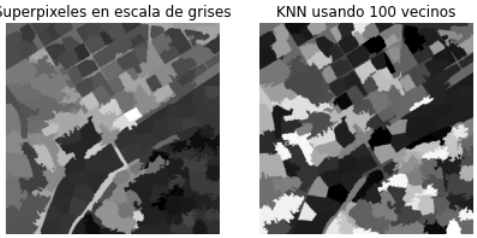
En la próxima parte del código, es en la que ya entrenamos y probamos el modelo con esta última imagen satelital. Usaremos 100 vecinos para este algoritmo:

```
Y = np.reshape(output_grey_features[:,:,0],(
    output_grey_features.shape[0]*
    output_grey_features.shape[1],1))
Y = Y.flatten()
X = np.reshape(output_grey_features[:,:,1:6],(
    output_grey_features.shape[0]*
    output_grey_features.shape[1],5))

X_train, X_test, y_train, y_test =
    train_test_split(X, Y, test_size=0.25,
    random_state=6)

# Creamos el clasificador KNN con 100 vecinos
knn = KNeighborsClassifier(n_neighbors = 100)
# Ajustamos el clasificador a los datos
Y_pred = knn.fit(X_train,y_train).predict(X_test
    )

# Evaluamos el modelo
report = classification_report(y_test, Y_pred)
print(report)

# Omitimos el puntaje de cada etiqueta, pues en
# ambas formas de clasificar, obteniamos en las
    # etiquetas unos (si era de la primera forma
    ) o # ceros (si era de la segunda forma):

### SI USAMOS LA PRIMERA FORMA:
              precision recall f1-score   support
accuracy                           1.00    102163
macro avg      1.00      1.00      1.00    102163
weighted avg   1.00      1.00      1.00    102163

### SI USAMOS LA PRIMERA FORMA:
              precision recall f1-score   support
accuracy                           0.09    102163
macro avg      0.01      0.02      0.02    102163
weighted avg   0.07      0.09      0.07    102163
```

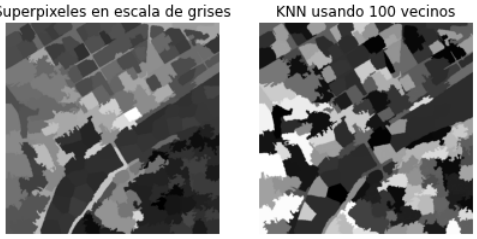
Finalmente, realizamos la predicción de las clases usando la imagen satelital que teníamos en un principio.

```

output_grey_features_knn_pred = knn.predict(np.
    reshape(output_grey_features[:,:,1:6],(
    output_grey_features.shape[0]*
    output_grey_features.shape[1],5)))
output_grey_features_knn_pred = np.reshape(
    output_grey_features_knn_pred,(output_grey.
    shape[0],output_grey.shape[1]))

# Graficamos por obvias razones la primera
    entrada de cada superpixel
fig, ax = plt.subplots(1, 2,sharex=True, sharey=
    True)

ax[0].imshow(output_grey,  cmap = plt.get_cmap('
    gray'))
ax[0].set_title('Superpixeles en escala de
    grises')
ax[0].set_axis_off()

ax[1].imshow(output_grey_features_knn_pred,
    cmap = plt.get_cmap('gray'))
ax[1].set_title('KNN usando 100 vecinos')
ax[1].set_axis_off()

plt.tight_layout()
plt.show()
```

Usando la primera forma, el resultado es el siguiente:



Usando la segunda forma, el resultado es el siguiente:



3) Analizar y explicar los resultados obtenidos

Uno podría pensar que **KMeans** fue el algoritmo que mejor hizo la predicción; pero hay que recordar que con **KNN** utilizamos distintas imágenes satelitales para entrenar y probar el modelo, y para predecir. Aunque no lo colocamos en el reporte, si usábamos la imagen satelital inicial para entrenar y probar el modelo, y para predecir, obteníamos tal cual misma imagen; es decir, el modelo de predicción era perfecto y en este caso **KNN** podría ser mejor que **KMeans**. Es por eso que decidimos utilizar otra imagen con características similares, pero que fuera de distinta forma, y así evitábamos posible sobreajuste del modelo.

Por otro lado, podemos notar que los resultados en las dos formas de clasificar eran muy similares; el cambio más notable fue en **KMeans** cuando teníamos una cantidad de centroides igual a 107, que era el total de posibles valores que tomaba la imagen satelital en escala de grises. En la primera forma obteníamos tal cual la imagen con los superpíxeles, mientras que de la segunda manera conseguimos casi la misma imagen satelital original en escala de grises. En contraparte, **KMeans** cuando teníamos una cantidad de centroides igual a 5 conseguimos parecidos resultados en las dos formas; y esto mismo ocurrió en el caso de **KNN**.

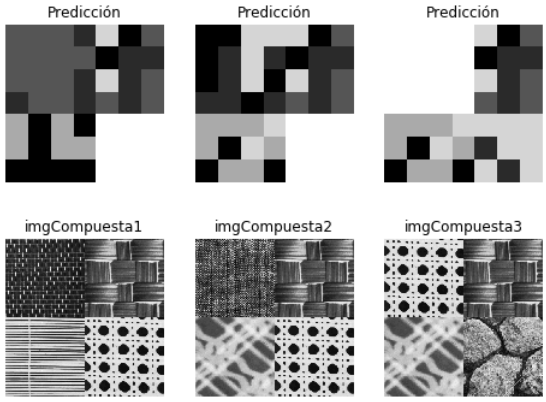
Finalmente, notamos que aunque los puntajes de evaluación del modelo en **KNN** eran muy diferentes entre una forma y la otra (Pues en la primera manera cada etiqueta se clasificaba correctamente, mientras que en la segunda manera la mayoría de las etiquetas se clasificaba erróneamente), obtuvimos texturas muy similares

de resultado. Asimismo, aunque uno pensaría que en el caso en que obtuvimos una clasificación perfecta (con puros unos en la primera forma), deberíamos de obtener la misma imagen, esto no ocurre posiblemente debido a que la imagen que usamos para entrenar y probar el modelo es distinta a la imagen que usamos para predecir.

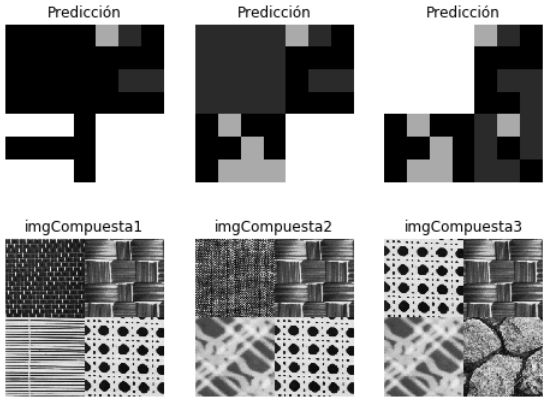
V. Resultados.

Parte A)

Para las pruebas con nuestro modelo de **K-NN** Obtuvimos los siguientes resultados:

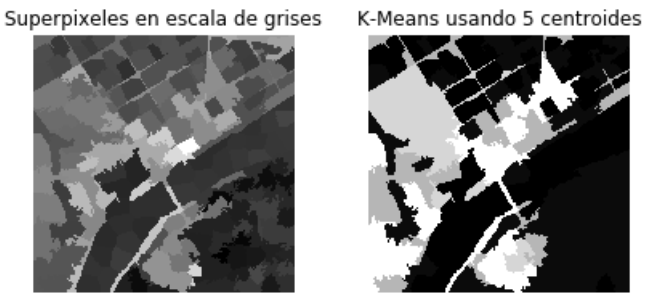


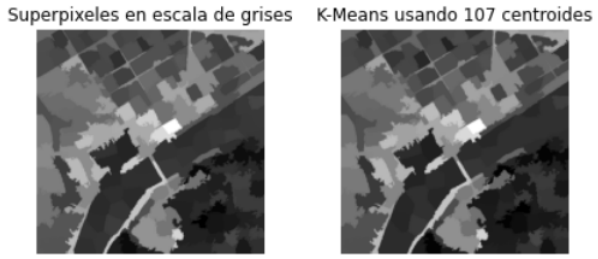
En cambio, para nuestro modelo de **Naive Bayes** obtuvimos lo siguiente:

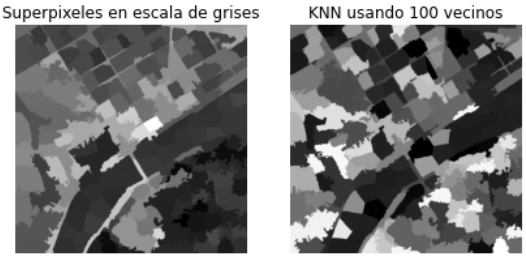


Parte B)

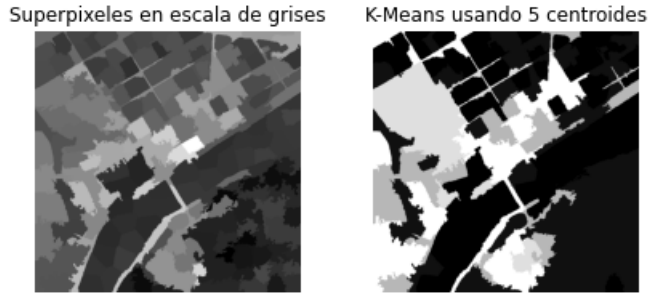
Usando la primera forma, los resultados son los siguiente:

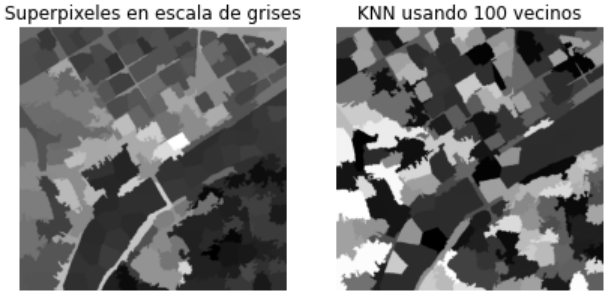






Usando la segunda forma, los resultados son los siguiente:

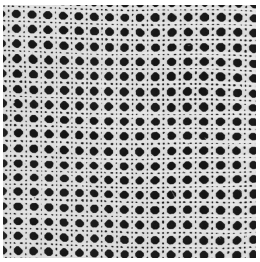




VI. Conclusiones.

En conclusión de la primera parte,a pesar de que nuestros modelos obtuvieron altos resultados dentro de nuestros scores, no son realmente precisos, de hecho falta bastante por ajustar, sin embargo, desde cierto punto se logra apreciar la separación entre las 4 clases debido a las diferencias que existen entre

ellas, además, es importante resaltar que de las clases que mejor logró identificar es la siguiente textura:



Queremos suponer que se debe al seguimiento específico de patrones que existen dentro de ella, es mas fácil de identificarlos.

Por último, en el caso de la segunda parte; logramos ver y utilizar dos formas para resolverla, y aunque obteníamos resultados similares en algunos casos, sí existió una diferencia entre ambas maneras. Más aún, si quisiéramos obtener una clasificación de la imagen con 4 etiquetas: agua, árboles, pasto y ciudad (casas), KMeans nos permite indicarle cuántas clases queremos y este algoritmo es el que nos serviría, ya que como KNN mantenía las etiquetas de los superpíxeles (que eran más que las clases), mantenía esa proporción. Más aún, si hacíamos la prmera forma, en KMeans (con los 107 centroides) obteníamos la imagen satelital con los superpíxeles en escala de grises; mientras que si usábamos la segunda forma, en KMeans (con los 107 centroides) obteníamos la imagen satelital original en escala de grises. Finalmente, si en KNN hacíamos la predicción con la imagen que entrenamos, si lo hacíamos de la prmera forma, obteníamos la imagen satelital con los superpíxeles en escala de grises; mientras que si usábamos la segunda forma, obteníamos la imagen satelital original en escala de grises. (Independiente de que el número de vecinos que pusiéramos como parámetro fuera muy bajo o muy alto).

La técnica de segmentación de imagen asigna una etiqueta a cada píxel en una imagen, de modo que ciertas características similares son compartidas por píxeles con la misma etiqueta. La segmentación simplifica la representación de una imagen en algo que es más significativo y más fácil de analizar.

Los algoritmos supersivados no son mejores que los no supervisados; ni los no supervisados son los mejores que los supervisados. Cada uno depende de la aplicación y de la información que se quiera obtener; y no siempre en ninguno de estos dos tipos se logran los mismos resultados.

VII. Referencias.

- http://bibing.us.es/proyectos/abreproy/11494/fichero/PROYECTO$\%$252FCapitulo+5.pdf
- http://bibing.us.es/proyectos/abreproy/11494/fichero/PROYECTO$\%$252FCapitulo+3.pdf
- https://eprints.ucm.es/56602/1/1138395766-324486$_$ALEJANDRO$_$RODR$\%$C3$\%$8DGUEZ$_$CHAC$\%$C3$\%$93N$_$Memoria$_$TFG$_$2018-2019$_$3940146$_$997003508.pdf
- https://www.aprendemachinelearning.com/clasificar-con-k-nearest-neighbor-ejemplo-en-python/
- https://www.aprendemachinelearning.com/k-means-en-python-paso-a-paso/
- https://rpubs.com/Joaquin$_$AR/233932
- https://rpubs.com/Cristina$_$Gil/SVM
- https://scikit-image.org/docs/dev/auto_examples/segmentation/plot_segmentations.html#id6
- http://gauss.math.luc.edu/greicius/Math201/Fall2012/Lectures/primes2.article.pdf
- https://stackoverflow.com/questions/48885681/get-the-list-of-rgb-pixel-values-of-each-superpixel
- https://dzone.com/articles/cluster-image-with-k-means
- https://stackoverflow.com/questions/41578473/how-to-calculate-average-color-of-a-superpixel-in-scikit-image
- https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html