

# Práctica 3. Caracterización y Clasificación de Texturas

## Parte 1

### (Noviembre, 2021)

1<sup>st</sup> Guerra Silva Erick Iván  
*Ingeniería en Computación*  
*Universidad Nacional Autónoma de México*  
 Ciudad de México, México  
[erickivanguerra0.0@gmail.com](mailto:erickivanguerra0.0@gmail.com)

3<sup>rd</sup> Martínez Gutiérrez Carlos Giovanni  
*Ingeniería en Computación*  
*Universidad Nacional Autónoma de México*  
 Ciudad de México, México  
[cgiovanni@comunidad.unam.mx](mailto:cgiovanni@comunidad.unam.mx)

2<sup>nd</sup> Lázaro Martínez Abraham Josué  
*Ingeniería en Computación*  
*Universidad Nacional Autónoma de México*  
 Ciudad de México, México  
[abrahamlazaro@comunidad.unam.mx](mailto:abrahamlazaro@comunidad.unam.mx)

4<sup>th</sup> Castillo Sánchez Axel  
*Ingeniería Mecatrónica*  
*Universidad Nacional Autónoma de México*  
 Ciudad de México, México  
[axel\\_castillo98@hotmail.com](mailto:axel_castillo98@hotmail.com)

**Resumen**—Este documento presenta los resultados obtenidos para los ejercicios planteados en la práctica para realizar la caracterización y clasificación de texturas. Toda la práctica se realizó en lenguaje Python utilizando Google Collaboratory y se utilizaron módulos como MultinomialNB de Sklearn para la creación de un clasificador de Bayes, Neighbors también de Sklearn para la implementación de un clasificador K-NN, al igual que el módulo cross\_val\_score de Sklearn para medir el desempeño de los resultados obtenidos.

**Índice de Términos**—Procesamiento de Imágenes, caracterización y clasificación, clasificador, texturas, validación, distancia, ángulo, Haralick, segmentación, matriz.

## I. INTRODUCCIÓN

Realice una investigación sobre análisis y caracterización de texturas, métodos de validación y clasificadores K-NN, K-Means, y máquinas de soporte vectorial (SVM).

### Análisis y caracterización de texturas

El análisis de texturas hace referencia a la caracterización de las regiones de una imagen por su contenido de textura. El análisis de texturas intenta cuantificar

las cualidades intuitivas descritas por términos como áspero, suave, sedoso o accidentado en función de la variación espacial en las intensidades de píxeles. En este sentido, la rugosidad o bache se refiere a variaciones en los valores de intensidad, o niveles de gris.

El análisis de texturas puede ser útil cuando los objetos de una imagen se caracterizan más por su textura que por la intensidad, y las técnicas de umbral tradicionales no se pueden utilizar de forma eficaz.[1]

### Métodos de validación

La validación de un método es el proceso para confirmar que el procedimiento analítico utilizado para una prueba en concreto es adecuado para su uso previsto. Los resultados de la validación del método pueden utilizarse para juzgar la calidad, la fiabilidad y la constancia de los resultados analíticos, se trata de una parte integrante de cualquier buena práctica analítica.[2]

### Clasificadores K-NN

El K-NN es un algoritmo de aprendizaje supervisado, es decir, que a partir de un juego de datos inicial su objetivo

## RECONOCIMIENTO DE PATRONES. PRÁCTICA 3

será el de clasificar correctamente todas las instancias nuevas. El juego de datos típico de este tipo de algoritmos está formado por varios atributos descriptivos y un solo atributo objetivo (también llamado clase).

K-NN no genera un modelo fruto del aprendizaje con datos de entrenamiento, sino que el aprendizaje sucede en el mismo momento en el que se prueban los datos de test.[3]

### K-Means

K-means es un algoritmo de clasificación no supervisada (clusterización) que agrupa objetos en k grupos basándose en sus características. El agrupamiento se realiza minimizando la suma de distancias entre cada objeto y el centroide de su grupo o cluster. Se suele usar la distancia cuadrática.[4]

El algoritmo consta de tres pasos:

- Inicialización: una vez escogido el número de grupos, k, se establecen k centroides en el espacio de los datos, por ejemplo, escogiéndolos aleatoriamente.
- Asignación objetos a los centroides: cada objeto de los datos es asignado a su centroide más cercano.
- Actualización centroides: se actualiza la posición del centroide de cada grupo tomando como nuevo centroide la posición del promedio de los objetos pertenecientes a dicho grupo.

### Máquinas de Soporte Vectorial (SVM)

Son un conjunto de algoritmos de aprendizaje supervisado desarrollados por Vladimir Vapnik y su equipo en los laboratorios AT&T.

Estos métodos están propiamente relacionados con problemas de clasificación y regresión. Dado un conjunto de ejemplos de entrenamiento (de muestras) podemos etiquetar las clases y entrenar una SVM para construir un modelo que prediga la clase de una nueva muestra. Intuitivamente, una SVM es un modelo que representa a los puntos de muestra en el espacio, separando las clases a 2 espacios lo más amplios posibles mediante un hiperplano de separación definido como el vector entre los 2 puntos, de las 2 clases, más cercanos al que se llama vector soporte.[5]

## II. OBJETIVOS

El alumno:

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

## III. RESULTADOS

A continuación, se presentan los ejercicios desarrollados en esta práctica.

**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 subdividirla**

en “n” ventanas cuadradas (escoger las dimensiones adecuadas de acuerdo a las texturas que utilice) 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 cooccurrence 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.

Para la generación de la matriz de Haralick a partir de las ventanas de cada imagen se utilizaron las siguientes funciones.

### GLCM

Esta función nos ayudará a crear la matriz de Haralick, recibe 4 parámetros, los cuales son m (matriz inicial), distancia (distancia de movimiento entre los píxeles de la imagen), A (ángulo), scale (Tamaño de la matriz de co-ocurrencia).

```
def GLCM(m,distancia,A,scale):
```

Después de que la función obtiene los cuatro parámetros, se crea la matriz glcm, llenándola con ceros, además de crear la matriz inicial, a partir del ancho y el alto de la imagen a analizar.

Posteriormente se comienza a recorrer la imagen para comenzar a llenar la matriz glcm, esto se llenará dependiendo del ángulo que se escogió, a continuación se muestra el llenado de la matriz glcm, cuando el ángulo es cero.

```
for x in range(width):
    for y in range(height):
        if A==0:#Dirección 0 °
            if y+distancia < height:
                glcm[m[x][y]][m[x][y+distancia]]+=1
```

Al terminar este proceso, la función regresará la matriz glcm, sin embargo en el final se utilizará otra función para poder obtener los estadísticos de segundo orden.

Para obtener los estadísticos de segundo orden de la matriz glcm, se utiliza la función vectorizar.

```
def vectorizar(glcm):
    var = glcm.var()
    mean = glcm.mean()
    std = glcm.std()
    glcm_h = glcm.reshape((-1,1))
    h = entropy(glcm_h, base=2)
    return [mean, var, std, h[0]]
```

Esta función obtendrá los estadísticos de segundo orden de cada ventana a partir de la matriz glcm, los estadísticos que se obtendrán, serán la media, varianza, desviación estándar y la entropía.

## RECONOCIMIENTO DE PATRONES. PRÁCTICA 3

La siguiente función utilizada es la función generar pruebas, esta función nos ayudará a crear las ventanas en cada imagen, los parámetros que recibe esta función son tres, la imagen, la distancia para crear las ventanas y el tamaño de la imagen.

```
def generar_pruebas(imagen, distancia, tamaño):
```

Para obtener el número de píxeles que se utilizará para cada venta, se utilizó el siguiente cálculo.

```
n = distancia*2+1
```

Por lo tanto si se manda una distancia de 2, la ventana que obtendremos será de 5x5, para poder obtener las ventanas, se va recorriendo la matriz de la imagen, a partir del tamaño ingresado, posteriormente, cada ventana se guarda en una lista generada anteriormente.

```
for i in range(0,tamaño-n,distancia):
    for j in range(0,tamaño-n,distancia):
        ventanas.append(imagen[i:i+n,j:j+n])
```

La función regresará las ventanas en una lista.

También se utiliza la función etiquetar, esta función nos ayuda a etiquetar las ventanas con la imagen de entrenamiento a las que corresponden y no mezclar las ventanas de cada clase o imagen asociada.

Todas las funciones descritas anteriormente, se utilizan en una función que hace todo lo descrito anteriormente, esta función se llama sacar pruebas, esta función recibe cuatro parámetros, la url de la imagen, la distancia para crear las ventanas, el ángulo para obtener la matriz glcm y la etiqueta o clase asociada.

Primero se lee la imagen a analizar, a partir de la url que recibe la función.

```
img = io.imread(url)
data = img[:, :, 0]
```

En este caso se lee el canal B de la imagen, al estar la imagen en blanco y negro no existe diferencia entre haber tomado este canal o algún otro ya que todos contienen la misma información en escala de grises.

Esta escala indica el valor de los posibles píxeles en blanco y negro, en este caso 256.

```
escala = 256
```

Posteriormente, generamos las ventanas asociadas a las imágenes de entrenamiento.

```
lista_pruebas = generar_pruebas(data, distancia,
data.shape[0])
```

Una vez con las ventanas creadas, calculamos la matriz glcm de cada ventana y se obtienen los vectores con los estadísticos de segundo orden para cada ventana.

```
lista_pruebas_vectores = [GLCM(ventana,
distancia, angulo, escala) for ventana in
lista_pruebas]
```

Una vez calculados los vectores de estadísticos de segundo orden, los etiquetamos, asociando la clase de la imagen de

entrenamiento a la que corresponden los vectores de estadísticos y posteriormente poder entrenar los clasificadores con estos vectores.

```
lista_pruebas_vectores_etiquetados =
etiquetar(lista_pruebas_vectores, etiqueta)
```

Con esta función obtendremos una lista que tendrá los vectores de prueba, con su respectiva etiqueta.

```
return lista_pruebas_vectores_etiquetados
```

Para la creación de nuestro set de pruebas creamos un for en donde se recorre una tupla, en donde se encuentra el url de la imagen a analizar y su respectiva etiqueta, con ayuda del for se recorre cada imagen y cada etiqueta, para poder obtener ese set de pruebas.

```
texturas_url = [
    #(url, etiqueta)
    ("/content/D6.bmp", "1"),
    ("/content/D64.bmp", "2"),
    ("/content/D49.bmp", "3"),
    ("/content/D101.bmp", "4"),
]
```

Posteriormente, juntamos las pruebas para obtener un único conjunto de entrenamiento.

```
train_set_new = list()
for lista in train_set:
    train_set_new.extend(lista)
train_set = train_set_new
```

### 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 esta práctica ya no es necesario que se implementen de cero los clasificadores por lo que importamos las bibliotecas necesarias para los clasificadores, en este caso es el módulo MultinomialNB y neighbors de Sklearn.

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
```

Posteriormente se aplica un random, para evitar el sobreajuste en nuestro modelo y obtener mejores resultados.

```
np.random.shuffle(train_set)
```

Agrupamos nuestros datos en vector de estadísticos de segundo orden y en etiqueta de textura.

```
X = np.array([x[0] for x in train_set])
y = np.array([x[1] for x in train_set])
```

Aplicamos los clasificadores con los datos de entrenamiento. Una vez obtenidos los conjuntos de entrenamiento, instanciamos los clasificadores y los entrenamos para poder realizar predicciones.

- Clasificador Bayesiano

```
clf = MultinomialNB()
clf.fit(X, y)
print(clf.predict(X[2:3])) # [2]
```

## RECONOCIMIENTO DE PATRONES. PRÁCTICA 3

- Clasificador K-NN

```
neigh = KNeighborsClassifier(n_neighbors=4)
neigh.fit(X, y)
```

Para poder realizar las pruebas con los clasificadores, hicimos la función llamada `generar_prediccion` esta función recibe 6 parámetros, el primer parámetro es la imagen a analizar, el segundo parámetro es la distancia que se utilizará para poder crear las ventanas, el tercer parámetro es el tamaño de la imagen, el cuarto parámetro es el ángulo que se utilizará para crear la matriz glcm, el quinto parámetro, llamado escala, es el número de posibles pixeles que se pueden ocupar, esto guiándonos a través de la escala de grises y el sexto parámetro que recibe, es el clasificador con el que se realizarán las pruebas.

La función realiza los siguientes pasos, primero se genera un variable que contendrá el tamaño de la ventana a analizar.

```
n = distancia*2+1
nuevaimagen = np.zeros_like(imagen)
```

Posteriormente se comenzará a iterar dependiendo el tamaño de la imagen y la distancia, esto para poder obtener la matriz glcm y el vector con los estadísticos de segundo orden.

```
for i in range(0,tamaño-n,distancia):
    for j in range(0,tamaño-n,distancia):
        Vector_Prediccion=GLCM((imagen[i:i+n,j:j+n]),distancia,angulo,escala)
```

Después de obtener el vector con sus respectivos estadísticos de segundo orden, realizamos las predicciones para los vectores de la imagen de prueba.

```
Solucion=clasificador.predict([Vector_Prediccion])
```

Después de aplicar el clasificador, pintaremos los pixeles de la imagen a analizar, con el color correspondiente a la clase que nos arrojó el clasificador, para poder observar el resultado.

```
nuevaimagen[i:i+n,j:j+n]=85*int(Solucion[0])
```

**b. Para cada clasificador escoger y explicar cuál será el método de validación que utilice.**

La validación cruzada o cross validation es un método para obtener la precisión de un algoritmo de clasificación o pronóstico y consiste en dividir el dataset de entrenamiento en k segmentos, donde k-1 segmentos fungirán como data de entrenamiento y el segmento k será para validar el modelo. En cada iteración se irá variando el segmento de prueba y con el resultado de cada uno de estos se obtendrá la precisión general del modelo.

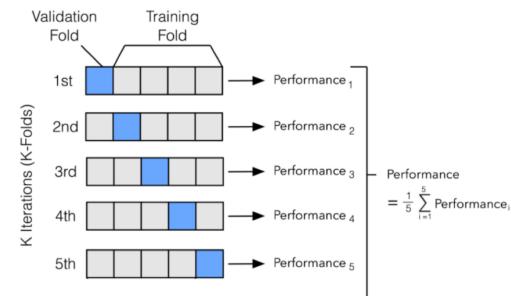


Fig. 1. Diagrama de validación cruzada

La ventaja que ofrece este método de validación con respecto a otros es que la información no permanece "estática", ya que la información de prueba y de entrenamiento va variando en comparación con un modelo que solo entrena con una parte de los datos y otra de pruebas sin variaciones. Por lo tanto, la precisión que ofrecerá la validación cruzada será ligeramente más elevada que la mencionada anteriormente.

Para utilizarla únicamente hay que importar la siguiente librería:

```
from sklearn.model_selection import
cross_val_score
```

En ambas validaciones se considera una k = 4.

### K-NN

```
cv_KNN = cross_val_score(neigh, X, y, cv = 4)
```

### Multinomial Naive Bayes

```
cv_NB = cross_val_score(clf, X, y, cv = 4)
```

Además, notamos que conforme se aumenta la distancia entre los pixeles de la ventana, aumenta la precisión de nuestro modelo. Sin embargo, cuando probamos dicho modelo con alguna imagen nueva, el resultado es deficiente.

Al variar esta distancia llegamos a la conclusión de que existe una relación inversa entre la precisión del modelo y el resultado final. A continuación se puede observar en la tabla las precisiones de los modelos obtenidas con una distancia entre pixeles de la ventana variable. Además, se incluyen las imágenes ya clasificadas como ejemplo.

| Distancia | Precisión Bayes | Precisión K-NN |
|-----------|-----------------|----------------|
| 2         | 39%             | 40%            |
| 4         | 49%             | 52%            |
| 8         | 67%             | 78%            |
| 16        | 75%             | 84%            |

## RECONOCIMIENTO DE PATRONES. PRÁCTICA 3

Imagen a clasificar:

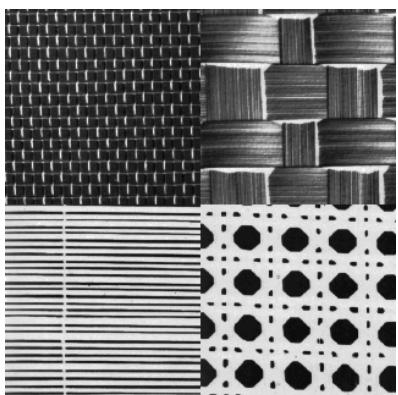


Fig. 2. Imagen de prueba con texturas combinadas.

Resultados del clasificador con distancias variables usando K-NN

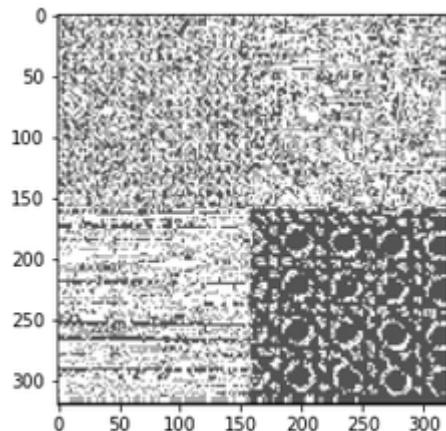


Fig. 3. Texturas obtenidas con Distancia = 2 usando un clasificador K-NN con una precisión del 40%

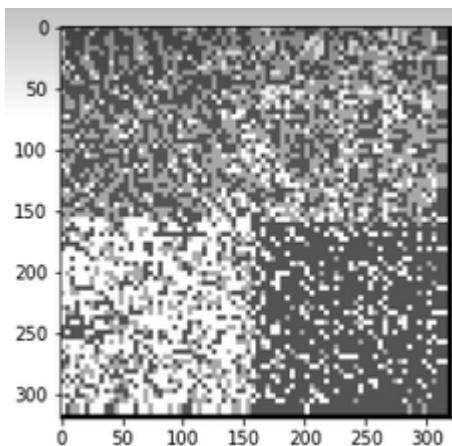


Fig. 4. Texturas obtenidas con Distancia = 4 usando un clasificador K-NN con una precisión del 52 %

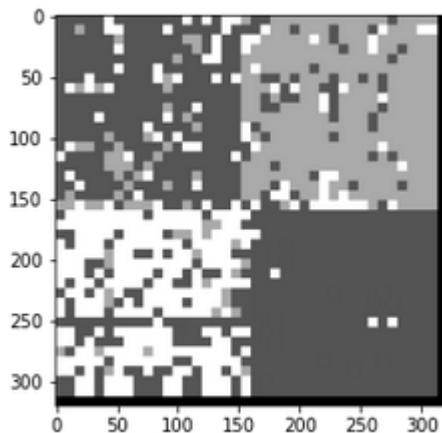


Fig. 5. Texturas obtenidas con Distancia = 8 usando un clasificador K-NN con una precisión del 78%

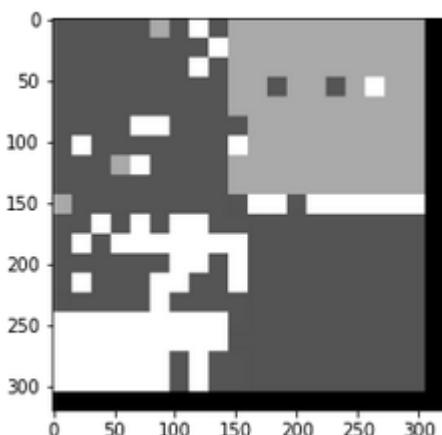


Fig. 6. Texturas obtenidas con Distancia = 16 usando un clasificador K-NN con una precisión del 84%

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.

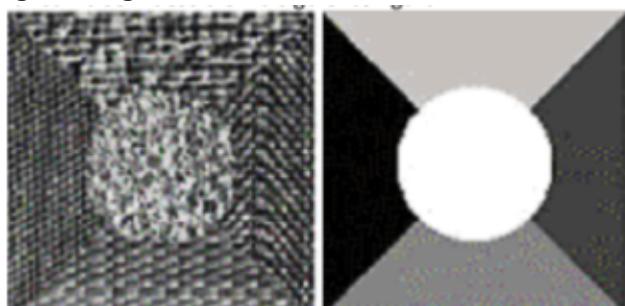


Figura 7. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para el entrenamiento de los clasificadores se utilizaron las siguientes texturas.

## RECONOCIMIENTO DE PATRONES. PRÁCTICA 3

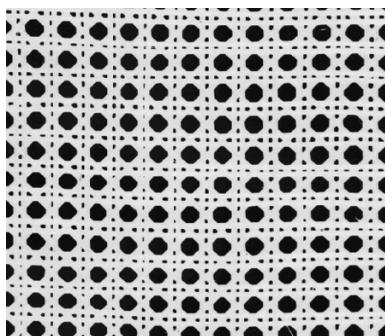


Fig. 8. Textura utilizada en el entrenamiento

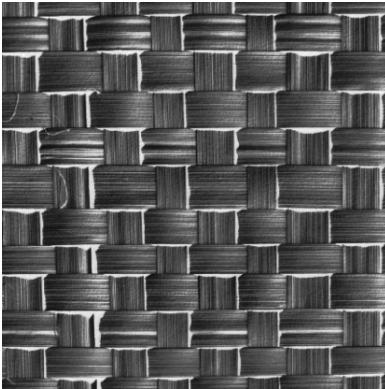


Fig. 9. Textura utilizada en el entrenamiento

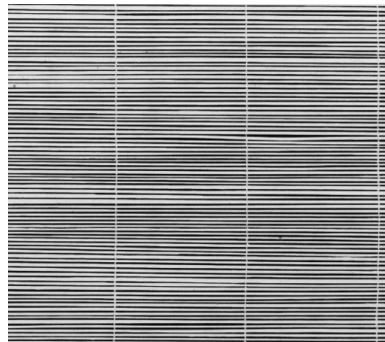


Fig. 10. Textura utilizada en el entrenamiento

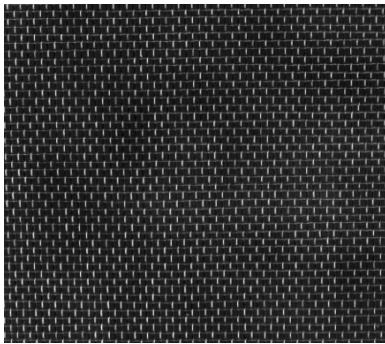


Fig. 11. Textura utilizada en el entrenamiento

### RESULTADOS DEL CLASIFICADOR (MULTINOMIALNB)

A continuación, se presentan algunos de los resultados obtenidos con el clasificador Bayesiano.

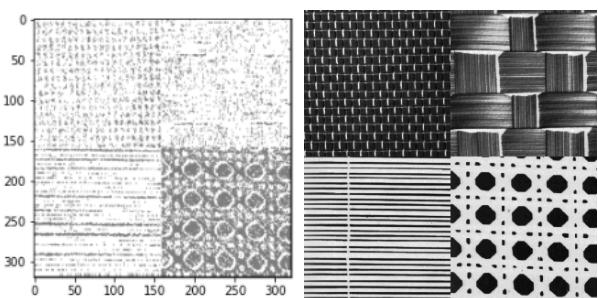


Figura 12. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta imagen compuesta, se obtuvieron buenos resultados, pudiendo distinguir las 4 texturas respecto a la imagen original, cabe destacar que en esta imagen compuesta, las 4 imágenes que la componían estaban dentro del conjunto de entrenamiento.

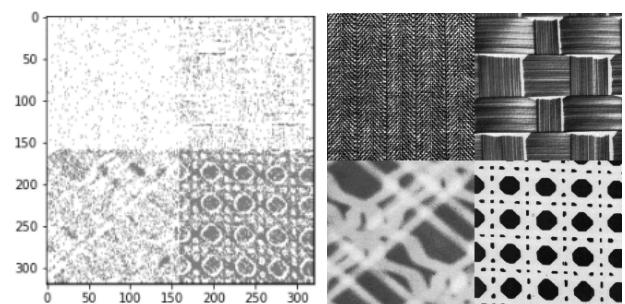


Figura 13. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta segunda prueba, la imagen compuesta contenía una imagen en la esquina inferior izquierda que no era conocida para el conjunto de entrenamiento, a pesar de eso, los resultados obtenidos fueron muy buenos, pudiendo distinguir 4 tonos de grises asociados cada uno a una textura diferente.

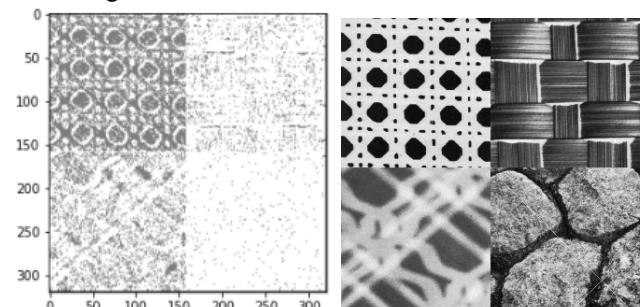


Figura 14. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta imagen compuesta, se obtuvieron buenos resultados, pudiendo distinguir las 4 texturas respecto a la imagen original, cabe destacar que en esta imagen compuesta, solo dos de las imágenes se encontraban en el conjunto de entrenamiento.

## RECONOCIMIENTO DE PATRONES. PRÁCTICA 3

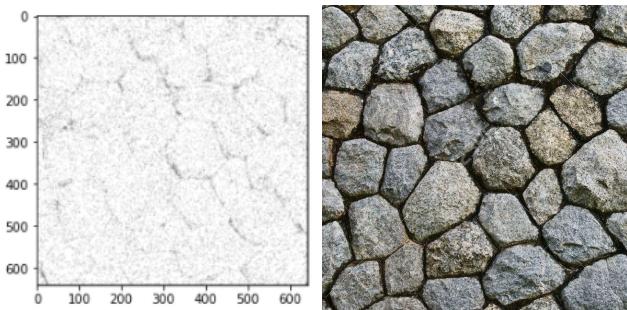


Figura 15. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta imagen, se obtuvieron buenos resultados, ya que se pudo distinguir una única textura respecto a la imagen original, cabe destacar que la textura de esta imagen, no se encontraba en el conjunto de entrenamiento.

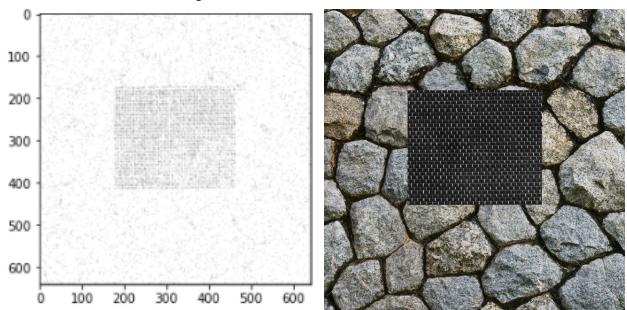


Figura 16. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta última imagen también se obtuvieron buenos resultados, ya que se pudieron distinguir las dos texturas a pesar de que la textura de rocas del fondo no se encontraba en el conjunto de entrenamiento.

### RESULTADOS DEL CLASIFICADOR (K NEIGHBORSCLASSIFIER)

A continuación, se presentan algunos de los resultados obtenidos con el clasificador K-NN.

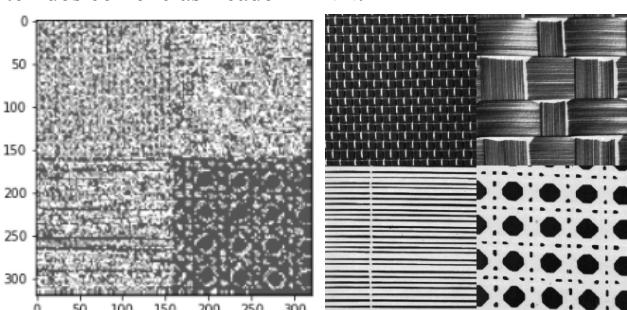


Figura 17. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta imagen compuesta, se obtuvieron buenos resultados, pudiendo distinguir las 4 texturas respecto a la imagen original, cabe destacar que en esta imagen compuesta, las 4 imágenes que la componían estaban dentro del conjunto de entrenamiento. Para esta imagen, las texturas superior derecha e inferior izquierda resultaron muy similares después de clasificar a pesar de que las imágenes eran muy diferentes. Creemos que esto se debe a las líneas horizontales que tienen ambas y pudieron ser identificadas como texturas muy

parecidas.

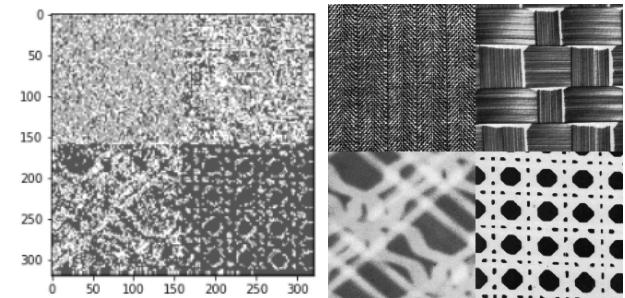


Figura 18. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta segunda prueba, la imagen compuesta contenía una imagen en la esquina inferior izquierda que no era conocida para el conjunto de entrenamiento, a pesar de eso, los resultados obtenidos fueron muy buenos, pudiendo distinguir 4 tonos de grises asociados cada uno a una textura diferente.

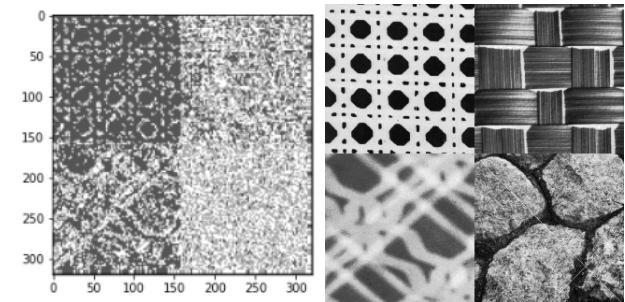


Figura 19. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta imagen compuesta, se obtuvieron buenos resultados, pudiendo distinguir las 4 texturas respecto a la imagen original, cabe destacar que en esta imagen compuesta, solo dos de las imágenes se encontraban en el conjunto de entrenamiento.

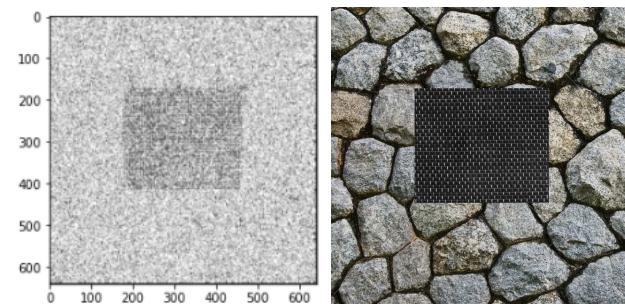


Figura 20. Imagen de textura compuesta y su correspondiente máscara de segmentación

Para esta última imagen también se obtuvieron buenos resultados, ya que se pudieron distinguir las dos texturas a pesar de que la textura de rocas del fondo no se encontraba en el conjunto de entrenamiento.

Con ambos clasificadores se obtuvieron muy buenos resultados, aunque tal vez fue un poco más complicado encontrar los valores de distancia y ángulo adecuados para obtener buenos resultados con el clasificador K-NN.

## RECONOCIMIENTO DE PATRONES. PRÁCTICA 3

### IV. CONCLUSIONES

Esta práctica fue muy interesante, ya que gracias al desarrollo de ésta pudimos obtener un clasificador de texturas, utilizando dos diferentes clasificadores para poder comparar los resultados que se obtenían con cada uno, para poder realizar este clasificador, nos apoyamos de la matriz GLCM, el uso de esta matriz fue muy útil ya que a partir de ella pudimos obtener los estadísticos de segundo orden, primero obteníamos ventanas de nuestra imagen principal, para posteriormente pasar estas ventanas a nuestra función que realizará la matriz GLCM y así poder obtener el vector característico de segundo orden de cada una de las ventanas; este vector contenía la media, la desviación estándar, la entropía y la varianza, este vector nos ayudó a entrenar nuestros dos clasificadores que utilizamos en la práctica, uno de los clasificadores que utilizamos en la práctica fue el clasificador Bayesiano, sin embargo como ya lo habíamos utilizado en una práctica anterior, la implementación no se nos complicó demasiado, por otra parte, el segundo clasificador que utilizamos fue el de KNN, este clasificador recibe los mismos parámetros del primer clasificador, por lo que tampoco tuvimos problemas con su implementación. Después de entrenar ambos clasificadores, comenzamos a realizar distintas pruebas para poder observar cuál de las pruebas nos arrojaba mejores resultados, primero comenzamos variando la distancia a partir de la precisión que obteníamos con cada clasificador, sin embargo, nos dimos cuenta que al aumentar la distancia obteníamos mejor precisión, pero al momento de mostrar la predicción, la imagen resultante no era tan precisa. Otra prueba que realizamos, fue con una textura que no implementamos en el entrenamiento de los clasificadores, sin embargo aunque esta textura era desconocida, los dos clasificadores nos arrojaron buenos resultados, lo mismo pasó con una imagen de prueba en donde se encimaron dos texturas diferentes y una de ellas no era conocida por nuestros clasificadores.

Con ambos clasificadores se obtuvieron muy buenos resultados, aunque tal vez fue un poco más complicado encontrar los valores de distancia y ángulo adecuados para obtener buenos resultados con el clasificador K-NN.

os (accessed Nov. 18, 2021).

[4] kmeans. (s. f.). unioviendo. Recuperado 18 de noviembre de 2021, de [https://www.unioviendo.es/comppnum/laboratorios\\_py/kmeans/kmeans.html](https://www.unioviendo.es/comppnum/laboratorios_py/kmeans/kmeans.html) (accessed Nov. 18, 2021).

[5] colaboradores de Wikipedia. (2021, 14 mayo). Máquinas de vectores de soporte. Wikipedia, la enciclopedia libre. Recuperado 18 de noviembre de 2021, de [https://es.wikipedia.org/wiki/M%C3%A1quinas\\_de\\_vectores\\_de\\_soporte](https://es.wikipedia.org/wiki/M%C3%A1quinas_de_vectores_de_soporte) (accessed Nov. 18, 2021).

[5] colaboradores de Wikipedia. (s/f). Cross validation. Wikipedia, la enciclopedia libre. Recuperado 21 de noviembre de 2021, de [https://en.wikipedia.org/wiki/Cross-validation\\_\(statistics\)](https://en.wikipedia.org/wiki/Cross-validation_(statistics))

### REFERENCIAS

- [1] Roman, V. (2019, 29 abril). “Algoritmos Naive Bayes: Análisis de texturas - MATLAB & Simulink - MathWorks España. (s. f.). mathworks. <https://es.mathworks.com/help/images/texture-analysis-1.html> (accessed Nov. 18, 2021).
- [2] [2] Glosario: Validación de métodos. (s. f.). greenfacts. Recuperado 18 de noviembre de 2021, de <https://www.greenfacts.org/es/glosario/tuv/validacion.htm> (accessed Nov. 18, 2021).
- [3] [3] El algoritmo K-NN y su importancia en el modelado de datos | Blog. (s. f.). Merkle. Recuperado 18 de noviembre de 2021, de <https://www.merkleinc.com/es/es/blog/algoritmo-knn-modelado-datos>

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from skimage import io, feature
from scipy.stats import entropy
from itertools import chain
```

## ▼ Entrenamiento

```
def GLCM(m,distancia,A,scale):
    """
    m: matriz inicial
    distancia: distancia de movimiento
    A: ángulo
    Scale: tamaño de la matriz de co-ocurrencia
    """

    glcm = np.zeros((scale,scale))
    width, height = m.shape
    for x in range(width):
        for y in range(height):
            if A==0:#Dirección 0 °
                if y+distancia < height: # Verificación fuera de límites
                    glcm[m[x][y]][m[x][y+distancia]] += 1

            elif A==45:
                if (x-distancia) >= 0 and 0 < y-distancia:# Verificación fuera de límites
                    glcm[m[x][y]][m[x-distancia][y-distancia]] += 1

            elif A==90: # Verificación fuera de límites
                if x+distancia < width:
                    glcm[m[x][y]][m[x+distancia][y]] += 1

            elif A==135:
                if (y-distancia) >= 0 and x+distancia < width: # Verificación fuera de límites
                    glcm[m[x][y]][m[x+distancia][y-distancia]] += 1

    #Verificar si la matriz es simetrica
    if np.array_equal(glcm, glcm.T) != type('True'):
        glcm=glcm+glcm.T
        #return vectorizar(glcm.astype("int32"))
    #else:
    return vectorizar(glcm.astype("int32"))
```

```
def vectorizar(glcm):
    var = glcm.var()
    mean = glcm.mean()
    std = glcm.std()

    glcm_h = glcm.reshape((-1,1))
    h = entropy(glcm_h, base=2)
```

```

    return [mean, var, std, h[0]]

def generar_pruebas(imagen, distancia, tamaño):
    n = distancia*2+1

    ventanas = list()
    for i in range(0,tamaño-n,distancia):
        for j in range(0,tamaño-n,distancia):
            ventanas.append(imagen[i:i+n,j:j+n])

    return ventanas

def etiquetar(lista_muestras, etiqueta):
    return [(x, etiqueta) for x in lista_muestras]

```

```

def sacar_pruebas(url, distancia, angulo, etiqueta):
    # cargamos la imagen
    img = io.imread(url)
    data = img[:, :, 0]

    # establecemos el valor posible de pixeles
    escala = 256

    # Se generan las pruebas
    lista_pruebas = generar_pruebas(data, distancia, data.shape[0])

    # Generamos las glcm de cada prueba y calculamos los vectores de las pruebas
    lista_pruebas_vectores = [GLCM(ventana, distancia, angulo, escala) for ventana in lista_pruebas]

    # etiquetamos los vectores
    lista_pruebas_vectores_etiquetados = etiquetar(lista_pruebas_vectores, etiqueta)

    return lista_pruebas_vectores_etiquetados

```

```

# para una sola
#pruebas = sacar_pruebas("/content/D1.bmp",2,"1")

```

```

# para generar el train-set hay que hacerlo para todas las texturas
texturas_url = [
    #(url, etiqueta)
    ("/content/D6.bmp","1"), ("/content/D64.bmp","2"), ("/content/D49.bmp","3"), ("/content/D1.bmp","4"),
    ("/content/D100.bmp","5"), ("/content/D101.bmp","6"), ("/content/D102.bmp","7"), ("/content/D103.bmp","8"),
    ("/content/D104.bmp","9"), ("/content/D105.bmp","10"), ("/content/D106.bmp","11"), ("/content/D107.bmp","12"),
    ("/content/D108.bmp","13"), ("/content/D109.bmp","14"), ("/content/D110.bmp","15"), ("/content/D111.bmp","16"),
    ("/content/D112.bmp","17"), ("/content/D113.bmp","18"), ("/content/D114.bmp","19"), ("/content/D115.bmp","20"),
    ("/content/D116.bmp","21"), ("/content/D117.bmp","22"), ("/content/D118.bmp","23"), ("/content/D119.bmp","24"),
    ("/content/D120.bmp","25"), ("/content/D121.bmp","26"), ("/content/D122.bmp","27"), ("/content/D123.bmp","28"),
    ("/content/D124.bmp","29"), ("/content/D125.bmp","30"), ("/content/D126.bmp","31"), ("/content/D127.bmp","32"),
    ("/content/D128.bmp","33"), ("/content/D129.bmp","34"), ("/content/D130.bmp","35"), ("/content/D131.bmp","36"),
    ("/content/D132.bmp","37"), ("/content/D133.bmp","38"), ("/content/D134.bmp","39"), ("/content/D135.bmp","40"),
    ("/content/D136.bmp","41"), ("/content/D137.bmp","42"), ("/content/D138.bmp","43"), ("/content/D139.bmp","44"),
    ("/content/D140.bmp","45")
]

```

```

train_set = [sacar_pruebas(url,2,45,etq) for url,etq in texturas_url]

```

```

# encadenamos las pruebas
train_set_new = list()
for lista in train_set:
    train_set_new.extend(lista)
train_set = train_set_new

```

```

#https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html#sklearn.naive
# Con el clasificador de bayes
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier

np.random.shuffle(train_set)

X = np.array([x[0] for x in train_set])
y = np.array([x[1] for x in train_set])

clf = MultinomialNB()
clf.fit(X, y)
print(clf.predict(X[2:3])) # [2]

['1']

neigh = KNeighborsClassifier(n_neighbors=4)
neigh.fit(X, y)

#print(neigh.predict([[1.1]]))

KNeighborsClassifier(n_neighbors=4)

```

## ▼ Pruebas

```

def generar_prediccion(imagen, distancia, tamaño, angulo, escala, clasificador):

n = distancia*2+1

nuevaimagen = np.zeros_like(imagen)
for i in range(0,tamaño-n,distancia):
    for j in range(0,tamaño-n,distancia):
        Vector_Prediccion=GLCM((imagen[i:i+n,j:j+n]),distancia,angulo,escala)
        Solucion=clasificador.predict([Vector_Prediccion])
        nuevaimagen[i:i+n,j:j+n]=85*int(Solucion[0])
return nuevaimagen

```

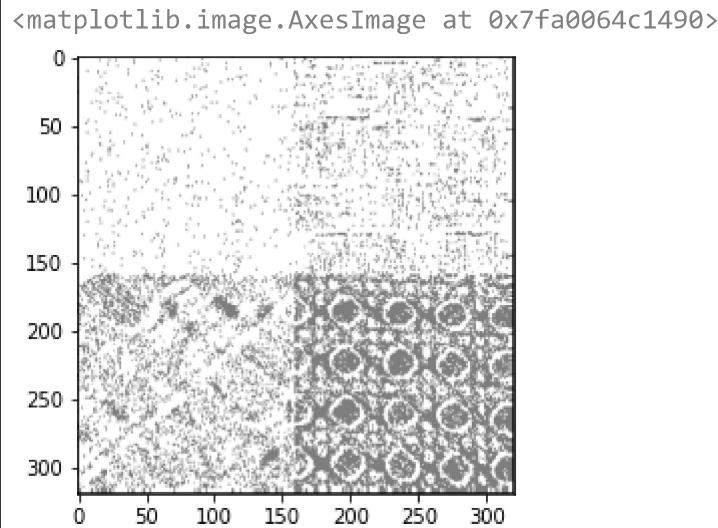
```

data = io.imread("/content/imgCompuesta2.png")
escala = 256
Img_ventanas = generar_prediccion(data, 1, data.shape[0], 45, escala, clf)
Img_ventanas

```

```
array([[170, 170, 170, ..., 170, 170, 0],
       [170, 170, 170, ..., 170, 170, 0],
       [170, 170, 170, ..., 170, 170, 0],
       ...,
       [170, 84, 84, ..., 170, 170, 0],
       [170, 84, 84, ..., 170, 170, 0],
       [ 0,  0,  0, ...,  0,  0,  0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow
plt.imshow(Img_ventanas,cmap='gray')
```

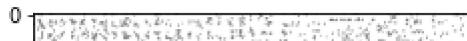


```
data = io.imread("/content/imgCompuesta1.png")
escala = 256
Img1 = generar_prediccion(data, 1, data.shape[0],45,escala, clf)
Img1
```

```
array([[170, 170, 170, ..., 170, 170, 0],
       [170, 170, 170, ..., 170, 170, 0],
       [170, 170, 170, ..., 170, 170, 0],
       ...,
       [170, 170, 170, ..., 170, 170, 0],
       [170, 170, 170, ..., 170, 170, 0],
       [ 0,  0,  0, ...,  0,  0,  0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow
plt.imshow(Img1,cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7fa0130de310>
```

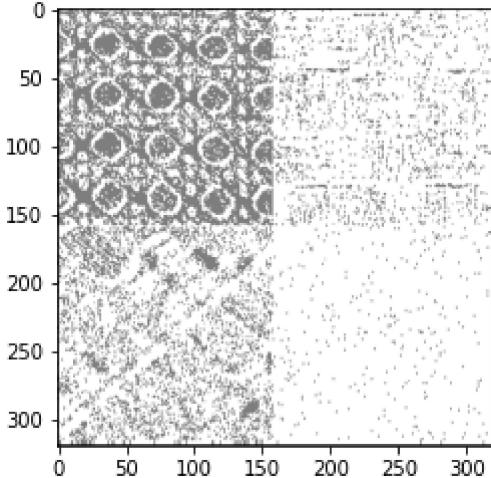


```
data = io.imread("/content/imgCompuesta3.png")
escala = 256
Img3 = generar_prediccion(data, 1, data.shape[0], 45, escala, clf)
Img3
```

```
array([[170,  84,  84, ..., 170, 170,   0],
       [ 84,  84,  84, ..., 170, 170,   0],
       [ 84,  84,  84, ..., 170, 170,   0],
       ...,
       [170,  84,  84, ..., 170, 170,   0],
       [170,  84,  84, ..., 170, 170,   0],
       [  0,   0,   0, ...,   0,   0,   0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow
plt.imshow(Img3, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7fa01130a990>
```



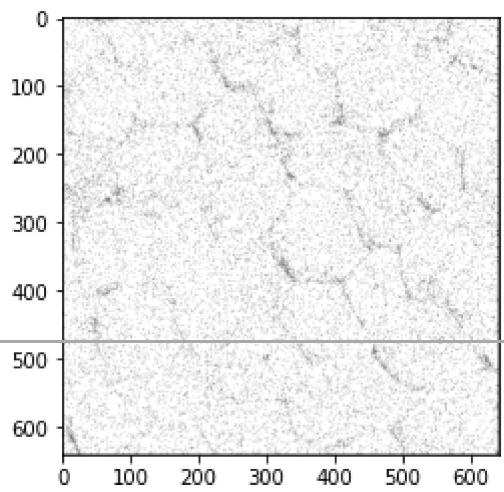
```
data = io.imread("/content/Piedras.png")
escala = 256
Piedras = generar_prediccion(data[:, :, 0], 1, data.shape[0], 45, escala, clf)
Piedras
```

```
array([[170, 170, 170, ..., 84, 84,   0],
       [170, 170, 170, ..., 170, 170,   0],
       [170, 170, 170, ..., 170, 170,   0],
       ...,
       [170, 170, 170, ..., 84, 84,   0],
       [170, 170, 170, ..., 84, 84,   0],
       [  0,   0,   0, ...,   0,   0,   0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow
plt.imshow(Piedras, cmap='gray')
```



<matplotlib.image.AxesImage at 0x7fa00efd7710>



+ Code

+ Text



```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from skimage import io, feature
from scipy.stats import entropy
from itertools import chain

```

## ▼ Entrenamiento

```

def GLCM(m,distancia,A,scale):
    """
    m: matriz inicial
    distancia: distancia de movimiento
    A: ángulo
    Scale: tamaño de la matriz de co-ocurrencia
    """

    glcm = np.zeros((scale,scale))
    width, height = m.shape
    for x in range(width):
        for y in range(height):
            if A==0:#Dirección 0 °
                if y+distancia < height: # Verificación fuera de límites
                    glcm[m[x][y]][m[x][y+distancia]] += 1

            elif A==45:
                if (x-distancia) >= 0 and 0 < y-distancia:# Verificación fuera de límites
                    glcm[m[x][y]][m[x-distancia][y-distancia]] += 1

            elif A==90: # Verificación fuera de límites
                if x+distancia < width:
                    glcm[m[x][y]][m[x+distancia][y]] += 1

            elif A==135:
                if (y-distancia) >= 0 and x+distancia < width: # Verificación fuera de límites
                    glcm[m[x][y]][m[x+distancia][y-distancia]] += 1

    #Verificar si la matriz es simetrica
    if np.array_equal(glcm, glcm.T) != type('True'):
        glcm=glcm+glcm.T
        #return vectorizar(glcm.astype("int32"))
    #else:
    return vectorizar(glcm.astype("int32"))

def vectorizar(glcm):
    var = glcm.var()
    mean = glcm.mean()
    std = glcm.std()

    glcm_h = glcm.reshape((-1,1))
    h = entropy(glcm_h, base=2)

```

```

def vectorizar(glcm):
    var = glcm.var()
    mean = glcm.mean()
    std = glcm.std()

    glcm_h = glcm.reshape((-1,1))
    h = entropy(glcm_h, base=2)

```

```

    return [mean, var, std, h[0]]

def generar_pruebas(imagen, distancia, tamaño):
    n = distancia*2+1

    ventanas = list()
    for i in range(0,tamaño-n,distancia):
        for j in range(0,tamaño-n,distancia):
            ventanas.append(imagen[i:i+n,j:j+n])

    return ventanas

def etiquetar(lista_muestras, etiqueta):
    return [(x, etiqueta) for x in lista_muestras]

```

```

def sacar_pruebas(url, distancia, angulo, etiqueta):
    # cargamos la imagen
    img = io.imread(url)
    data = img[:, :, 0]

    # establecemos el valor posible de pixeles
    escala = 256

    # Se generan las pruebas
    lista_pruebas = generar_pruebas(data, distancia, data.shape[0])

    # Generamos las glcm de cada prueba y calculamos los vectores de las pruebas
    lista_pruebas_vectores = [GLCM(ventana, distancia, angulo, escala) for ventana in lista_pruebas]

    # etiquetamos los vectores
    lista_pruebas_vectores_etiquetados = etiquetar(lista_pruebas_vectores, etiqueta)

    return lista_pruebas_vectores_etiquetados

```

```

# para una sola
#pruebas = sacar_pruebas("/content/D1.bmp",2,"1")

```

```

# para generar el train-set hay que hacerlo para todas las texturas
texturas_url = [
    #(url, etiqueta)
    ("/content/D6.bmp","1"), ("/content/D64.bmp","2"), ("/content/D49.bmp","3"), ("/content/D16.bmp","4"),
    ("/content/D32.bmp","5"), ("/content/D128.bmp","6"), ("/content/D256.bmp","7")
]

#Cambié la distancia a 16
train_set = [sacar_pruebas(url,16,45,etq) for url,etq in texturas_url]

```

```

# encadenamos las pruebas
train_set_new = list()
for lista in train_set:

```

```
train_set_new.extend(lista)
train_set = train_set_new

#https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html#sklearn.naive
# Con el clasificador de bayes
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier

#Para la validación de ambos métodos.
from sklearn.model_selection import cross_val_score

np.random.shuffle(train_set)

X = np.array([x[0] for x in train_set])
y = np.array([x[1] for x in train_set])
```

## Clasificador Bayesiana (Naive Bayes).

```
clf = MultinomialNB()
clf.fit(X, y)
print(clf.predict(X[2:3])) # [2]

['2']
```

## Validación del modelo usando *Cross-Validation*

```
cv_NB = cross_val_score(clf, X, y, cv = 4) #cv es el número de grupos en el cual
                                             #será dividida la data de entrenamiento

print(cv_NB)
print("Media de los 4 resultados: {}" .format(np.mean(cv_NB)))
```

## Validación usando el método *score* de MultinomialNB

```
clf.score(X,y)
```

## Clasificador K vecinos más cercanos(K nearest neighbours)

```
neigh.=KNeighborsClassifier(n_neighbors=4)
neigh.fit(X,·y)

#print(neigh.predict([[1.1]]))

KNeighborsClassifier(n_neighbors=4)
```

## Validación del modelo usando Cross-Validation

```
cv_KNN = cross_val_score(neigh, X, y, cv = 4) #cv es el número de grupos en el cual  
#será dividida la data de entrenamiento  
  
print(cv_KNN)  
print("Media de los 4 resultados: {}" .format(np.mean(cv_KNN)))
```

Validación usando el método score de KNeighborsClassifier

```
neigh.score(X,y)
```

## ▼ Pruebas

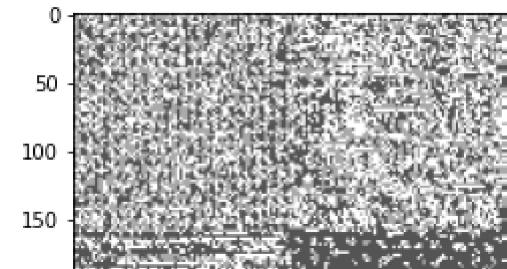
```
def generar_prediccion(imagen, distancia, tamaño,angulo,escala, clasificador):  
  
    n = distancia*2+1  
  
    nuevaimagen = np.zeros_like(imagen)  
    for i in range(0,tamaño-n,distancia):  
        for j in range(0,tamaño-n,distancia):  
            Vector_Prediccion=GLCM((imagen[i:i+n,j:j+n]),distancia,angulo,escala)  
            Solucion=clasificador.predict([Vector_Prediccion])  
            nuevaimagen[i:i+n,j:j+n]=85*int(Solucion[0])  
    return nuevaimagen
```

```
data = io.imread("/content/imgCompuesta1.png")  
escala = 256  
img1n = generar_prediccion(data, 2, data.shape[0],45,escala, neigh)  
img1n
```

```
array([[170, 170, 85, ..., 170, 170, 0],  
       [170, 170, 85, ..., 170, 170, 0],  
       [170, 170, 170, ..., 170, 170, 0],  
       ...,  
       [ 85,  85, 255, ...,  84,  84, 0],  
       [ 85,  85, 255, ...,  84,  84, 0],  
       [ 0,  0,  0, ...,  0,  0, 0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow  
plt.imshow(img1n,cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7fa7180b53d0>
```

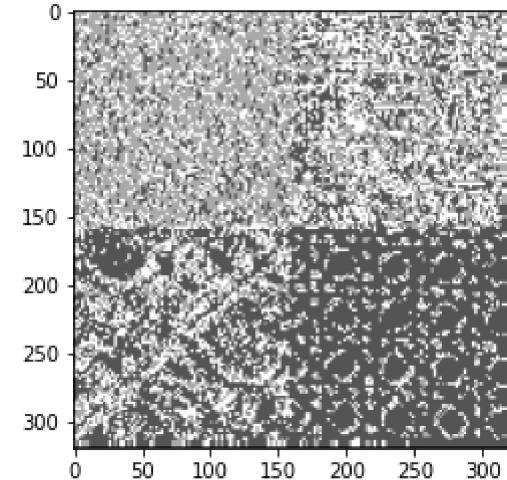


```
data = io.imread("/content/imgCompuesta2.png")
escala = 256
img2n = generar_prediccion(data, 2, data.shape[0], 45, escala, neigh)
img2n
```

```
array([[ 85,  85, 255, ..., 170, 170,   0],
       [ 85,  85, 255, ..., 170, 170,   0],
       [170, 170, 170, ..., 170, 170,   0],
       ...,
       [255, 255, 85, ..., 84, 84,   0],
       [255, 255, 85, ..., 84, 84,   0],
       [  0,   0,   0, ...,   0,   0,   0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow
plt.imshow(img2n, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7fa71804d950>
```

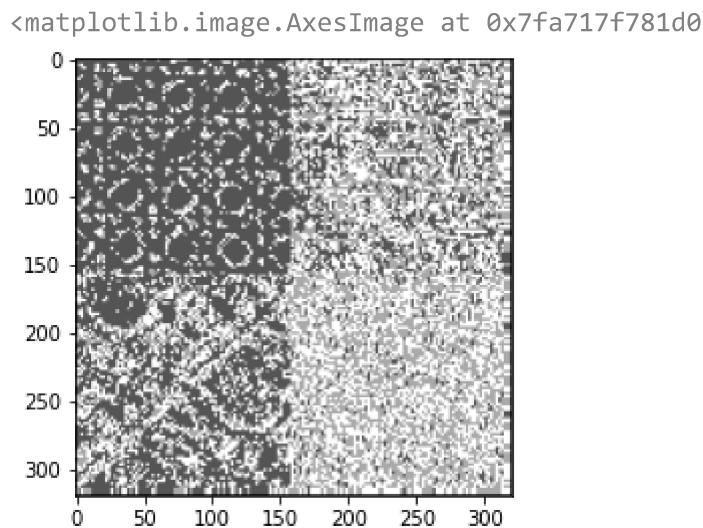


```
data = io.imread("/content/imgCompuesta3.png")
escala = 256
img3n = generar_prediccion(data, 2, data.shape[0], 45, escala, neigh)
img3n
```

```
array([[ 84,  84,  84, ..., 170, 170,   0],
       [ 84,  84,  84, ..., 170, 170,   0],
       [ 84,  84,  84, ..., 170, 170,   0],
       ...,
       [255, 255, 85, ..., 255, 255,   0],
       [255, 255, 85, ..., 255, 255,   0],
       [  0,   0,   0, ...,   0,   0,   0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow
```

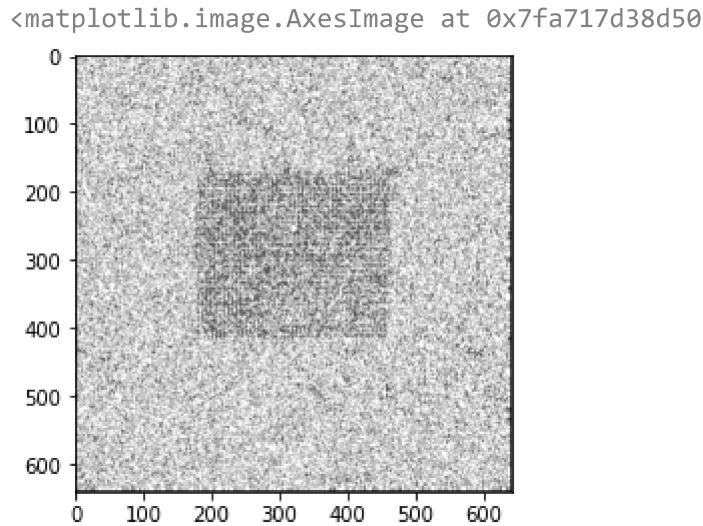
```
plt.imshow(img3n,cmap='gray')
```



```
data = io.imread("/content/PiedrasCompuesta.png")
escala = 256
Piedrasn = generar_prediccion(data[:, :, 0], 2, data.shape[0], 45, escala, neigh)
Piedrasn
```

```
array([[170, 170, 170, ..., 170, 0, 0],
       [170, 170, 170, ..., 170, 0, 0],
       [255, 255, 170, ..., 170, 0, 0],
       ...,
       [170, 170, 170, ..., 170, 0, 0],
       [ 0, 0, 0, ..., 0, 0, 0],
       [ 0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow
plt.imshow(Piedrasn, cmap='gray')
```



```
data = io.imread("/content/PiedrasCompuesta.png")
escala = 256
Piedras = generar_prediccion(data[:, :, 0], 1, data.shape[0], 45, escala, clf)
Piedras
```

```
array([[170, 170, 170, ..., 84, 84, 0],
```

```
[170, 170, 170, ..., 170, 170, 0],  
[170, 170, 170, ..., 170, 170, 0],  
...,  
[170, 170, 170, ..., 170, 170, 0],  
[170, 170, 170, ..., 170, 170, 0],  
[ 0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow  
plt.imshow(Piedras,cmap='gray')
```

