Instituto Politécnico Nacional

Escuela Superior de Cómputo

Proyecto Semestral 2:

Reconocimiento de formas

GRUPO: 6CV1

ALUMNOS:

Ponce Anaya Carlos
García Islas Asael
Rivera Torres Iván Gilberto

PROFESOR:

García Floriano Andrés

Materia:

Inteligencia Artificial

Índice

Introducción

Los momentos invariantes son descriptores matemáticos utilizados en el análisis de imágenes y reconocimiento de patrones para caracterizar y representar la forma, estructura o características de objetos en una imagen. El concepto de momentos proviene del análisis matemático, donde se utilizan para cuantificar la distribución de masa en un objeto. En el procesamiento de imágenes, los momentos se adaptan para describir la distribución de las intensidades de píxeles dentro de una imagen.

- Momentos Invariantes de Hu:

La principal característica de los Momentos Invariantes de Hu es su capacidad para mantener su valor constante frente a transformaciones como rotación, escala y traslación, lo que los hace útiles para la identificación de formas y objetos en imágenes.

- Momentos de Zernike:

Estos momentos son especialmente útiles para caracterizar las propiedades de simetría y asimetría en objetos y formas en imágenes.

Teniendo un poco de conocimiento de toda esta teoría lo que haremos ahora es tener un gran amplio campo de imágenes con figuras como lo son triángulos, círculos, estrellas y lo que haremos es mediante código poder obtener el patrón de nuestras imágenes y al usar los clasificadores, observar cual es el mejor para la tarea que necesitamos.

Desarrollo

Para el desarrollo de esta práctica, adquirimos el banco de imágenes de Kaggle denominado “shapes”, el cual contiene imágenes de formas las cuales son: círculos, cuadrados, estrellas y triángulos, por lo que se nos pide que crear una muestra de 1000 imágenes, el código queda de la siguiente manera:

```
# Librerías.
import os, random, shutil
import matplotlib.pyplot as plt
import cv2
import mahotas
import pandas as pd
import numpy as np
```

Muchas de estas librerías nos ayudaran como lo es para el acceso a los datos como lo es numpy, pandas, así como uno de los más importantes como mahotas y cv2 que nos ofrecerá algoritmos para en análisis de imágenes

```
# Importación del banco de datos 'shapes'.
ruta = "C:/Users/asael/Documents/IA/proyecto/muestras"
# Listado del contenido de 'shapes'.
listado_archivos = os.listdir(ruta)
print(listado_archivos)
```

```
['circle', 'square', 'star', 'triangle']
```

Aquí es donde podremos ver el listado de archivos que se tendrán y las figuras que tendremos para poder utilizar, como lo es el circulo, cuadro, estrella y triangulo

```
# Función para contar el total de imágenes de cada clase.
def conteo_imagenes(datos_directorio):
    for nombre_clases in os.listdir(datos_directorio):
        clases_directorio = os.path.join(datos_directorio, nombre_clases)
        patrones = os.listdir(clases_directorio)
        print(f"La clase {nombre_clases} tiene {len(patrones)} imágenes.")
datos_directorio = ruta
conteo_imagenes(datos_directorio)
```

```
La clase circle tiene 3720 imágenes.
La clase square tiene 3765 imágenes.
La clase star tiene 3765 imágenes.
La clase triangle tiene 3720 imágenes.
```

Vemos la cantidad de imágenes de cada figura

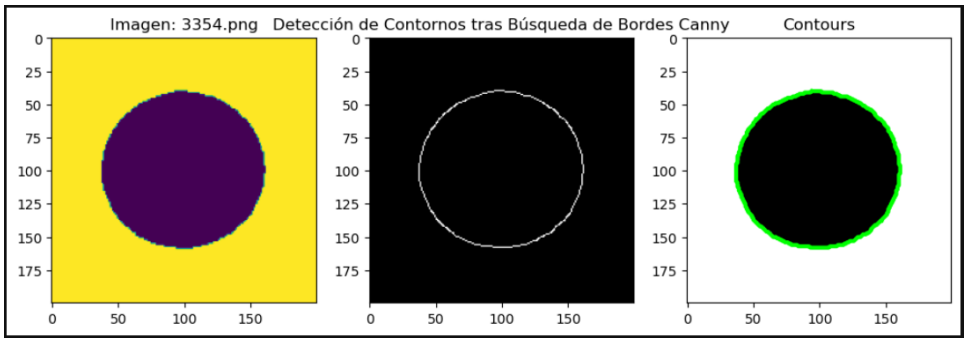
```
# Creación de muestra de 1000 patrones
def crear_muestra(datos_directorio):
    muestra = {}
    for nombre_clases in os.listdir(datos_directorio):
        clases_directorio = os.path.join(datos_directorio, nombre_clases)
        patrones = os.listdir(clases_directorio)
        muestra[nombre_clases] = random.sample(patrones, 250)
    return muestra
datos_directorio = ruta
muestra = crear_muestra(datos_directorio)

total_muestreo = 0
for nombre_clases, patrones in muestra.items():
    print(f"Clase: {nombre_clases} (elegidas aleatoriamente: {len(patrones)})")
    total_muestreo += len(patrones)
    for i in range(2):
        image_path = os.path.join(datos_directorio, nombre_clases, patrones[i])
        image = plt.imread(image_path)
        plt.imshow(image)
        plt.title(f"Imagen: {patrones[i]}")
        plt.show()
print(f'\nTotal de muestras creadas por todas las clases: {total_muestreo}')
```

Esta función tiene como objetivo generar una muestra aleatoria de patrones de imágenes a partir de un directorio de datos que contiene varias clases.

```python
# Función para procesar la muestra y obtener contornos.
def procesar_muestra(datos_directorio, muestra):
    contornos_dict = {}
    for nombre_clases, patrones in muestra.items():
        contornos_clase = []
        for i in range(len(patrones)):
            image_path = os.path.join(datos_directorio, nombre_clases, patrones[i])
            # Imagen en escala de grises.
            gray = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

            # Suiavizado de la imagen.
            gray = cv2.GaussianBlur(gray, (5, 5), 0)
            # Binarización de la imagen.
            ret, binary_image = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)

            # Encontrar los bordes de Canny.
            edged = cv2.Canny(binary_image, 30, 200)
            # Encontrando los contornos.
            contours, hierarchy = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
            # Almacenamiento de los contornos en la lista de contornos de la clase.
            contornos_clase.append(contours)
        # Almacenamiento de la lista de contornos en el diccionario.
        contornos_dict[nombre_clases] = contornos_clase
    return contornos_dict

# Procesar la muestra y obtener contornos.
contornos_dict = procesar_muestra(datos_directorio, muestra)
```

Esta función tiene como objetivo procesar la muestra de imágenes generada anteriormente mediante la detección de contornos en cada imagen. Para cada clase y cada imagen de la muestra, se aplican una serie de operaciones de procesamiento de imágenes para resaltar los contornos presentes en las mismas.

```python
# Visualización de contornos.
def visualizar_contornos(datos_directorio, muestra, contornos_dict):
    for nombre_clases, patrones in muestra.items():
        for i in range(2):
            image_path = os.path.join(datos_directorio, nombre_clases, patrones[i])
            image = plt.imread(image_path)
            # Imagen en escala de grises.
            gray = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

            # Suiavizado de la imagen.
            gray = cv2.GaussianBlur(gray, (5, 5), 0)
            # Binarización de la imagen.
            ret, binary_image = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)

            # Encontrar los bordes de Canny.
            edged = cv2.Canny(binary_image, 30, 200)
            # Encontrando los contornos.
            contours = contornos_dict[nombre_clases][i]

            # Ploteo de la imagen original.
            plt.figure(figsize=(12, 4))
            plt.subplot(131), plt.imshow(image)
            plt.title(f"Imagen: {patrones[i]}")

            # Ploteo de bordes Canny.
            plt.subplot(132), plt.imshow(edged, cmap='gray')
            plt.title("Detección de Contornos tras Búsqueda de Bordes Canny")

            # Ploteo del contorno de la figura.
            plt.subplot(133)
            plt.imshow(image, cmap='gray')
            plt.title("Contours")
            for contour in contours:
                plt.plot(contour[:, 0, 0], contour[:, 0, 1], color='lime', linewidth=3)
            plt.show()

# Visualización de contornos.
visualizar_contornos(datos_directorio, muestra, contornos_dict)
```



Aquí con las funciones que pudimos obtener lo que vamos a ver en un ploteo de las imágenes, el contorno de la figura que estamos utilizando

```python
# Función para el cálculo de los momentos de Hu de los contornos.
def calcular_momentos_hu(contornos_dict):
    momentos_hu_dict = {}
    for nombre_clases, contornos_clase in contornos_dict.items():
        momentos_hu_clase = []
        for contorno in contornos_clase:
            # Conversión del contorno a un formato aceptado por cv2.moments().
            contorno_array = np.vstack(contorno).squeeze()
            # Cálculo de los momentos geométricos y momentos de Hu.
            momentos_hu = cv2.HuMoments(cv2.moments(contorno_array)).flatten()
            momentos_hu_clase.append(momentos_hu)
        momentos_hu_dict[nombre_clases] = momentos_hu_clase
    return momentos_hu_dict

# Llamada a la función.
momentos_hu_dict = calcular_momentos_hu(contornos_dict)

# Visualización de los momentos de Hu.
def visualizar_momentos_hu(momentos_hu_dict):
    for nombre_clases, momentos_hu_clase in momentos_hu_dict.items():
        for i, momentos_hu in enumerate(momentos_hu_clase):
            print(f"Clase: {nombre_clases}, Contorno: {i + 1}")
            print("Momentos de Hu:")
            print(momentos_hu)
            print("\n")

# Muestra los momentos de Hu calculados para cada contorno.
#visualizar_momentos_hu(momentos_hu_dict)
```

La función tiene como objetivo calcular los momentos de Hu para los contornos de las imágenes procesadas

```python
# Función para el cálculo de los momentos de Zernike de los contornos.
def calcular_momentos_zernike(contornos_dict, orden_momento):
    momentos_zernike_dict = {}
    for nombre_clases, contornos_clase in contornos_dict.items():
        momentos_zernike_clase = []
        for contorno in contornos_clase:
            # Conversión del contorno a un formato aceptado por mahotas.
            contorno_array = np.vstack(contorno).squeeze()
            # Cálculo de los momentos de Zernike.
            momentos_zernike = mahotas.features.zernike_moments(contorno_array, orden_momento)
            momentos_zernike_clase.append(momentos_zernike)
        momentos_zernike_dict[nombre_clases] = momentos_zernike_clase
    return momentos_zernike_dict

# Llamada a la función.
orden_momento = 50  # radio de los momentos de Zernike.
momentos_zernike_dict = calcular_momentos_zernike(contornos_dict, orden_momento)

# Visualización de los momentos de Zernike.
def visualizar_momentos_zernike(momentos_zernike_dict):
    for nombre_clases, momentos_zernike_clase in momentos_zernike_dict.items():
        for i, momentos_zernike in enumerate(momentos_zernike_clase):
            print(f"Clase: {nombre_clases}, Contorno: {i + 1}")
            print("Momentos de Zernike:")
            print(momentos_zernike)
            print("\n")

# Muestra los momentos de Zernike calculados para cada contorno.
#visualizar_momentos_zernike(momentos_zernike_dict)
```

Así como en estas imágenes vemos el fragmento de código donde se puede observar los momentos de zernike

```
# Métodos de validación.
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score, KFold


# Clasificadores.
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import GradientBoostingClassifier

# Métricas.
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import recall_score
```

Se importan librerías relacionadas con métodos de validación para evaluar el rendimiento de modelos de aprendizaje automático., también varios algoritmos de clasificación que se pueden utilizar para entrenar modelos de aprendizaje automático. Y por último métricas que se utilizan para evaluar el rendimiento de los modelos de clasificación.

```python
# Conversión de los momentos de Hu y Zernike a arreglos numpy.
def preparar_datos(momentos_dict):
    X = []
    y = []
    for clase, momentos_clase in momentos_dict.items():
        for momentos in momentos_clase:
            X.append(momentos)
            y.append(clase)
    return np.array(X), np.array(y)

# División de momentos y sus respectivas clases.
X_hu, y_hu = preparar_datos(momentos_hu_dict)
X_zernike, y_zernike = preparar_datos(momentos_zernike_dict)
```

```python
print(len(X_hu))
print(len(y_hu))
print('\n')
print(len(X_zernike))
print(len(y_zernike))
```

```
1000
1000


1000
1000
```

```python
from sklearn.preprocessing import LabelEncoder

# Instancia de LabelEncoder.
label_encoder = LabelEncoder()

# Conversión de las etiquetas categóricas a numéricas para y_hu.
y_hu = label_encoder.fit_transform(y_hu)

# Conversión de las etiquetas categóricas a numéricas para y_zernike.
y_zernike = label_encoder.fit_transform(y_zernike)

# Verificamos que se haya hecho exitosamente.
print("Clases para y_hu:")
print(dict(zip(label_encoder.classes_, range(len(label_encoder.classes_)))))
print("\nClases para y_zernike:")
print(dict(zip(label_encoder.classes_, range(len(label_encoder.classes_)))))
```

Esta función se encarga de convertir etiquetas categóricas en valores numéricos. Este proceso es necesario cuando se trabaja con algoritmos de aprendizaje automático que requieren entradas numéricas, como muchos de los clasificadores

```
# División de datos en conjuntos de entrenamiento y prueba (70/30).
X_train_hu, X_test_hu, y_train_hu, y_test_hu = train_test_split(X_hu, y_hu, test_size=0.3, random_state=42)
X_train_zernike, X_test_zernike, y_train_zernike, y_test_zernike = train_test_split(X_zernike, y_zernike, test_size=0.3, random_state=42)
```

```
from sklearn.preprocessing import StandardScaler

# Normalización para momentos de Hu y Zernike para obtención de mejores valores de métricas.

# Momentos de Hu escalados.
scaler_hu = StandardScaler()
X_scaled_hu = scaler_hu.fit_transform(X_train_hu)
X_test_hu = scaler_hu.transform(X_test_hu)

# Momentos de Zernike escalados.
scaler_zernike = StandardScaler()
X_train_scaled_zernike = scaler_zernike.fit_transform(X_train_zernike)
X_test_scaled_zernike = scaler_zernike.transform(X_test_zernike)
```

La normalización es un proceso común en el preprocesamiento de datos que ayuda a asegurar que todas las características tengan una escala similar, lo que será beneficioso para nuestros algoritmos de aprendizaje automático y en este caso mejorar las métricas

```
# Lista de clasificadores.
clfs = {
    'kNN, k=1': KNeighborsClassifier(n_neighbors=1),
    'kNN, k=3': KNeighborsClassifier(n_neighbors=3),
    'kNN, k=5': KNeighborsClassifier(n_neighbors=5),
    'kNN, k=7': KNeighborsClassifier(n_neighbors=7),
    'kNN, k=9': KNeighborsClassifier(n_neighbors=9),
    'Naive Bayes Gaussiano': GaussianNB(),
    'Random Forest': RandomForestClassifier(),
    'Decision Tree': DecisionTreeClassifier(),
    'Gradient Boosting': GradientBoostingClassifier(),
    'SVM Lineal': SVC(kernel='linear'),
    'SVM Gaussiano': SVC(kernel='rbf', C=3, class_weight="balanced"),
    'SVM Polinomial':SVC(kernel='poly', coef0 = 20),
    'Red Neuronal': MLPClassifier(hidden_layer_sizes=(100,), max_iter=1900, random_state=42),
    'Regresión Lineal': LogisticRegression(max_iter=200)
}

def evaluar_clasificador(clf, X_train, y_train, X_test, y_test):
    # Entrenamiento del clasificador.
    clf.fit(X_train, y_train)

    # Predicciones en el conjunto de prueba.
    y_pred = clf.predict(X_test)

    # Evaluación con accuracy.
    accuracy = accuracy_score(y_test, y_pred)
    print(f'Accuracy: {accuracy:.4f}')

    # Matriz de confusión.
    cm = confusion_matrix(y_test, y_pred)
    print('Confusion Matrix:')
    print(cm)
    print('\n')
```

Se crea un diccionario que contiene una lista de clasificadores junto con sus configuraciones específicas. Además de que se proporciona una manera rápida y eficiente de evaluar el rendimiento de cualquier clasificador proporcionado en términos de precisión y matriz de confusión.

```
# Evaluación para momentos de Hu.
print("Evaluación para momentos de Hu:")
for clf_name, clf in clfs.items():
    print(f"Classifier: {clf_name}")
    evaluar_clasificador(clf, X_train_scaled_hu, y_train_hu, X_test_scaled_hu, y_test_hu)

# Evaluación para momentos de Zernike.
print("Evaluación para momentos de Zernike:")
for clf_name, clf in clfs.items():
    print(f"Classifier: {clf_name}")
    evaluar_clasificador(clf, X_train_scaled_zernike, y_train_zernike, X_test_scaled_zernike, y_test_zernike)
```

Se realiza la evaluación de cada clasificador en dos conjuntos de datos diferentes: uno correspondiente a los momentos de Hu y otro a los momentos de Zernike.

```
# k-fold (k=5) para momentos de Hu.
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

for train_index, test_index in kf.split(X_hu, y_hu):
    X_train_hu, X_test_hu = X_hu[train_index], X_hu[test_index]
    y_train_hu, y_test_hu = y_hu[train_index], y_hu[test_index]

    # Normalización para momentos de Hu.
    scaler_hu = StandardScaler()
    X_train_scaled_hu = scaler_hu.fit_transform(X_train_hu)
    X_test_scaled_hu = scaler_hu.transform(X_test_hu)

    # Evaluación para momentos de Hu.
    print("Evaluación para momentos de Hu:")
    for clf_name, clf in clfs.items():
        print(f"Classifier: {clf_name}")
        evaluar_clasificador(clf, X_train_scaled_hu, y_train_hu, X_test_scaled_hu, y_test_hu)
```

```
# k-fold (k=5) para momentos de Zernike.
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

for train_index, test_index in kf.split(X_zernike, y_zernike):
    X_train_zernike, X_test_zernike = X_zernike[train_index], X_zernike[test_index]
    y_train_zernike, y_test_zernike = y_zernike[train_index], y_zernike[test_index]

    # Normalización para momentos de Zernike.
    scaler_zernike = StandardScaler()
    X_train_scaled_zernike = scaler_zernike.fit_transform(X_train_zernike)
    X_test_scaled_zernike = scaler_zernike.transform(X_test_zernike)

    # Evaluación para momentos de Zernike.
    print("Evaluación para momentos de Zernike:")
    for clf_name, clf in clfs.items():
        print(f"Classifier: {clf_name}")
        evaluar_clasificador(clf, X_train_scaled_zernike, y_train_zernike, X_test_scaled_zernike, y_test_zernike)
```

Se realiza ahora con el método de validación k-Fold Cross Validation con k = 5, donde se realiza la normalización de los momentos para obtener un poco mejor valor en las métricas.

```python
# k-fold (k=10) para momentos de Hu.
kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

for train_index, test_index in kf.split(X_hu, y_hu):
    X_train_hu, X_test_hu = X_hu[train_index], X_hu[test_index]
    y_train_hu, y_test_hu = y_hu[train_index], y_hu[test_index]

    # Normalización para momentos de Hu.
    scaler_hu = StandardScaler()
    X_train_scaled_hu = scaler_hu.fit_transform(X_train_hu)
    X_test_scaled_hu = scaler_hu.transform(X_test_hu)

    # Evaluación para momentos de Hu.
    print("Evaluación para momentos de Hu:")
    for clf_name, clf in clfs.items():
        print(f"Classifier: {clf_name}")
        evaluar_clasificador(clf, X_train_scaled_hu, y_train_hu, X_test_scaled_hu, y_test_hu)
```

```python
# k-fold (k=10) para momentos de Zernike.
kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

for train_index, test_index in kf.split(X_zernike, y_zernike):
    X_train_zernike, X_test_zernike = X_zernike[train_index], X_zernike[test_index]
    y_train_zernike, y_test_zernike = y_zernike[train_index], y_zernike[test_index]

    # Normalización para momentos de Zernike.
    scaler_zernike = StandardScaler()
    X_train_scaled_zernike = scaler_zernike.fit_transform(X_train_zernike)
    X_test_scaled_zernike = scaler_zernike.transform(X_test_zernike)

    # Evaluación para momentos de Zernike.
    print("Evaluación para momentos de Zernike:")
    for clf_name, clf in clfs.items():
        print(f"Classifier: {clf_name}")
        evaluar_clasificador(clf, X_train_scaled_zernike, y_train_zernike, X_test_scaled_zernike, y_test_zernike)
```

Se realiza ahora con el método de validación k-Fold Cross Validation con k = 10, donde se realiza la normalización de los momentos para obtener un poco mejor valor en las métricas.

Por último, dejamos en esta sección el código para la generación de los momento invariantes el cual esta en el siguiente link de github:

https://github.com/Ch4rlesPA/MomentosInvariables_Equipo3_6CV1

Conclusión

Este proyecto sobre reconocimiento de formas mediante los momentos de Hu y Zernike ha sido un ejercicio integral que aborda el procesamiento de imágenes, la extracción de características y la aplicación de algoritmos de aprendizaje automático. A lo largo de este trabajo, se han utilizado diversas herramientas y técnicas para construir un sistema robusto de reconocimiento de formas.

Se aplicó un procesamiento de imágenes que incluyó suavizado, binarización y detección de contornos. Estos pasos fueron esenciales para resaltar las características relevantes de las formas y preparar los datos para su análisis además de los procesos importantes como la normalización de datos para garantizar una escala consistente y la transformación de etiquetas categóricas a numéricas para facilitar la aplicación de algoritmos de aprendizaje automático.

Se ha demostrado la viabilidad y eficacia de utilizar momentos de Hu y Zernike en conjunto con algoritmos de aprendizaje automático para el reconocimiento de formas en imágenes. Los resultados obtenidos proporcionan una base sólida para futuras mejoras y aplicaciones en campos como la visión por computadora

Aquí dejamos las conclusiones de que modelo fue el mejor:

```
Mejor modelo para momentos de Hu: kNN, k=5
Mejor accuracy para momentos de Hu: 0.965
Confusion Matrix para momentos de Hu:
[[50  0  0  0]
 [ 2 46  2  0]
 [ 0  0 50  0]
 [ 0  0  3 47]]


Mejor modelo para momentos de Zernike: SVM Polinomial
Mejor accuracy para momentos de Zernike: 0.955
Confusion Matrix para momentos de Zernike:
[[50  0  0  0]
 [ 1 45  0  4]
 [ 0  1 48  1]
 [ 1  1  0 48]]
```