



PROBABILIDAD CONDICIONAL

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

PROBABILIDAD CONDICIONAL

Se selecciona al azar dos bolas sin reemplazo de una urna que contiene 4 blancas y 8 negras.

1. Calcular la probabilidad de que ambas sean blancas.
2. Calcular la probabilidad de que la segunda bola sea blanca.

Eventos:

- A: La primera bola es blanca.
 - B: La segunda bola es blanca.
 - C: Ambas bolas son blancas.
1. Calcular la probabilidad de que ambas sean blancas.

Tenemos calcular la probabilidad del evento C. Como el evento C es el caso donde el evento A y el evento B suceden, entonces $C = A \cap B \rightarrow P(C) = P(A \cap B)$

Sabemos que:

$$\begin{aligned} \cdot P(A|B) &= \frac{P(A \cap B)}{P(B)} \\ \cdot P(A \cap B) &= P(B)P(A|B) \\ \cdot P(B \cap A) &= P(B) + P(A) - P(B \cup A) \end{aligned}$$

Entonces:

$$\begin{aligned} P(C) &= P(A \cap B) \\ P(C) &= P(A)P(B|A) \end{aligned}$$

$$\bullet \quad P(A) = \frac{4}{12} \quad P(B|A) = \frac{3}{11}$$

$$\begin{aligned} P(C) &= \frac{4}{12} \cdot \frac{3}{11} \\ P(C) &= \frac{1}{11} \end{aligned}$$

2. Calcular la probabilidad de que la segunda bola sea blanca.

Sabemos que:

$$\begin{aligned}P(\bar{A} \cap B) &= P(B) - P(A \cap B) \\P(A) &= P(\bar{B} \cap A) + P(A \cap B) \\P(A) &= P(\bar{B})P(A|\bar{B}) + P(B)P(A|B)\end{aligned}$$

Traducido al caso de nuestro problema:

$$P(B) = P(A)P(B|A) + P(\bar{A})P(B|\bar{A})$$

$$\bullet \quad P(A) = \frac{4}{12} \quad P(B|A) = \frac{3}{11} \quad P(\bar{A}) = \frac{8}{12} \quad P(B|\bar{A}) = \frac{4}{11}$$

$$\begin{aligned}P(B) &= \frac{4}{12} \cdot \frac{3}{11} + \frac{8}{12} \cdot \frac{4}{11} \\P(B) &= \frac{1}{3}\end{aligned}$$



FUNCIÓN NORMAL

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

TAREA 2: FUNCIÓN NORMAL

Hacer un programa en python que evalúe la función de densidad de probabilidad normal, dados los parámetros μ y σ . (graficar la función).

1. Generar 2 Gaussianas (traslapadas) con diferente media (μ_1, μ_2), y diferente varianza (σ_1, σ_2) para modelar la probabilidad condicional $P(x|w_1)$ y $P(x|w_2)$.
2. Dado un punto x , clasificar dicho punto usando la siguiente regla de decisión. Usar probabilidades a priori iguales $p(w_1) = p(w_2) = \frac{1}{2}$:
Elige w_1 si $p(x|w_1) * p(w_1) > p(x|w_2) * p(w_2)$, de lo contrario elige w_2 .

Resultados

1. Generar y graficar 2 Gaussianas con diferente μ y σ en python.

Para generar 2 gaussianas de densidad de probabilidad, tenemos que evaluar la función normal.

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{\sigma^2}}$$

Adelantandonos al punto 2, se va a crear una clase, la cual, va a calcular una función de densidad normal, una función discriminante y la probabilidad de un punto dado.

```
class classifier:
    # Se inicializan las variables mu y sigma
    def __init__(self, mu, sigma):
        self.mu = mu
        self.sigma = sigma

    # Se calcula la funcion normal para un valor x con los parametros mu y sigma dados
    def normal(self, x):
        b = (-1/2) * ((x-self.mu)**2) / (self.sigma**2)
        P = (1/(np.sqrt(2*np.pi)*self.sigma)) * np.exp(b)
        return P

    # Se calcula la funcion discriminante para cualquier valor de x utilizando la
    # probabilidad a priori y la de la funcion normal
    def disc(self, P, prior):
        logP = np.log(P)
        disc = logP + np.log(prior)
        return disc

    # Se calcula la funcion discriminante para un valor especifico de x
    def final(self, x_test, prior):
        P = self.normal(x_test)
        discriminant = self.disc(P, prior)
        return discriminant
```

Para este primer punto, se utiliza el primer metodo **normal()**, el cual, calcula la función de densidad normal calculando primero el exponente.

$$b = -1/2 \frac{(x - \mu)^2}{\sigma^2}, \quad P = \frac{1}{\sqrt{2\pi}\sigma} e^b$$

Seguido, se piden los valores de μ y σ para las gaussianas.

```
x = np.linspace(-5, 5, 1000) # Se define el rango de valores de x
mu1 = float(input("Ingrese mu 1: ")) # Se solicitan los parametros] para la primera
    ↪ media
sigma1 = float(input("Ingrese sigma 1: ")) # Se solicitan los parametros para la
    ↪ primera desviacion
mu2 = float(input("Ingrese mu 2: ")) # Se solicitan los parametros para la segunda
    ↪ media
sigma2 = float(input("Ingrese sigma 2: ")) # Se solicitan los parametros para la
    ↪ segunda desviacion
```

Con estos datos, se inicializa la clase y se grafican las gaussianas.

```
#Se crean los clasificadores para cada funcion
classification1 = classifier(mu1, sigma1)
classification2 = classifier(mu2, sigma2)

# Se calculan las probabilidades para cada funcion
P1 = classification1.normal(x)
P2 = classification2.normal(x)

# Se grafican las dos distribuciones normales
plt.figure(figsize=(10, 5))
plt.title("Funcion Normal")
plt.plot(x, P1, label=f"Media: {mu1}, Desviacion: {sigma1}. Clase 1")
plt.plot(x, P2, label=f"Media: {mu2}, Desviacion: {sigma2}. Clase 2")
plt.legend()
plt.xlabel("x")
plt.ylabel("P(x)")
```

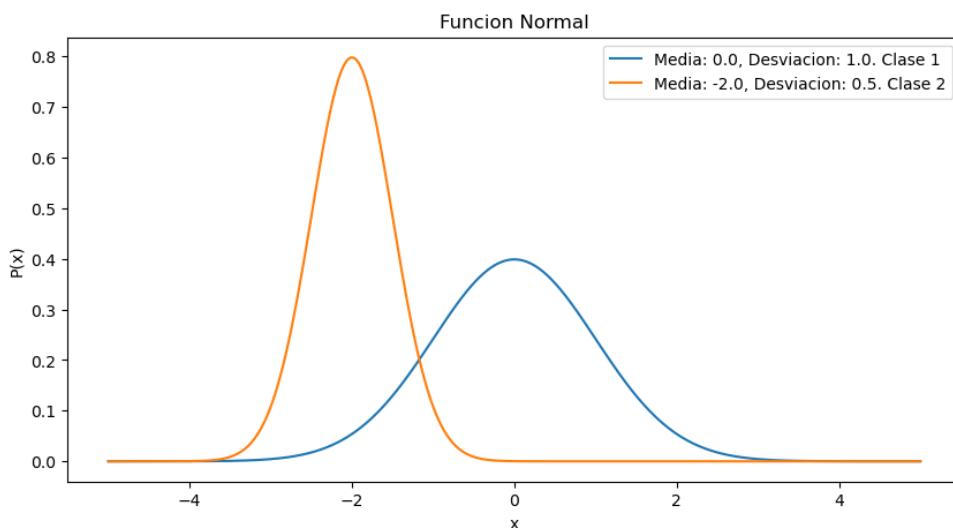


Figure 1: Distribución normal

2. Clasificar un punto x usando la regla de decisión.

Para este segundo punto, se van a utilizar el resto de los métodos de la clase. `disc()` va a calcular la función discriminante utilizando la siguiente ecuación:

$$g_i(x) = \ln(P(x|w_i)) + \ln(P(w_i))$$

$P(x|w_i)$ ya lo calculamos con el método `normal()` y $P(w_i)$ es la probabilidad a priori, la cual es un valor establecido de 0.5. Entonces, solo queda calcular los logaritmos de cada uno y sumarlos. Por último, el método `final()` va a calcular la función normal para un punto x dado, para luego calcular la discriminante. Ahora que tenemos todos los datos necesarios para hacer la clasificación, se solicita el punto x a clasificar, se establece la probabilidad a priori y se calcula la discriminante del punto con respecto a (μ_1, σ_1) y (μ_2, σ_2) .

```
x_test = float(input("Ingrese x: ")) # Se solicita el valor de x específico para
    ↪ calcular la función discriminante
prior = 0.5 # Se define la probabilidad a priori (1/2)

# Se calcula la función discriminante para el valor de x con las dos distribuciones
disc1 = classification1.final(x_test, prior)
disc2 = classification2.final(x_test, prior)

# Se grafica el punto x con respecto a las distribuciones
plt.plot([x_test, x_test], [0, 1], 'r--', label=f"x = {x_test}")
plt.legend()
plt.show()

print(f'Función discriminante para x = {x_test} en la clase 1: {disc1}')
print(f'Función discriminante para x = {x_test} en la clase 2: {disc2} \n')

# Se imprime la clase a la que pertenece el valor de x
if disc1 > disc2:
    print(f"x = {x_test}, está en la clase 1")
else:
    print(f"x = {x_test}, está en la clase 2")
```

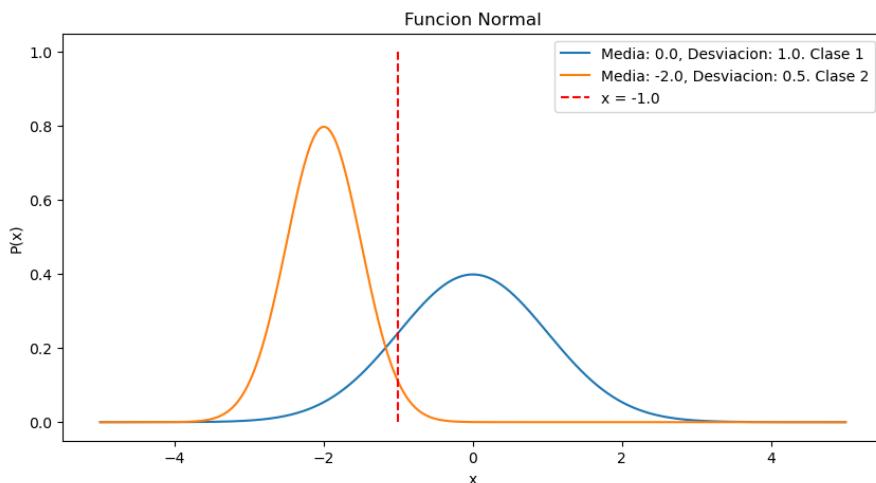


Figure 2: Punto x con respecto a las distribuciones

Al final, se imprime el valor de las discriminantes y se establece una regla de decisión para determinar a que clase pertenece el valor de x, imprimiendo así, la clase a la que pertenece.

```
Funcion discriminante para x = -1 en la clase 1: -2.112085713764618  
Funcion discriminante para x = -1 en la clase 2: -2.9189385332046727
```

```
x = -1, esta en la clase 1
```



CLASIFICADOR BAYESIANO

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

TAREA 3: CLASIFICADOR BAYESIANO

Hacer un programa en python para clasificar un vector característico \mathbf{x} (proporcionado por el usuario), dada la siguiente información:

- Clase 1: $R_1(3, 0), R_2(5, -2), R_3(3, -4), R_4(1, -2)$; $R = \{R_1, R_2, R_3, R_4\}$
 - Clase 2: $N_1(3, 8), N_2(4, 6), N_3(3, 4), N_4(2, 6)$; $N = \{N_1, N_2, N_3, N_4\}$
-
- Graficar el diagrama de dispersión.
 - Calcular los vectores de medias y las matrices de covarianza (graficar las medias en el diagrama de dispersión).
 - Graficar el vector característico (punto a clasificar) en el diagrama de dispersión y mostrar a que clase pertenece. Asumir $P(R) = 0.49$ y $P(N) = 0.51$.

Resultados

- Graficar el diagrama de dispersión.

Para graficar el diagrama de dispersión, primero tenemos que declarar los vectores correspondientes a cada clase para luego graficarlos en el diagrama de dispersión.

```
# Se declaran los puntos
R1 = np.array([3, 0])
R2 = np.array([5, -2])
R3 = np.array([3, -4])
R4 = np.array([1, -2])

N1 = np.array([3, 8])
N2 = np.array([4, 6])
N3 = np.array([3, 4])
N4 = np.array([2, 6])

R = np.array([R1, R2, R3, R4])
N = np.array([N1, N2, N3, N4])

# Se hace el diagrama de dispersion
plt.figure(1, figsize=(10, 5))
plt.scatter(R[:, 0], R[:, 1], color='blue', label='Clase 1')
plt.scatter(N[:, 0], N[:, 1], color='red', label='Clase 2')
plt.xlabel('x')
plt.ylabel('y')
plt.title("Diagrama de dispersion")
```

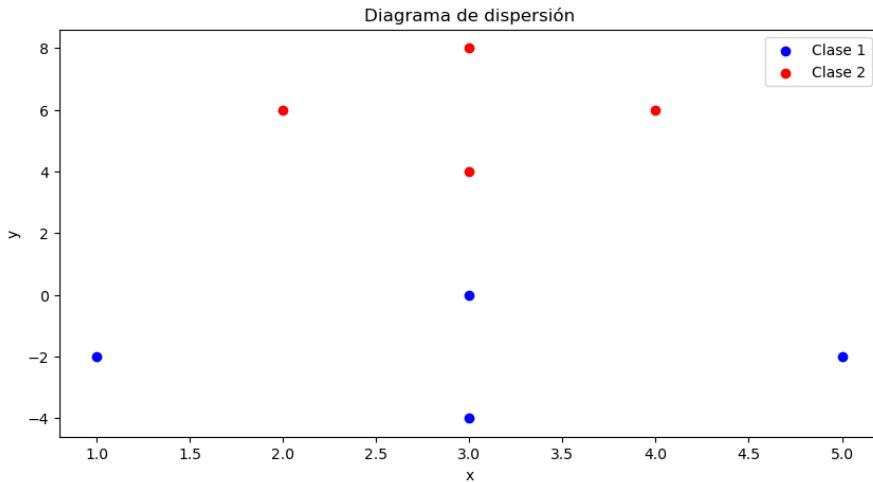


Figure 1: Diagrama de dispersión

- b) Calcular vectores de medias y matrices de covarianza.

Para calcular los vectores de medias y matrices de covarianza, se utilizan las funciones **mean()** y **cov()** de numpy respectivamente. Se grafican las medias en el diagrama de dispersión.

```
# Se calculan las medias de cada clase y se grafican en el diagrama
mean_R = np.mean(R, axis=0)
mean_N = np.mean(N, axis=0)
plt.scatter(mean_R[0], mean_R[1], marker='x', color='green', label='Media Clase 1')
plt.scatter(mean_N[0], mean_N[1], marker='x', color='orange', label='Media Clase 2')

# Se calculan las covarianzas de cada clase
cov_R = np.cov(R.T)
cov_N = np.cov(N.T)
```

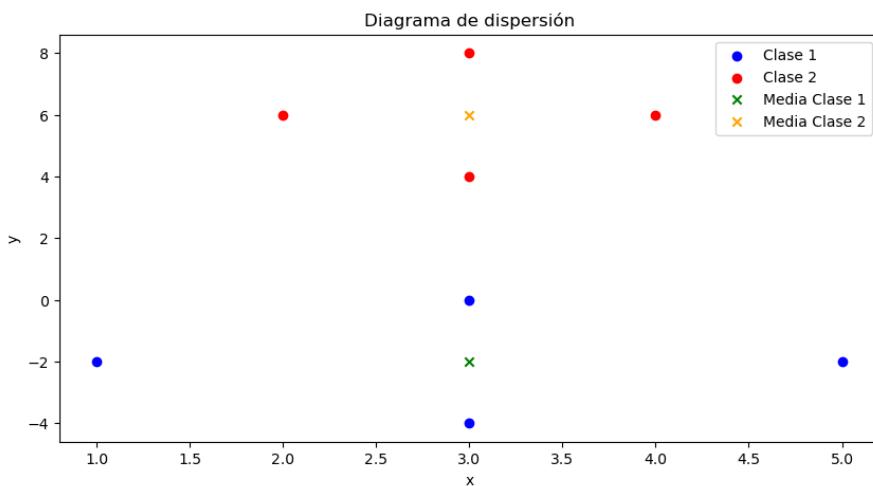


Figure 2: Medias de cada clase

- c) Dar un punto a clasificar y graficarlo en el diagrama de dispersión. Asumir $P(R) = 0.49$ y $P(N) = 0.51$.

Se va a solicitar al usuario que ingrese un vector, el cual se grafica en el diagrama de dispersión. Ademas se calcula la discriminante para este vector, utilizando el caso 3 de la funcion, donde Σ_i es arbitrario.

$$g_i(x) = (-1/2)(x - \mu_i)^t \sum_i^{-1}(x - \mu_i) - (1/2)\ln(|\sum_i|) + \ln(P(w_i))$$

Para esto, se crea un metodo el cual separa la ecuacion en 3 partes, las calcula y luego las suma para obtener la discriminante del vector con respecto a la media y covarianza de cada clase, para finalmente, mostrar los valores de la discriminante, aplicar la regla de decisión y clasificar el vector.

```
def disc(x, mu, cov, prior):
    c = np.log(prior)
    b = -1/2 * np.log(np.linalg.det(cov))
    a = -1/2 * (x-mu).T @ np.linalg.inv(cov) @ (x-mu)
    return a + b + c

# Se solicita el punto a clasificar
x = np.array([float(input("Ingrese x: ")), float(input("Ingrese y: "))])

plt.scatter(x[0], x[1], marker='D', color='black', label='Punto a clasificar')
plt.legend()

# Calculamos las funciones discriminantes
disc_R = disc(x, mean_R, cov_R, 0.49)
disc_N = disc(x, mean_N, cov_N, 0.51)

print("Discriminante Clase 1:", disc_R)
print("Discriminante Clase 2:", disc_N)

if disc_R > disc_N:
    print("El punto pertenece a la clase 1")
else:
    print("El punto pertenece a la clase 2")
```

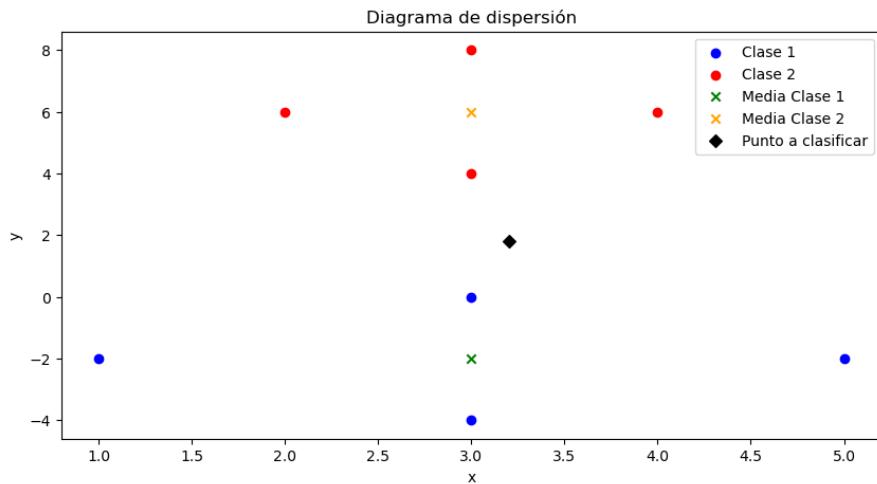


Figure 3: Punto a clasificar

```
Discriminante Clase 1: -4.40917914088919  
Discriminante Clase 2: -4.298526625715547
```

```
El punto pertenece a la clase 2
```



REGIONES DE DECISIÓN

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

TAREA 4: REGIONES DE DECISIÓN

Usando el script de clase (ejemplo de la región de decisión usando Simpy), encuentre las regiones de decisión de la tarea 3.

- Encontrar las ecuaciones para cada región de decisión.
- Evaluar y graficar la región de decisión junto con los datos (diagrama de dispersión).

Resultados

Sabemos de los datos de la tarea 3, que tenemos 2 clases, **R** y **N**.

$$R_1(3, 0), R_2(5, -2), R_3(3, -4), R_4(1, -2) \rightarrow \mathbf{R} = \{R_1, R_2, R_3, R_4\}$$

$$N_1(3, 8), N_2(4, 6), N_3(3, 4), N_4(2, 6) \rightarrow \mathbf{N} = \{N_1, N_2, N_3, N_4\}$$

Utilizando el código desarrollado para resolver la tarea 3, obtenemos diferentes valores: $\mu_R, \Sigma_R, \mu_N, \Sigma_N$. Lo cual nos deja con los siguientes datos:

$$\underbrace{P(x|R)}_{=w_1} \simeq N\left(\begin{pmatrix} 3 \\ 2 \end{pmatrix}, \begin{pmatrix} 2.666 & 0 \\ 0 & 2.666 \end{pmatrix}\right) \quad \underbrace{P(x|N)}_{=w_2} \simeq N\left(\begin{pmatrix} 3 \\ 6 \end{pmatrix}, \begin{pmatrix} .666 & 0 \\ 0 & 2.666 \end{pmatrix}\right) \quad P(R) = 0.49 \quad P(N) = 0.51$$

Seguido, teniendo en cuenta la función discriminante para un punto:

$$g_i(x) = (-1/2)(x - \mu_i)^T \Sigma_i^{-1} (x - \mu_i) - (1/2) \ln(|\Sigma_i|) + \ln(P(w_i))$$

Sabiendo que tenemos dos clases (**R** y **N**) podemos definir dos funciones discriminantes, una para cada clase, g_R y g_N :

$$g_R(x) = \left(-\frac{1}{2}\right) (x - \mu_R)^T \Sigma_R^{-1} (x - \mu_R) - \left(\frac{1}{2}\right) \ln(|\Sigma_R|) + \ln(P(R))$$

$$g_N(x) = \left(-\frac{1}{2}\right) (x - \mu_N)^T \Sigma_N^{-1} (x - \mu_N) - \left(\frac{1}{2}\right) \ln(|\Sigma_N|) + \ln(P(N))$$

Generalizando para un punto $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, podemos sustituir los valores en la función $g_i(x)$:

$$g_R(x) = \left(-\frac{1}{2}\right) ((x_1 - 3)^2 + (x_2 - 2)^2) \cdot \begin{pmatrix} 2.666 & 0 \\ 0 & 2.666 \end{pmatrix}^{-1} - \left(\frac{1}{2}\right) \ln(|\begin{pmatrix} 2.666 & 0 \\ 0 & 2.666 \end{pmatrix}|) + \ln(0.49)$$

$$g_N(x) = \left(-\frac{1}{2}\right) ((x_1 - 3)^2 + (x_2 - 6)^2) \cdot \begin{pmatrix} .666 & 0 \\ 0 & 2.666 \end{pmatrix}^{-1} - \left(\frac{1}{2}\right) \ln(|\begin{pmatrix} .666 & 0 \\ 0 & 2.666 \end{pmatrix}|) + \ln(0.51)$$

Simplificando las funciones discriminantes obtenemos:

$$g_R(x) = -4.13 + 1.12x_1 - 0.18x_1^2 - 0.75x_2 - 0.18x_2^2$$

$$g_N(x) = -14.46 + 4.50x_1 - 0.75x_1^2 + 2.25x_2 - 0.18x_2^2$$

Utilizando *Mathematica* como apoyo, solucionamos las ecuaciones para x_1 y x_2 dando como resultado:

$$x_2 = 0.0033(1033 - 338x_1 + 57x_1^2)$$

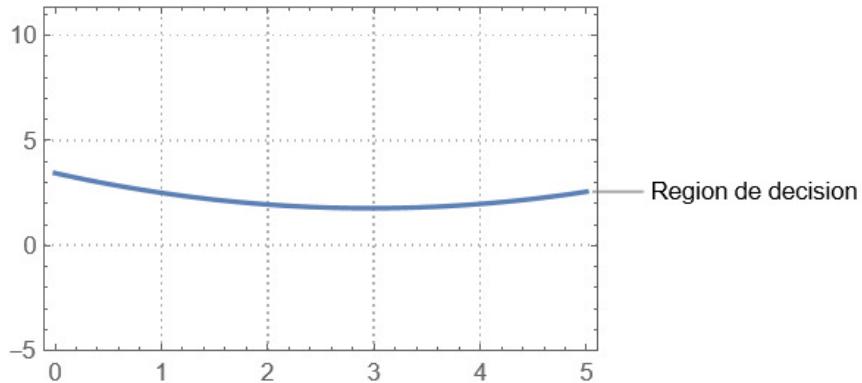


Figure 1: Grafica de la función x_2

Agregando la region de decision al código de la tarea 3, obtenemos el siguiente resultado:

```
#Agrega la region de decision
xr = np.linspace(0, 5, 100)
yr = 0.0033 * (1033-338*xr+57*xr**2)
plt.plot(xr, yr, label="Region de Decision")
plt.legend()
```

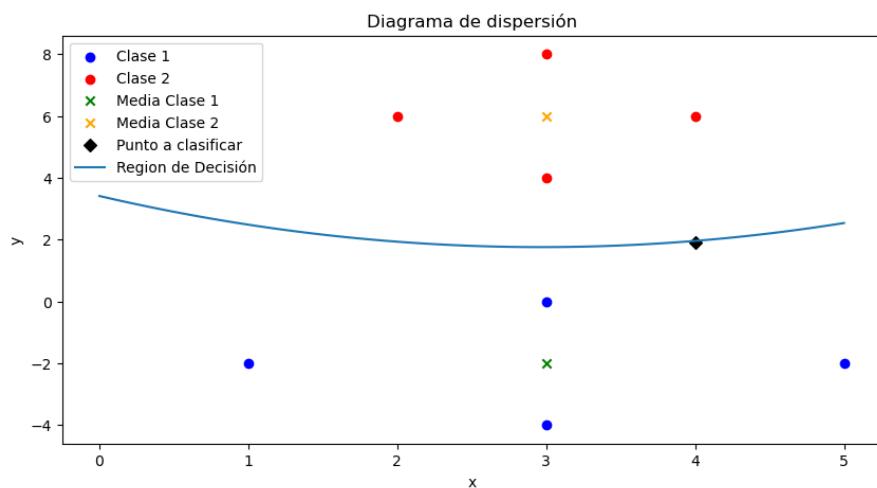


Figure 2: Grafica de la region de decision del programa de la tarea 3. $x = (4, 1.9)$

```
Discriminante Clase 1: -4.733554140889191
Discriminante Clase 2: -4.862901625715546
El punto pertenece a la clase 1
```

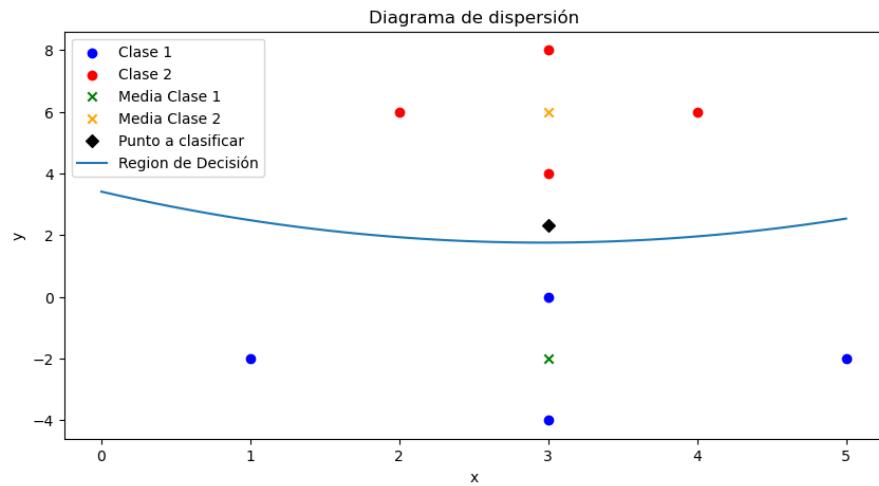


Figure 3: Grafica de la region de decision del programa de la tarea 3. $x = (3,2.2)$

```
Discriminante Clase 1: -5.16105414088919
Discriminante Clase 2: -3.527901625715547
El punto pertenece a la clase 2
```



CURVAS ROC

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

TAREA 5: CURVAS ROC

Hacer un programa en python para graficar la curva ROC a partir de los datos proporcionados en el archivo excel.

Resultados

Tenemos un archivo de excel con los datos obtenidos, nuestro objetivo, es calcular los True Positives (TP), True Negatives (TN), False Positives (FP) y False Negatives (FN) de esos datos, para luego graficar la curva ROC. Para ello, se desarrolla el siguiente algoritmo.

Empezamos cargando los datos del archivo excel, limpiado y acomodando la estructura.

```
# Curva ROC

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sys

#Cargar el archivo de excel
df = pd.read_excel('.../Reconocimiento de patrones/Bases de datos/
    ↪ RESULTADOS_COOKING_SOUNDS_MFCC_3dB_DC.xlsx')

# Quitar columnas y renombrar la primera a Mezcla. Quitar filas con NaN
df.drop(['Unnamed: 19', 'Unnamed: 20'], axis=1, inplace=True)
df.rename(columns={'Unnamed: 0': 'Mezcla'}, inplace=True)
df.dropna(inplace=True)
```

Inicializamos las variables TP, FP, FN y TN en 0 para ir almacenando los datos obtenidos, tambien se inician otras 3 variables, *seccion1* y *seccion2* que representan el inicio y el final de la sección de datos con respecto a una etiqueta, por ejemplo, *seccion1* = 0 y *seccion2* = 104 representan los datos de M_1_2_3 a M_1_15_16 que representan los datos de la etiqueta 1. Por ultimo, *truelabel* indica la etiqueta que estamos esperando en cierto momento dentro del ciclo, esta va a ir incrementando de 1 en 1.

```
# Se inicializa TP, FP, FN, TN en 0
TP = 0
FP = 0
FN = 0
TN = 0

# Se inicializa una seccion1 y seccion2 denotando el inicio y el final del audio a
    ↪ analizar (M1_x_x, M2_x_x, etc)
seccion1 = 0
seccion2 = 104
# Se inicializa truelabel denotando la clase a la que pertenece la seccion
truelabel = 1
```

Para calcular los TP, FP, FN y TN, se hace uso de la siguiente logica:

```

Si Distancia <= Umbral
    Si clase x = clase x
        TP = TP + 1
    else
        FP = FP + 1
Si Distancia > Umbral
    Si clase x = clase x
        FN = FN + 1
    else
        TN = TN + 1

```

Ya con los datos inicializados, se hace un ciclo *while* el cual consiste en varias operaciones. Primero se declara que el ciclo va a durar mientras que *seccion2* sea menor a 1695, o sea, hasta que recorra todos los renglones de la tabla. Seguido de esto, se declara un ciclo *for* el cual va a recorrer todos los valores del umbral (0-180).

Un nivel más abajo, se declara un *if* el cual va a recorrer los valores de *seccion1* a *seccion2*, o sea, los renglones correspondientes a cada clase. Dentro de este ciclo se propone la siguiente logica: si *df.iloc[i,17]*, o sea, si el valor del renglon *i* en la columna 17, el cual denota el valor maximo de los 16 audios para la mezcla, es menor o igual al umbral, entonces va a comparar si *df.iloc[i,18]*, o sea, la clase clasificada correspondiente a la mezcla, es igual a *truelabel*, la clase esperada.

Si esto se cumple, entonces se va a sumar 1 a TP. Si no se cumple, entonces se va a sumar 1 a FP, de la misma manera, si *df.iloc[i,17]* es mayor al umbral, entonces va a aplicar la misma logica para clasificarlo en FN o TN.

Al momento de finalizar los ciclos *for*, se incrementan los valores de *seccion1* y *seccion2* para que se recorran los audios de la siguiente clase, de misma manera, se incrementa *truelabel* para representar la nueva clase esperada.

```

while seccion2 < 1695:
    print(f'Calculando seccion {df.iloc[seccion1,0]} a {df.iloc[seccion2,0]} para la
          ↪ clase {truelabel}')
    for k in range(0,181): #Threshold
        for i in range(seccion1,seccion2): #Row
            if df.iloc[i, 17] <= k:
                if df.iloc[i, 18] == truelabel:
                    TP += 1
                else:
                    FP += 1
            else:
                if df.iloc[i, 18] == truelabel:
                    FN += 1
                else:
                    TN += 1

    seccion1 = seccion2+1
    seccion2 += 105
    truelabel += 1

```

Una vez finalizado el ciclo *while*, vamos a tener un número de valores para TP, FP, FN y TN, con estos valores vamos a calcular la sensibilidad y la especificidad con las siguientes formulas:

$$\text{sensitivity} = \frac{TP}{TP + FN} \quad (1)$$

$$\text{specificity} = \frac{TN}{TN + FP} \quad (2)$$

Estos valores son guardados en unas listas que van de 0 a 1.

```
sensitivity = TP / (TP + FN)
specificity = TN / (TN + FP)
print(sensitivity, specificity)

SENlist = [0,sensitivity,1]
SPClist = [0,specificity,1]
```

Finalmente, se grafican las curvas ROC con las listas obtenidas.

```
plt.plot(SENlist, SPClist, color='blue', lw=2, label='Curva ROC')
plt.xlim(0,1)
plt.ylim(0,1)
plt.xlabel('1-Specificidad')
plt.ylabel('Sensibilidad')
plt.legend()
```

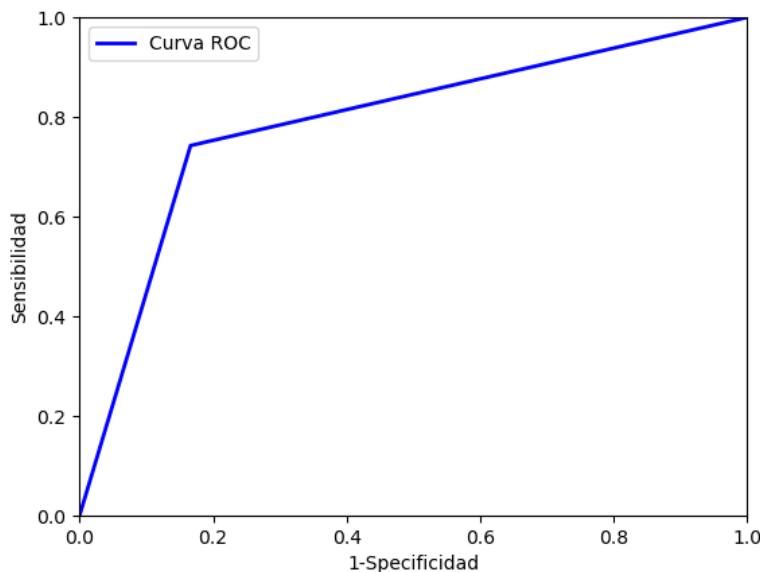


Figure 1: Curva ROC para (0.16, 0.74)



TAREA 3: DYNAMIC TIME WARPING

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

Programar el método de Dynamic Time Warping para medir la distancia entre Cepstrums. Los cepstrum fueron obtenidos a partir de 3 audios de voz, donde los archivos denominados "cepstrum_1.csv" y "cepstrum_2.csv" corresponde al mismo audio (una frase dicha por un hombre y una mujer respectivamente). El archivo denominado "cepstrum_3.csv" contiene el cepstrum de un audio de voz de una frase totalmente diferente.

- a) Obtener las 3 matrices de distancias al comparar el cepstrum_1 vs cepstrum_2, cepstrum_1 vs cepstrum_3 y cepstrum_2 vs cepstrum_3. Graficar dichas matrices en forma de imagen. Usar la distancia coseno para construir la matriz de distancias.
- b) Generar las trayectorias de doblado en cada matriz. Graficar la trayectoria de doblado sobre las imágenes.
- c) Comprobar que la distancia mínima se obtiene al comparar cepstrum_1 vs cepstrum_2. Despliega la distancia total para cada caso (cepstrum_1 vs cepstrum_2, cepstrum_1 vs cepstrum_3 y cepstrum_2 vs cepstrum_3).
- d) Anotar sus conclusiones.

Resultados

- a) Obtener las 3 matrices de distancias al comparar el cepstrum_1 vs cepstrum_2, cepstrum_1 vs cepstrum_3 y cepstrum_2 vs cepstrum_3. Graficar dichas matrices en forma de imagen. Usar la distancia coseno para construir la matriz de distancias.

Para la obtención de las matrices de distancias, se hace uso de la distancia coseno, la cual se define de la siguiente manera:

$$\text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \cdot \sqrt{\sum_{i=1}^n B_i^2}}, \quad (1)$$

```
#Dynamic Time Warping

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def cosine_distance(X, Y):

    X = np.asarray(X).flatten()
    Y = np.asarray(Y).flatten()

    Nx = np.sqrt(np.sum(X**2))
    Ny = np.sqrt(np.sum(Y**2))

    P = np.sum(X * Y)

    d_x_y = P / (Nx * Ny)

    return d_x_y
```

Una vez definida la distancia coseno, se puede obtener la matriz de distancias resultantes de la comparación de los cepstrums.

Para desarrollar el metodo de DTW se hace uso base del pseudocodigo de Wikipedia [1] y del repositorio de Github <https://github.com/talcs/simpledtw> [2], asi como el blog de Romain Tavenard [3] para implementar el algoritmo de DTW con la distancia coseno para la comparación de los cepstrums.

```

def dtw(X, Y, distance_func=cosine_distance):
    matrix = np.zeros((len(X) + 1, len(Y) + 1))
    matrix[0, :] = np.inf
    matrix[:, 0] = np.inf
    matrix[0, 0] = 0

    for i, vec1 in enumerate(X):
        for j, vec2 in enumerate(Y):
            cost = distance_func(vec1, vec2)
            matrix[i + 1, j + 1] = cost + min(
                matrix[i, j + 1],
                matrix[i + 1, j],
                matrix[i, j]
            )

    matrix = matrix[1:, 1:]

    i, j = matrix.shape[0] - 1, matrix.shape[1] - 1
    matches = []
    mappings_series_1 = [list() for _ in range(matrix.shape[0])]
    mappings_series_2 = [list() for _ in range(matrix.shape[1])]

    while i > 0 or j > 0:
        matches.append((i, j))
        mappings_series_1[i].append(j)
        mappings_series_2[j].append(i)

        option_diag = matrix[i - 1, j - 1] if i > 0 and j > 0 else np.inf
        option_up = matrix[i - 1, j] if i > 0 else np.inf
        option_left = matrix[i, j - 1] if j > 0 else np.inf

        move = np.argmin([option_diag, option_up, option_left])
        if move == 0:
            i -= 1
            j -= 1
        elif move == 1:
            i -= 1
        else:
            j -= 1

    matches.append((0, 0))
    mappings_series_1[0].append(0)
    mappings_series_2[0].append(0)
    matches.reverse()

    for mp in mappings_series_1:
        mp.reverse()

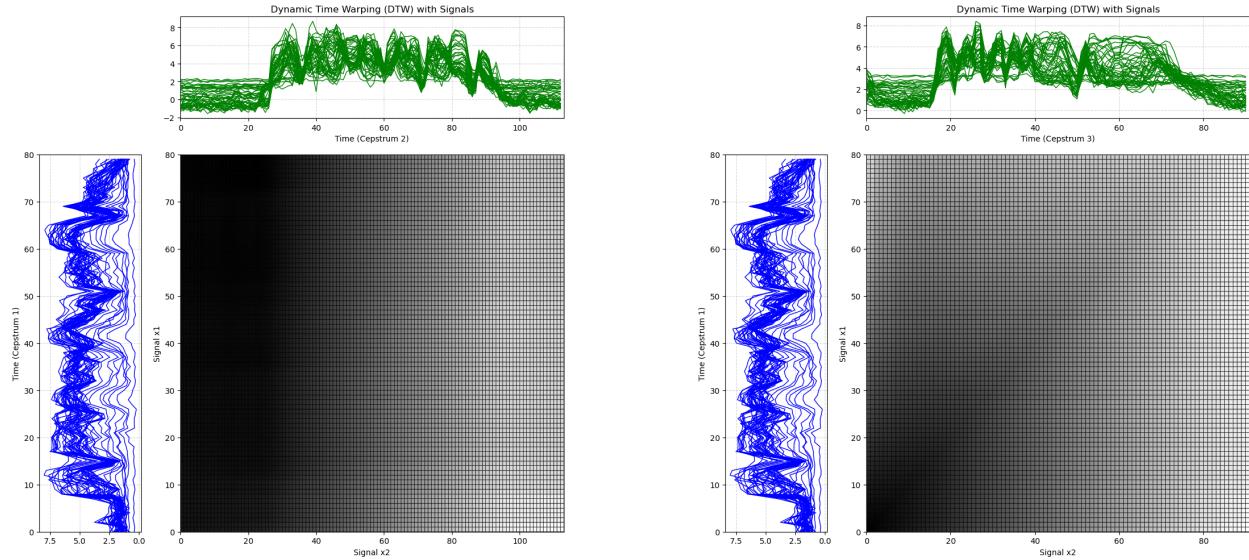
```

```

for mp in mappings_series_2:
    mp.reverse()

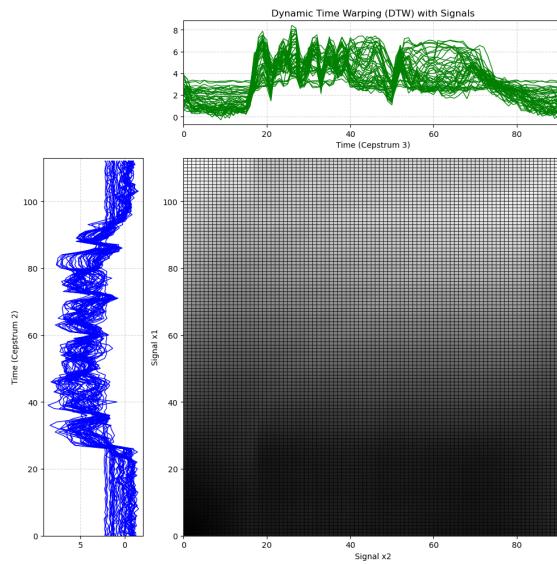
return matches, matrix[-1, -1], mappings_series_1, mappings_series_2, matrix

```



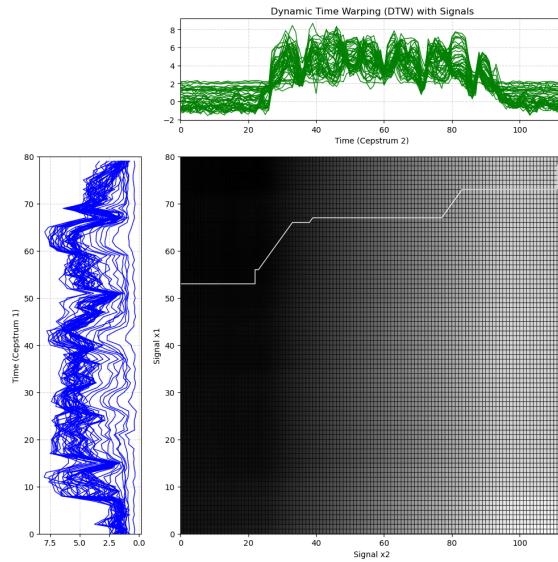
(a) Matriz de distancia entre cepstrum_1 y cepstrum_2

(b) Matriz de distancia entre cepstrum_1 y cepstrum_3

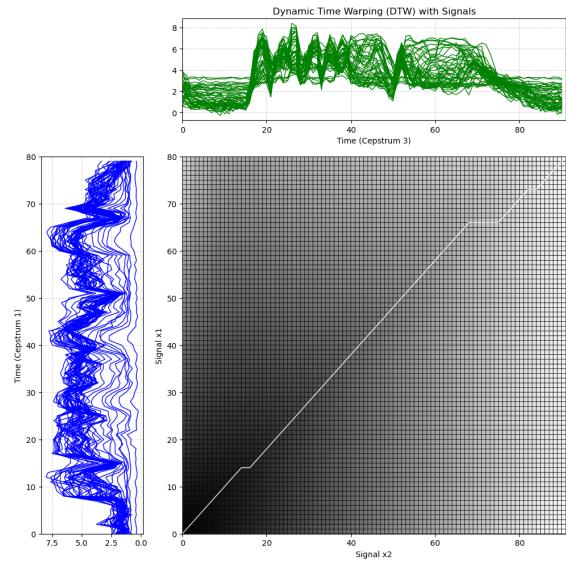


(c) Matriz de distancia entre cepstrum_2 y cepstrum_3

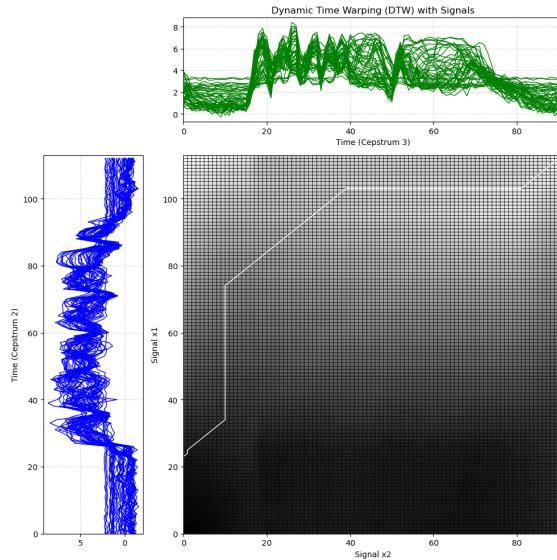
- b) Generar las trayectorias de doblado en cada matriz. Graficar la trayectoria de doblado sobre las imágenes.



(a) Matriz de distancia con trayectoria de doblado entre cepstrum_1 y cepstrum_2



(b) Matriz de distancia con trayectoria de doblado entre cepstrum_1 y cepstrum_3



(c) Matriz de distancia con trayectoria de doblado entre cepstrum_2 y cepstrum_3

- c) Comprobar que la distancia mínima se obtiene al comparar cepstrum_1 vs cepstrum_2. Despliega la distancia total para cada caso (cepstrum_1 vs cepstrum_2, cepstrum_1 vs cepstrum_3 y cepstrum_2 vs cepstrum_3).
- Distancia minima entre cepstrum_1 y cepstrum_2: 59.89831444619493
 - Distancia minima entre cepstrum_1 y cepstrum_3: 80.26619317948834
 - Distancia minima entre cepstrum_2 y cepstrum_3: 76.67493532097427

- d) Anotar sus conclusiones.

El dynamic time warping es un algoritmo utilizado para comparar la similitud entre dos series de tiempo que pueden variar en velocidad. Lo que el DTW busca hacer, es encontrar el alineamiento óptimo que reduzca la distancia entre las dos series de tiempo.

Comparando las distancias de los cepstrum 1 y 2 (59.89) contra la distancia de los cepstrum 1 y 3 (80.27) y la distancia de los cepstrum 2 y 3 (76.67), podemos concluir que la distancia mínima se obtiene al comparar cepstrum_1 vs cepstrum_2. Esto quiere decir que la serie de cepstrum 1 es más similar a la serie de cepstrum 2 que a la serie de cepstrum 3. Confirmando así, que la frase dicha por los locutores de la serie de cepstrum 1 y cepstrum 2, es la misma.

Referencias

- 1 Dynamic time warping. (2024). In Wikipedia. https://en.wikipedia.org/w/index.php?title=Dynamic_time_warping&oldid=1262411913
- 2 talcs. (2025). Tales/simpledtw [Python]. <https://github.com/talcs/simpledtw> (Original work published 2018)
- 3 An introduction to dynamic time warping. (n.d.). Retrieved April 10, 2025, from <https://rtavenar.github.io/blog/dtw.html>



TAREA 1: HISTOGRAMA

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

A partir de un conjunto de datos, generar un histograma usando un ancho de bin de:

- a) Usando el criterio de Freedman.
- b) Usando la raiz cuadrada de N.
- c) Anotar sus conclusiones.

Resultados

Para este trabajo, se utilizará una base de datos, que consiste en diferentes masas de cuerpos astronomicos.

$$x = \{10, 10, 20, 25, 40, \dots, 95\}$$

Lo primero que tenemos que realizar es cargar la base de datos y convertirla en un array de numpy.

```
# Histogramas

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import iqr
import seaborn as sns

# Datos
df = pd.read_csv(".../Reconocimiento de patrones/Bases de datos/mass1.csv")

x = np.array(df["mass1"])
```

- a) Usando el criterio de Freedman-Diaconis

El criterio de Freedman-Diaconis dice que el ancho de bin debe ser:

$$k = \frac{2IQR(x)}{\sqrt[3]{M}} \quad (1)$$

Donde $IQR(x)$ es el rango intercuartilico, y M es el tamaño de la muestra.

```
# Freedman - Diaconis

IQR = iqr(x)
M = len(df) ** (1 / 3)
k = round((2 * IQR) / M)

sns.histplot(x, bins=13)
plt.title(f"Histograma con Freedman-Diaconis de ancho de bin {k}")
plt.show()
```

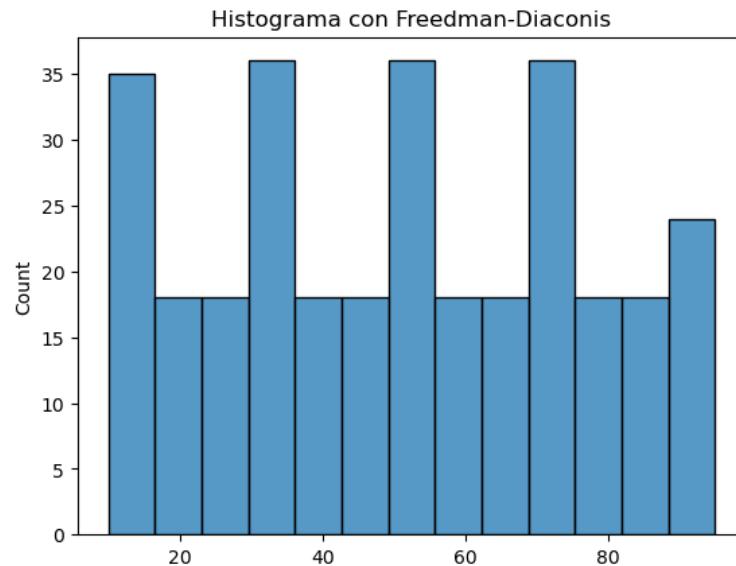


Figure 1: Histograma con Freedman-Diaconis

a) Usando la raiz cuadrada de N

Otro criterio para calcular el ancho de bin es la raiz cuadrada de N.

$$k = \sqrt{N} \quad (2)$$

```
#sqrt M
k = round(np.sqrt(len(df)))

sns.histplot(x, bins=k).set_title(f"Histograma con sqrt(M) de ancho de bin {k}")
plt.show()
```

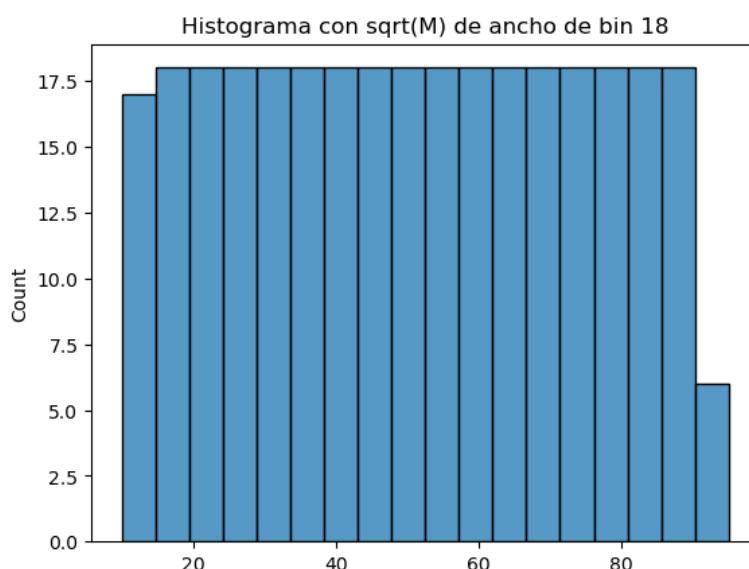


Figure 2: Histograma con sqrt(M)

a) Conclusiones

Aunque el criterio de la raíz cuadrada de N (2) para calcular los anchos de bins es simple y directo, puede no adaptarse bien a todos los conjuntos de datos. En cambio, el criterio de Freedman-Diaconis (1) utiliza la distribución de los datos (IQR) para determinar un ancho de bin más adecuado.



TAREA 2: VENTANAS DE PARZEN

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

Programar el método de estimación de PDF por Ventanas de Parzen usando un conjunto de datos 2-dimensionales. Usar el código de la función normal multivariada programada en clase.

Resultados

Como habíamos visto en clase, la estimación de la densidad en x por ventanas de Parzen está dada por:

$$\hat{p}(x) = \frac{1}{N} \sum_{n=1}^N \phi\left(\frac{x - x_n}{h}\right) \frac{1}{h^d} \quad (1)$$

Y por el problema de las discontinuidades, podemos utilizar un kernel gaussiano para la estimación de la densidad.

$$\hat{p}(x) = \frac{1}{N} \sum_{n=1}^N G(x, x_n, \Sigma) \quad (2)$$

De esta manera, para calcular la densidad de una clase, de datos unidimensionales, la PDF se puede calcular de la siguiente manera:

$$G(x, x_n, h) = \frac{1}{\sqrt{2\pi}\sigma} e^{-1/2} \frac{(x - x_n)^2}{\sigma^2} \quad (3)$$

Pero para calcular la PDF de datos multivariados, debemos cambiar el kernel a una multivariada.

$$G(x, x_n, h) = \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} e^{-1/2(x-x_n)^T \Sigma^{-1} (x-x_n)} \quad (4)$$

Teniendo esto en mente, y apoyandonos del código de la función normal multivariada desarrollada en clase, podemos realizar la estimación de la PDF por ventanas de Parzen.

Se inicializa la clase

```
# Parzen

import numpy as np
import matplotlib.pyplot as plt
import sys

class gaussiana:
    def __init__(self, datos, sigma = 0.01):
        self.datos = datos
        self.sigma = sigma
        self.d = len(datos)
        self.N = len(datos[0])
```

Utilizamos el metodo desarrollado en clase para calcular la PDF de datos multivariados

```
def probabilidad_normal(self, X, M, S):
    #Encuentra la dimension del vector caracteristico
    d = len(X)
    #Evalua la parte exponencial de la funcion normal multivariada
    e = (-1/2)*((X-M)@np.linalg.pinv(S)@(X-M).T)
    #Obtencion de los eigenvalores
    V, U = np.linalg.eig(S)
    #Calculo del seudodeterminante
    det = 1
    for i in range(0,d):
        if V[i] >= sys.float_info.epsilon:
            det = det*V[i]
    #Calculo de la constante de la funcion normal
    CTE = 1/((2*np.pi)**(d/2)*det**(1/2))
    p = CTE*np.exp(e)
    return p
```

Se programa un metodo para realizar la estimación de la PDF por ventanas de Parzen

```
def probabilidad_parzen(self,X):
    px = 0
    for i in range(self.N):
        diff = X - self.datos[:, i]
        kernel = np.exp(-0.5 * np.dot(diff, diff) / (self.sigma**2))
        px += kernel

    px = px / (self.N * (2 * np.pi) ** (self.d / 2) * self.sigma ** self.d)
    return px
```

Se integra y expande el metodo para calcular las distribuciones de los datos

```
def distribucion(self, plot = None, metodo = 'normal'):
    #Numero de dimensiones
    d = len(np.shape(self.datos))
    #Calculo de las medias
    M = np.zeros(d)
    for i in range(0,d):
        M[i] = np.mean(self.datos[i, :])

    if metodo == 'normal':

        Sigma = self.sigma**2 * np.eye(self.d)

        X = np.array(self.datos[:, 0])

        x_min, x_max = np.min(self.datos[0, :]), np.max(self.datos[0, :])
        y_min, y_max = np.min(self.datos[1, :]), np.max(self.datos[1, :])

        XX, YY = np.meshgrid(np.linspace(x_min, x_max, 50),
                             np.linspace(y_min, y_max, 50))

        ZZ = np.zeros_like(XX)
        for i in range(XX.shape[0]):
```

```

        for j in range(XX.shape[1]):
            point = np.array([XX[i,j], YY[i,j]])
            ZZ[i,j] = self.probabilidad_normal(point, M, Sigma)

    title = 'Grafica con funcion normal'

elif metodo == 'parzen':

    x_min, x_max = np.min(self.datos[0, :]), np.max(self.datos[0, :])
    y_min, y_max = np.min(self.datos[1, :]), np.max(self.datos[1, :])

    XX, YY = np.meshgrid(np.linspace(x_min, x_max, 50),
                         np.linspace(y_min, y_max, 50))

    ZZ = np.zeros_like(XX)
    for i in range(XX.shape[0]):
        for j in range(XX.shape[1]):
            point = np.array([XX[i,j], YY[i,j]])
            ZZ[i,j] = self.probabilidad_parzen(point)

    title = 'Grafica con ventana de Parzen'

if plot == True:
    #Grafica
    plt.figure(2)
    ax = plt.axes(projection='3d')
    ax.contour3D(XX,YY,ZZ, 100)
    ax.set_xlabel('Caracteristica eje X')
    ax.set_ylabel('Caracteristica eje Y')
    ax.set_zlabel('Probabilidad')
    ax.set_title(title)
    plt.show()

else:
    return ZZ

```

Para evaluar la PDF de datos multivariados, se proponen los siguientes conjuntos:

$$x_1 = \{(-.6705, -.6532, -.7735, -.8528, -.6322, -.6139, -.6731, -.9493), \\ (-.8029, -.7535, -.7618, -.7335, -.7451, -.6399, -.7949, -.8542)\}$$

$$x_2 = \{(-.6628, -.3807, -.4556, -.3183, -.1202, -.3452, -.0667, -.3145), \\ (-.1635, -.3594, -.4593, -.3124, -.6132, -.6313, -.3036, -.5229)\}$$

A los cuales se les calcula la PDF con la ventana de Parzen y la función normal.

```

x1 = np.array([[-.6705, -.6532, -.7735, -.8528, -.6322, -.6139,
                -.6731, -.9493], [-.8029, -.7535, -.7618, -.7335,
                -.7451, -.6399, -.7949, -.8542]])
x2 = np.array([[-.6628, -.3807, -.4556, -.3183, -.1202, -.3452,
                -.0667, -.3145], [-.1635, -.3594, -.4593, -.3124,
                -.6132, -.6313, -.3036, -.5229]])
gaussiana1 = gaussiana(x1, sigma=0.05)

```

```

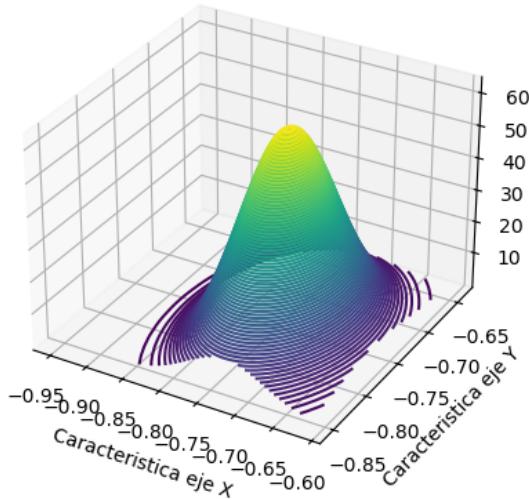
gaussiana2 = gaussiana(x2, sigma=0.05)

gaussiana1.distribucion(plot=True, metodo='normal')
gaussiana1.distribucion(plot=True, metodo='parzen')
gaussiana2.distribucion(plot=True, metodo='normal')
gaussiana2.distribucion(plot=True, metodo='parzen')

```

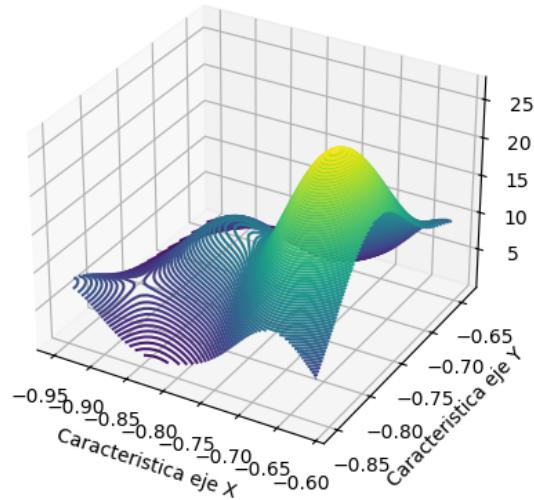
Obteniendo las siguientes graficas:

Grafica con función normal



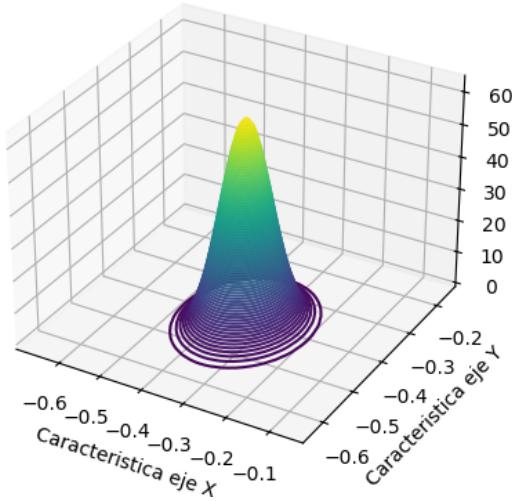
(a) Gráfica con función normal para x1

Grafica con ventana de Parzen



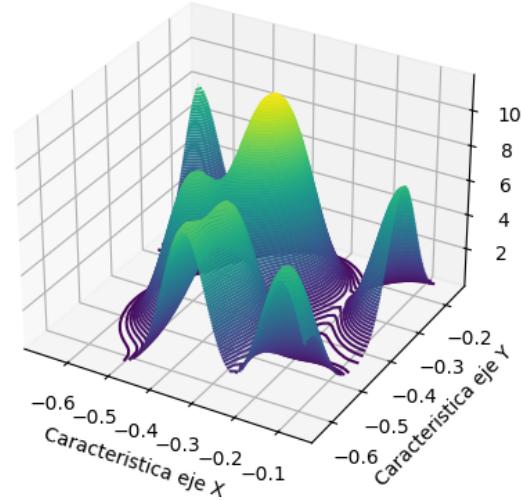
(b) Gráfica con ventana de Parzen para x1

Grafica con función normal



(c) Gráfica con función normal para x2

Grafica con ventana de Parzen



(d) Gráfica con ventana de Parzen para x2

Figure 1: Comparación entre métodos de estimación de densidad



TAREA 4: MAPEO DE TRIANGULOS

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

Calcular la distancia entre Clusters usando el método de Mapeo de Triángulos sobre el plano Complejo dados los conjuntos de datos proporcionados en los archivos de Excel.

- Tomar de cada archivo de Excel dos espectros de manera aleatoria. Graficar los espectros de cada archivo.
- Hacer el mapeo de los triángulos a partir de los datos de cada espectro. Usar la FFT para generar los vértices de cada triángulo. (Tomar ternas para generar el triángulo y usar la mitad de datos).
- Calcular la distancia entre Clusters usando la distancia Euclídea y la siguiente ecuación:

$$D_{total} = \frac{\sum d(x_i, y_j)}{N_x * N_y} \quad (1)$$

- Anotar sus conclusiones.

Resultados

El mapeo de triángulos sobre un círculo unitario es una técnica que puede ser utilizada para el reconocimiento de patrones. Esta técnica consiste en tomar una terna de puntos de una serie de datos y con ello construir un triángulo, el cual será mapeado a un punto del plano complejo.

Para mapear los triángulos a un punto del plano complejo se utiliza la siguiente ecuación:

$$\lambda = e^{2\pi i/3} \quad (2)$$

$$\theta(z_1, z_2, z_3) = \frac{z_1 + \lambda z_2 + \lambda^2 z_3}{z_1 + \lambda^2 z_2 + \lambda z_3} \quad (3)$$

1. Tomar de cada archivo de Excel dos espectros de manera aleatoria. Graficar los espectros de cada archivo.

Cargamos los archivos de Excel, los asignamos a un dataframe, se normaliza el dataframe y se seleccionan 2 señales aleatorias.

```
file_path1 = ".../90_10.xlsx"
file_path2 = ".../70_30.xlsx"
df1 = pd.read_excel(file_path1, header=None)
df2 = pd.read_excel(file_path2, header=None)
scale = StandardScaler()
df1 = pd.DataFrame(scale.fit_transform(df1))
df2 = pd.DataFrame(scale.fit_transform(df2))

random_cols_df1 = np.random.choice(df1.columns, size=2, replace=False)
signal1, signal2 = df1[random_cols_df1[0]], df1[random_cols_df1[1]]
random_cols_df2 = np.random.choice(df2.columns, size=2, replace=False)
signal3, signal4 = df2[random_cols_df2[0]], df2[random_cols_df2[1]]
```

Para graficar los espectros de cada archivo, se utilizan las siguientes funciones:

```
def FFT(signal):
    Xf = np.fft.fft(signal)
    lenxf = len(Xf)
    N = int(lenxf / 2)
    half_fft = Xf[:N]
    return half_fft

def FFT_plot(signal1, signal2):
    signal1 = signal1
    signal2 = signal2

    Xf1 = FFT(signal1)
    Xf2 = FFT(signal2)

    plt.figure(figsize=(10, 5))
    plt.subplot(2, 2, 1)
    plt.plot(np.abs(Xf1))
    plt.title(f"FFT for Signal 1: {signal1.name}")
    plt.xlabel("Frecuencia")

    plt.subplot(2, 2, 2)
    plt.plot(signal1)
    plt.title("Signal 1")
    plt.xlabel("Muestras")

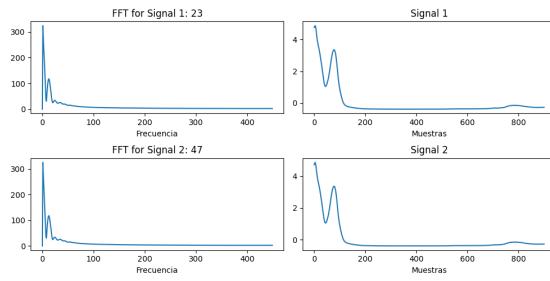
    plt.subplot(2, 2, 3)
    plt.plot(np.abs(Xf2))
    plt.title(f"FFT for Signal 2: {signal2.name}")
    plt.xlabel("Frecuencia")

    plt.subplot(2, 2, 4)
    plt.plot(signal2)
    plt.title("Signal 2")
    plt.xlabel("Muestras")

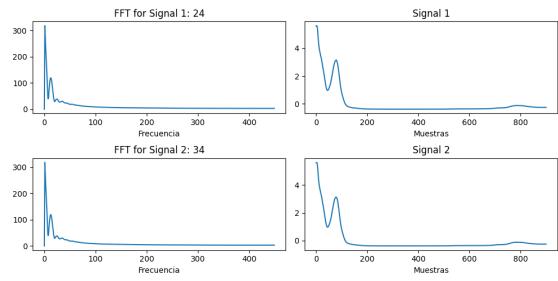
    plt.tight_layout()
    plt.show()
```

La función *FFT()* calcula la FFT de una señal y devuelve la mitad del espectro. La función *FFT_plot()* grafica la FFT de dos espectros y sus señales originales.

```
FFT_plot(signal1, signal2)
FFT_plot(signal3, signal4)
```



(a) Transformada FFT para dos señales del archivo 90% - 10% y su señal original



(b) Transformada FFT para dos señales del archivo 70% - 30% y su señal original

Figure 1: Graficas de dos señales y su transformada FFT

2. Hacer el mapeo de los triángulos a partir de los datos de cada espectro. Usar la FFT para generar los vértices de cada triángulo. (Tomar ternas para generar el triángulo y usar la mitad de datos).

Para generar el mapeo de los triangulos, se utilizan las siguientes funciones:

```
def ComplexMapping(data_list, marker_style="*", color="g", label="Data"):
    N = 3
    l = np.exp(2 * np.pi * 1.0j / N)
    points_inside = 0
    points_on_circumference = 0
    truncated_length = len(data_list) - (len(data_list) % 3)
    truncated_data = data_list[:truncated_length]
    all_points = []
    inside_points = []

    # print(truncated_data)
    for i in range(0, len(truncated_data), 3):
        a = truncated_data[i]
        b = truncated_data[i + 1]
        c = truncated_data[i + 2]

        z1 = a
        z2 = b
        z3 = c
        tr = np.array([z1, z2, z3], dtype=np.complex128)

        suma_num = 0
        suma_den = 0
        for n in range(N):
            suma_num += l**n * tr[n]
            suma_den += l**-n * tr[n]

        if np.abs(suma_den) > 1e-12:
            point = suma_num / suma_den
            magnitude = np.abs(point)

            epsilon = 1e-10
            if magnitude < (1.0 - epsilon):
                plt.plot(
                    point.real, point.imag, marker_style, color=color, markersize=4
                )
                all_points.append(point)
                if magnitude < 1.0 - epsilon:
                    inside_points.append(point)
            else:
                points_on_circumference += 1
        else:
            points_on_circumference += 1
    return all_points, inside_points
```

```

        )
        points_inside += 1
        inside_points.append(point)
    else:
        points_on_circumference += 1
        all_points.append(point)

return points_inside, points_on_circumference, inside_points

```

La función *ComplexMapping()* recibe una lista de datos, la cual consta de la mitad de los datos de la FFT. Para la generación de los triangulos, se toma una terna de datos, para evitar un error por falta de datos, se trunca la lista en caso que no sea divisible por 3.

Los puntos que no esten en la circunferencia se consideran fuera del plano complejo.

Para graficar los puntos dentro del plano complejo, se utiliza la siguiente funcion:

```

def ComplexPlot(signal1, signal2, signal3, signal4):
    half_fft1 = FFT(signal1)
    half_fft2 = FFT(signal2)
    half_fft3 = FFT(signal3)
    half_fft4 = FFT(signal4)

    data_lists = [half_fft1, half_fft2, half_fft3, half_fft4]
    dataset_names = ["Signal 1-df1", "Signal 2-df1", "Signal 3-df2", "Signal 4-df2"]
    colors = ["g", "r", "b", "m"]
    markers = ["*", "o", "+", "x"]
    all_points_global = []
    all_points_by_signal = [[] for _ in range(len(data_lists))]

    plt.figure(figsize=(12, 10))
    x = np.linspace(-1, 1, 1000)
    y1 = np.sqrt(1 - x**2)
    y2 = -np.sqrt(1 - x**2)
    plt.plot(x, y1, "k-", linewidth=1)
    plt.plot(x, y2, "k-", linewidth=1)
    plt.grid(True)
    plt.axis("equal")
    plt.title("Triangle Mapping - Multiple Datasets")

    total_stats = []
    for i, data_list in enumerate(data_lists):
        points_inside, points_on_circumference, points = ComplexMapping(
            data_list, marker_style=markers[i], color=colors[i], label=dataset_names[i]
        )
        total_stats.append((points_inside, points_on_circumference))
        all_points_by_signal[i] = points
        all_points_global.extend(points)

    plt.plot([], [], markers[i], color=colors[i], label=dataset_names[i])

    global_distances, global_avg = calculate_distances(all_points_global)
    print(f"Global avg distance: {global_avg}")

```

```

print(f"Global distances: {global_distances}")

signal_distances = calculate_signal_distances(all_points_by_signal)
print("\nDistancias entre señales:")
for key, value in signal_distances.items():
    print(f"{key}: {value:.6f}")

plt.legend(loc="upper right")
plt.xlabel("Real Part")
plt.ylabel("Imaginary Part")
plt.tight_layout()
plt.show()

```

Esta función calcula las FFT de las señales y las agrega a una lista. Se generan dos listas vacías para almacenar todos los puntos y los puntos por cada señal. Se grafica el círculo unitario. Se genera un loop para iterar sobre la lista de las FFTs. Cada iteración llama a la función *ComplexMapping()* para graficar los puntos dentro del círculo unitario. Finalmente, se agrega el cálculo de las distancias.

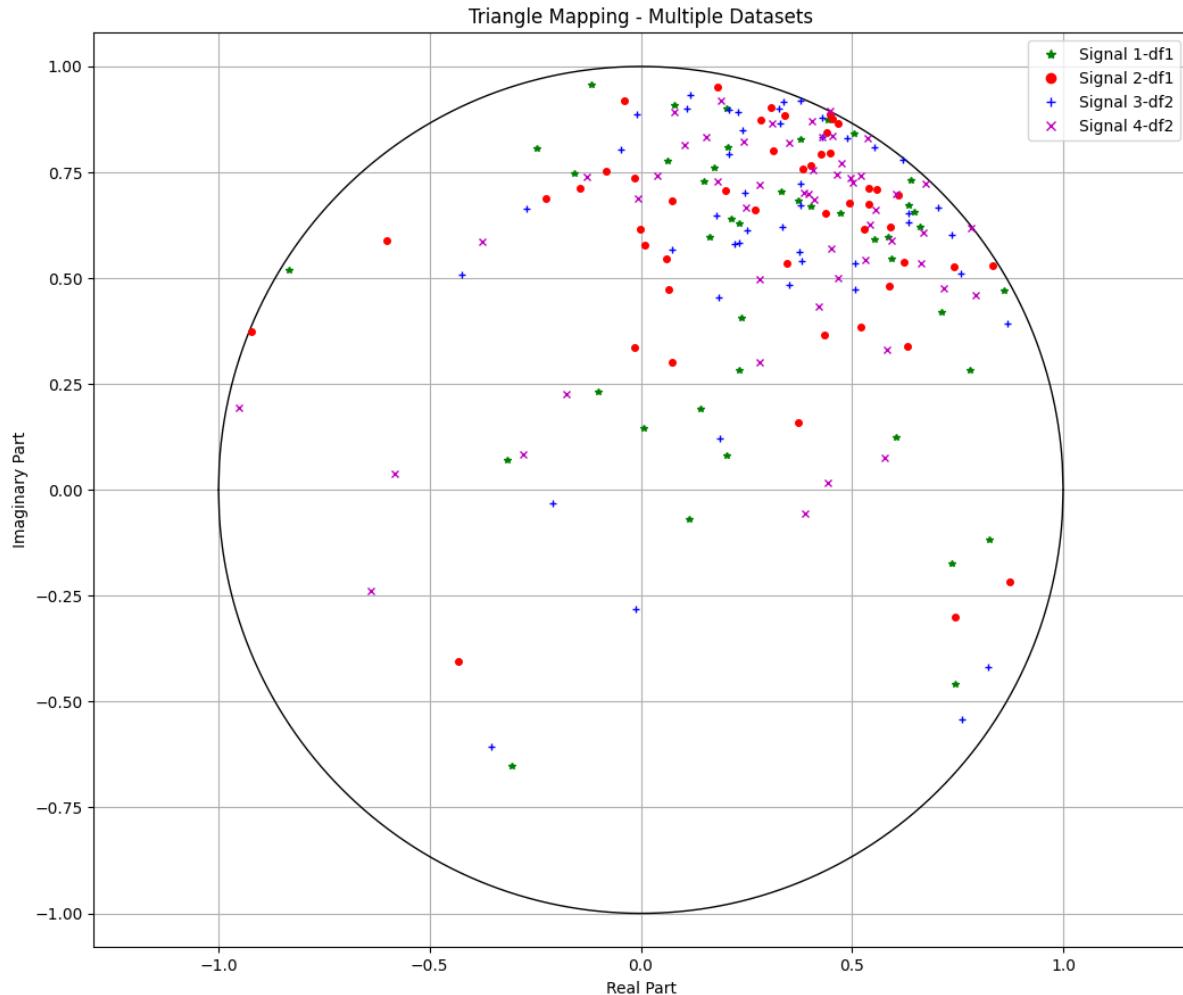


Figure 2: Mapeo de los triángulos al plano complejo

3. Calcular la distancia entre Clusters usando la distancia Euclideana

Para calcular la distancia entre los clusters se utilizan las siguientes funciones:

```

def calculate_distances(points):
    n = len(points)
    distances = np.zeros((n, n))
    distancias = []

    for i in range(n):
        for j in range(i + 1, n):
            z1 = points[i]
            z2 = points[j]
            d = np.sqrt((z1.real - z2.real) ** 2 + (z1.imag - z2.imag) ** 2)
            distances[i, j] = d
            distances[j, i] = d
            distancias.append(d)

    avg_distance = sum(distancias) / len(distancias) if distancias else 0
    return distances, avg_distance

def calculate_signal_distances(all_points_by_signal):
    num_signals = len(all_points_by_signal)
    distance_matrix = {}

    for i in range(num_signals):
        for j in range(num_signals):
            key = f"Signal {i+1} - Signal {j+1}"

            if i == j:
                # distance_matrix[key] = 0.0
                continue

            points_i = all_points_by_signal[i]
            points_j = all_points_by_signal[j]

            if len(points_i) != len(points_j):
                distance = compute_average_distance(points_i, points_j)
                distance_matrix[key] = distance
                continue

            identical = all(
                np.allclose(p1, p2, rtol=1e-12, atol=1e-12)
                for p1, p2 in zip(points_i, points_j)
            )

            if identical:
                distance_matrix[key] = 0.0
            else:
                distance = compute_average_distance(points_i, points_j)
                distance_matrix[key] = distance

    return distance_matrix

def compute_average_distance(points_i, points_j):

```

```

total_distance = 0
for p1 in points_i:
    for p2 in points_j:
        d = np.sqrt((p1.real - p2.real) ** 2 + (p1.imag - p2.imag) ** 2)
        total_distance += d
return total_distance / (len(points_i) * len(points_j))

```

La función *calculate_distances()* calcula las distancias entre todos los puntos en la lista *points*. *calculate_signal_distances()* calcula las distancias entre los clusters de cada señal y *compute_average_distance()* calcula la distancia media utilizando la ecuación (1).

$$D_{total} = \frac{\sum d(x_i, y_j)}{N_x * N_y}$$

Dando los siguientes resultados:

```

49 42 42 27
Global avg distance: 0.5579842541152327
Global distances: [[0. 0.14369518 0.35296864 ... 0.19786981 0.06875554 0
.24856741]
[0.14369518 0. 0.41894815 ... 0.18079596 0.10806736 0.30931939]
[0.35296864 0.41894815 0. ... 0.54845819 0.32552978 0.58414793]
...
[0.19786981 0.18079596 0.54845819 ... 0. 0.2283102 0.15705535]
[0.06875554 0.10806736 0.32552978 ... 0.2283102 0. 0.30840719]
[0.24856741 0.30931939 0.58414793 ... 0.15705535 0.30840719 0. ]]
Distancias entre senales:
Signal 1 - Signal 2: 0.589445
Signal 1 - Signal 3: 0.595380
Signal 1 - Signal 4: 0.580178
Signal 2 - Signal 1: 0.589445
Signal 2 - Signal 3: 0.542467
Signal 2 - Signal 4: 0.524381
Signal 3 - Signal 1: 0.595380
Signal 3 - Signal 2: 0.542467
Signal 3 - Signal 4: 0.532531
Signal 4 - Signal 1: 0.580178
Signal 4 - Signal 2: 0.524381
Signal 4 - Signal 3: 0.532531

```

Conclusiones

El mapeo de triángulos es una técnica novedosa para el reconocimiento de patrones por la manera en que simplifica la información de una señal a puntos dentro de un plano complejo. La distancia entre los clusters es un indicador importante para el reconocimiento de patrones. Si bien los resultados obtenidos no son los esperados, lo cual puede ser debido al modo como se manejan los datos, una revisión detallada de las funciones puede mejorar los resultados.



TAREA 5: AGRUPAMIENTO ESPECTRAL

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

Usando datos de sus tesis o de algún estudio, implementar en python el método de agrupamiento espectral.

- Para empezar el metodo de agrupamiento espectral, tenemos que generar una matriz de distancias euclidianas entre los puntos de un conjunto de datos. Para asegurarse que el codigo funcione correctamente, se recreará el ejercicio de clase.

```
pointstest = np.array(
    [[2, 5], [1, 1], [3, 2], [5, 3], [6, 4]]
) # Datos de prueba hechos en clase
```

```
# Usando datos de sus tesis o de algun estudio, implementar en python el metodo de
# agrupamiento espectral.

import pandas as pd
import numpy as np

def euclidean_distance(x1, x2):
    distance = np.linalg.norm(x1 - x2)
    return distance

def MatA(pointslist):
    A = np.zeros((len(pointslist), len(pointslist)))

    for i in range(len(pointslist)):
        for j in range(len(pointslist)):
            A[i][j] = euclidean_distance(pointslist[i], pointslist[j])
    return A
```

```
print(MatA(pointstest))

[0. 4.12310563 3.16227766 3.60555128 4.12310563]
[4.12310563 0. 2.23606798 4.47213595 5.83095189]
[3.16227766 2.23606798 0. 2.23606798 3.60555128]
[3.60555128 4.47213595 2.23606798 0. 1.41421356]
[4.12310563 5.83095189 3.60555128 1.41421356 0. ]
```

- El segundo paso es generar la matriz diagonal de acuerdo a la ecuacion $D_{ii} = \sum_j A_{ij}$

```
def MatDiagonal(pointslist):
    matrixA = MatA(pointslist)
    diag = np.zeros(matrixA.shape)
    for i in range(len(matrixA)):
        diag[i][i] = sum(matrixA[i])
    # print(diag)
    return diag
```

```
print(MatDiagonal(pointstest))

[[15.01404019 0. 0. 0. 0. ]
 [ 0. 16.66226145 0. 0. 0. ]
 [ 0. 0. 11.23996489 0. 0. ]
 [ 0. 0. 0. 11.72796877 0. ]
 [ 0. 0. 0. 0. 14.97382236]]
```

- El tercer paso es calcular la matriz Laplaciana con la ecuación $\mathcal{L} = D - A$

```
def Laplacian(pointslist):
    A = MatA(pointslist)
    D = MatDiagonal(pointslist)
    L = D - A
    # print(L)
    return L
```

```
print(Laplacian(pointstest))

[[15.01404019 -4.12310563 -3.16227766 -3.60555128 -4.12310563]
 [-4.12310563 16.66226145 -2.23606798 -4.47213595 -5.83095189]
 [-3.16227766 -2.23606798 11.23996489 -2.23606798 -3.60555128]
 [-3.60555128 -4.47213595 -2.23606798 11.72796877 -1.41421356]
 [-4.12310563 -5.83095189 -3.60555128 -1.41421356 14.97382236]]
```

- El cuarto paso es calcular los eigenvalores y eigenvalores de la matriz Laplaciana. Estos serán ordenados de menor a mayor.

```
def Eigen(pointslist):
    L = Laplacian(pointslist)
    eigenvalues, eigenvectors = np.linalg.eig(L)

    idx = eigenvalues.argsort()
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    return eigenvalues, eigenvectors
```

```
eigenvalues, eigenvectors = Eigen(pointstest)
print("Eigenvalues:\n", eigenvalues)
print("Eigenvectors:\n", eigenvectors)

Eigenvalues:
[3.55271368e-15 1.31212496e+01 1.50872446e+01 1.89753795e+01
 2.24341839e+01]
Eigenvectors:
[[-0.4472136 -0.049275 0.1981772 0.87069438 -0.01375012]
 [-0.4472136 -0.15165554 0.32178108 -0.29945827 0.76405646]
 [-0.4472136 0.67817809 -0.57296169 -0.0594764 0.09084014]
 [-0.4472136 -0.6862645 -0.48706082 -0.1617268 -0.25623667]
 [-0.4472136 0.20901695 0.54006424 -0.35003291 -0.58490981]]
```

- Por ultimo, utilizamos el eigenvalor dominante (el ultimo) y su eigenvector asociado para la agrupación de los datos.

```
def Clustering(pointslist):
    _, eigenvectors = Eigen(pointslist)

    newpoints = -eigenvectors[:, -1]
    Cluster = []
    for i in range(len(newpoints)):
        if newpoints[i] > 0:
            newpoint = 1
            Cluster.append(newpoint)
        if newpoints[i] < 0:
            newpoint = 0
            Cluster.append(newpoint)
    return Cluster
```

Si el valor del punto es positivo, entonces se agrupan en 1, si es negativo se agrupan en 0

```
clusterstest = Clustering(pointstest)
print("\n Cluster de apuntes de clase:", clusterstest)
```

```
Cluster de apuntes de clase: [1, 0, 0, 1, 1]
```

Una vez asegurandonos que el codigo funciona de manera correcta, podemos insertar nuestros propios datos para generrar los clusters.

Para esto se van a utilizar 2 listas que consisten en un par de masas de cuerpos astronomicos los cuales generan un evento.

```
df = pd.read_csv(
    "/Users/patrickbustamante/Library/CloudStorage/GoogleDrive-p317694@uach.mx/My Drive/
     ↳ Main/Maestria/Tesis/LSTM_test/dftest.csv"
)
df = df.drop(
    [
        "Unnamed: 0",
        "Zoomed",
        "time",
        "strain",
        "clean",
        "spin1",
        "spin2",
        "tilt1",
        "tilt2",
        "phi12",
        "phijl",
        "theta_jn",
        "psi",
        "phase",
        "geocent_time",
        "ra",
        "dec",
        "approximant",
        "reference_frequency",
    ]
)
```

```
"min_frequency",
"Zoomed",
],
axis=1,
)

x1 = df.iloc[:, 0].to_numpy() # Masa 1
x2 = df.iloc[:, 1].to_numpy() # Masa 2

points = np.column_stack((x1, x2)) # Crear tuplas de datos con las masas de los eventos
```

Una vez definida la tupla de datos, podemos utilizar la función Clustering para obtener los clusters.

```
clusters = Clustering(points)
print("Cluster assignments: \n", clusters)
```



TAREA 1: ALGORITMO DE VITERBI

Alumno:

Patricio Bustamante
Maestría en ingeniería en computación

Universidad:

Universidad Autónoma de Chihuahua
Facultad de ingeniería

Realizar un programa en python para encontrar la secuencia de estados más probable de un modelo HMM. Usar como base el artículo de Rabiner y el otro archivo adjunto.

Utilizando como base el artículo de Rabiner con el cual desarrollamos el script de clase y del libro Speech and Language Processing. Daniel Jurafsky & James H. Martin. Obtenemos que el algoritmo de Viterbi se comporta de la siguiente manera:

```
function VITERBI(observations of len  $T$ ,state-graph of len  $N$ ) returns best-path, path-prob
    create a path probability matrix viterbi[ $N,T$ ]
    for each state  $s$  from 1 to  $N$  do ; initialization step
        viterbi[ $s,1$ ]  $\leftarrow \pi_s * b_s(o_1)$ 
        backpointer[ $s,1$ ]  $\leftarrow 0$ 
    for each time step  $t$  from 2 to  $T$  do ; recursion step
        for each state  $s$  from 1 to  $N$  do
             $viterbi[s,t] \leftarrow \max_{s'=1}^N viterbi[s',t-1] * a_{s',s} * b_s(o_t)$ 
            backpointer[ $s,t$ ]  $\leftarrow \operatorname{argmax}_{s'=1}^N viterbi[s',t-1] * a_{s',s} * b_s(o_t)$ 
    bestpathprob  $\leftarrow \max_{s=1}^N viterbi[s,T]$  ; termination step
    bestpathpointer  $\leftarrow \operatorname{argmax}_{s=1}^N viterbi[s,T]$  ; termination step
    bestpath  $\leftarrow$  the path starting at state bestpathpointer, that follows backpointer[] to states back in time
    return bestpath, bestpathprob
```

Figure A.9 Viterbi algorithm for finding optimal sequence of hidden states. Given an observation sequence and an HMM $\lambda = (A, B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence.

Figure 1: pseudo-código del algoritmo de Viterbi

De esta manera, se puede implementar el algoritmo de Viterbi en nuestro script.

```
for n in range(0, N):
    # Ciclo para obtener tamano de la secuencia
    T = M
    for m in range(0, T):
        if ms[n, m] == 0:
            T = m
            print(T)
            break

V = np.zeros((Q, T))
B = np.zeros((Q, T))
for i in range(0, Q):
    V[i, 0] = h[i, 0] * bjk[i, ms[n, 0] - 1]
    B[i, 0] = 0

    for t in range(1, T):
        for j in range(0, Q):
            vals = []
```

```

for i in range(0, Q):
    val = V[i, t - 1] * aij[i, j] * bjk[j, ms[n, t] - 1]
    vals.append(val)
V[j, t] = max(vals)
B[j, t] = np.argmax(vals)

bestpathprob = max(V[:, T - 1])
bestpath = np.zeros((1, T))
bestpath[0, T - 1] = np.argmax(V[:, T - 1])

print(bestpath + 1)
print(bestpathprob)

```

Utilizando el ejemplo de la presentación

An Example

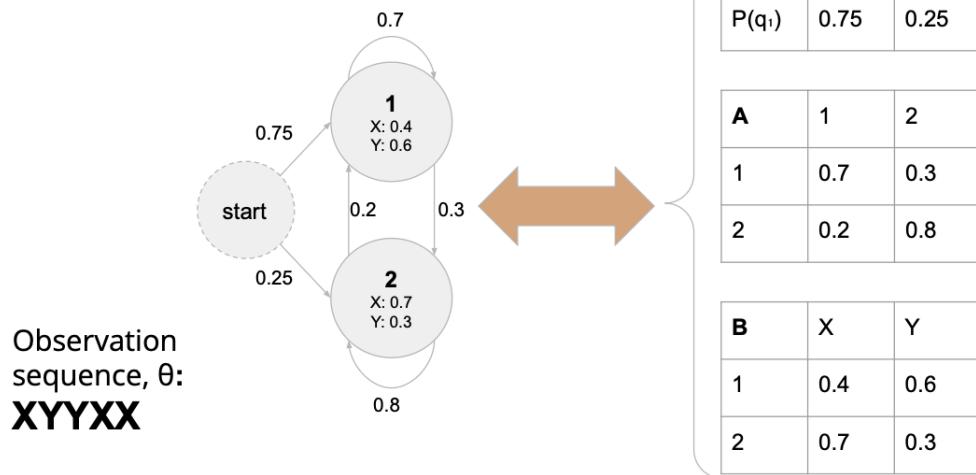


Figure 2: Ejemplo de secuencia

Podemos implementarlo en nuestro código actual

```

# Matriz de probabilidad de transiciones entre estados.
aij = np.array([[0.7, 0.3], [0.2, 0.8]])
# Matriz de probabilidades de producción de símbolos.
bjk = np.array([[0.4, 0.6], [0.7, 0.3]])
# Vector de probabilidades de iniciar en un estado.
h = np.array([[0.75], [0.25]])
# MATRIZ DE SECUENCIAS DE OBSERVACIONES
ms = np.array(
    [
        [1, 2, 2, 1, 1, 0], # XYYXX
    ]
)

```

Obteniendo de esta manera la secuencia de estados y la probabilidad de la misma.

```
[[1. 1. 1. 1. 2.]]  
0.005448001119516439
```