Logotipo

Descripción generada automáticamenteImagen que contiene señal, firmar, parada, tráfico

Descripción generada automáticamente

INSTITUTO POLITÉCTICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

**UNIDAD DE APRENDIZAJE:**

MACHINE LEARNING

**PRÁCTICA 5:** K-MEANS

**INTEGRANTES:**

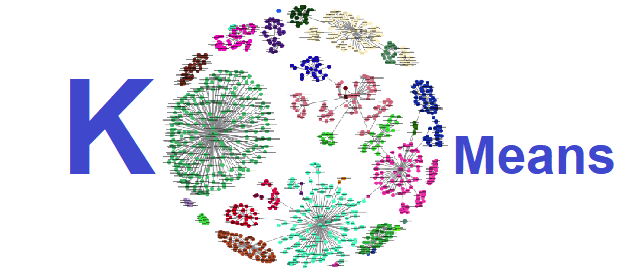
Hernández Hernández Roberto Isaac

Gonzalez Llamosas Noe Ramses

**PROFESOR:**

Ortiz Castillo Marco Antonio

**FECHA DE ENTREGA:** 15/01/2025



**CONTENIDO**

[1.PRÁCTICA: K-MEANS 3](#_Toc187844831)

[1.1. INTRODUCCIÓN 3](#_Toc187844832)

[1.1.1. Funcionamiento del algoritmo 3](#_Toc187844833)

[1.1.2. K-means ++ 4](#_Toc187844834)

[1.2. DESARROLLO 5](#_Toc187844835)

[1.3. CONCLUSIÓN 21](#_Toc187844836)

[2.PRÁCTICA: SUPPORT VECTOR MACHINE 23](#_Toc187844837)

[2.1. INTRODUCCIÓN 23](#_Toc187844838)

[2.1.1. Funcionamiento de las SVM 24](#_Toc187844839)

[2.2. DESARROLLO 25](#_Toc187844840)

[2.3. CONCLUSIÓN 33](#_Toc187844841)

# PRÁCTICA: K-MEANS

## INTRODUCCIÓN

K-means clustering es un algoritmo de aprendizaje no monitorear que se emplea para la agrupación de datos, que agrupa puntos de datos no etiquetados en grupos o clústeres. Es uno de los métodos de agrupación más populares utilizados en el machine learning. A diferencia del aprendizaje supervisado, los datos de entrenamiento que utiliza este algoritmo no están etiquetados, lo que significa que los puntos de datos no tienen una estructura de clasificación definida. Si bien existen varios tipos de algoritmos de agrupamiento, incluidos los exclusivos, superpuestos, jerárquicos y probabilísticos. Esta forma de agrupación estipula que un punto de datos puede existir en un solo clúster. Este tipo de análisis de clústeres se usa comúnmente en la ciencia de datos para la segmentación del mercado, la agrupación de documentos, la segmentación de imágenes y la compresión de imágenes. El algoritmo k-means es un método ampliamente empleado en el análisis de conglomerados porque es eficiente, eficaz y sencillo [1].

K-means es un algoritmo de agrupamiento iterativo basado en centroides que divide un conjunto de datos en grupos similares en función de la distancia entre sus centroides. El centroide, o centro del clúster, es la media o la mediana de todos los puntos dentro del clúster, según las características de los datos [1].

### Funcionamiento del algoritmo

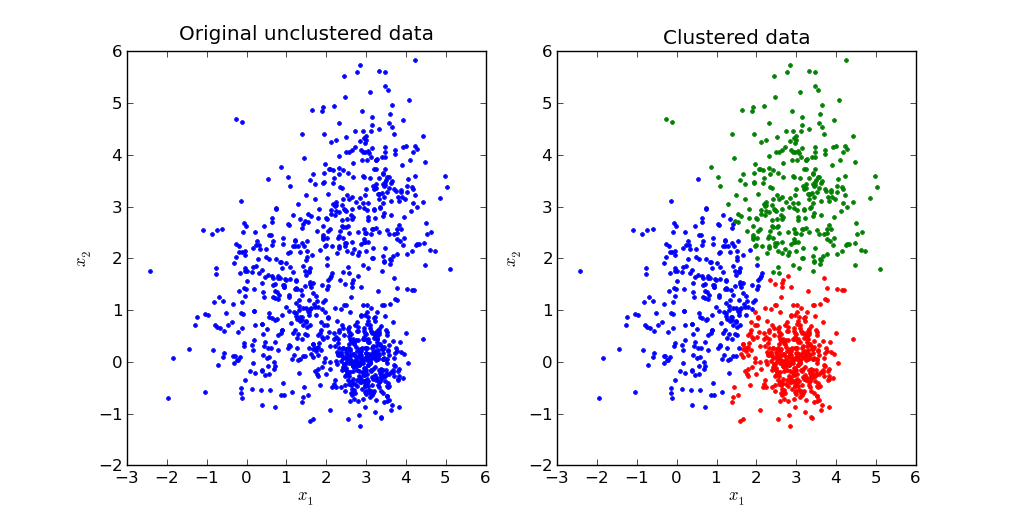
La agrupación en clústeres de K-means es un proceso iterativo para minimizar la suma de distancias entre los puntos de datos y sus centroides de clúster. El algoritmo de agrupamiento de medias k funciona categorizando puntos de datos en clústeres utilizando una medida matemática de distancia, generalmente euclidiana, desde el centro del clúster. El objetivo es minimizar la suma de distancias entre los puntos de datos y sus clústeres asignados. Los puntos de datos más cercanos a un centroide se agrupan dentro de la misma categoría. Un valor k más alto, o el número de conglomerados, significa conglomerados más pequeños con mayor detalle, mientras que un valor k más bajo da lugar a conglomerados más grandes con menos detalle [1].

* **Inicializar K:** El primer paso es inicializar k centroides donde k es igual al número de clústeres elegidos para un conjunto de datos específico. Este enfoque utiliza métodos de selección aleatoria o muestreo centroide inicial.
* **Asignar centroides:** El siguiente paso incluye un proceso iterativo de dos etapas basado en el algoritmo de machine learning de maximización de expectativas. El paso de expectativa asigna cada punto de datos a su centroide más cercano en función de la distancia (de nuevo, normalmente euclídea). El paso de maximización calcula la media de todos los puntos de cada conglomerado y reasigna el centro del conglomerado, o centroide. Este proceso se repite hasta que las posiciones de los centroides hayan alcanzado la convergencia o se haya alcanzado el número máximo de iteraciones [1].

El agrupamiento de medias K es simple pero sensible a las condiciones iniciales y los valores atípicos. Es importante optimizar la inicialización del centroide y el número de clústeres k, para lograr los clústeres más significativos. Hay varias formas de evaluar y optimizar los componentes de agrupación del algoritmo mediante el uso de métricas de evaluación y métodos de muestreo de centroide inicial [1].

### K-means ++

K-means++ es un algoritmo k-means que optimiza la selección del centroide o centroides iniciales del clúster. Desarrollado por los investigadores Arthur y Vassilvitskii, k-means++ mejora la calidad de la asignación final del clúster. El primer paso para la inicialización mediante el método de medias k++ es elegir un centroide del conjunto de datos. Para cada centroide subsiguiente, calcule la distancia de cada punto de datos desde su centro de clúster más cercano. El siguiente centroide se selecciona teniendo en cuenta la probabilidad de que un punto se encuentre a una distancia proporcional del centroide más cercano elegido anteriormente. El proceso ejecuta iteraciones hasta que se haya inicializado el número elegido de centros de clúster. En la siguiente figura se muestra la manera en que se clasifica de forma óptima un conjunto de datos mediante el K-means++.



**Figura.** Ejemplo de clasificación mediante K-means++. Fuente: [1].

## DESARROLLO

A manera de observar el comportamiento de este algoritmo es necesario implementar los siguientes puntos:

1. Crear 8 clases de puntos (20 puntos por clase) e implementar K-means, pruebe para diferentes valores de K.
2. Cambie el método de inicialización a K-means++ o considerando dispersión compare sus resultados con el punto anterior.
3. Sea el problema del titanic, realizar un programa que en cada muestra tenga las siguientes características eligiendo K=2, determine para 100 muestras como se clasifica.

#### Diagrama de flujo

**KMEANS**

Escala de tiempo

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Escala de tiempo

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

#### Códigos

**ALGORITMO KMEANS**

**import** matplotlib.pyplot **as** plt  
**import** random  
**import** numpy **as** np  
  
**def** distancia\_ecluidiana(X, x\_test):  
    x\_test = np.array(x\_test)  
    d = (X - x\_test) \*\* 2  
    distancia = [np.sqrt(d[i][0] + d[i][1]) **for** i **in** range(len(d))]  
    **return** distancia

**def** Kmeans(data, k, epocas):  
    data = np.array(data)  
    num\_datos, num\_caracteristicas = data.shape  
  
    *# Inicializar aleatoriamente los centroides sin duplicados    indices\_usados = [-1] \* k*  
    centroides = [[0] \* num\_caracteristicas **for** \_ **in** range(k)]  
    **for** i **in** range(k):  
        **while** **True**:  
            random\_centroides = random.randint(0, num\_datos - 1)  
            **if** random\_centroides **not** **in** indices\_usados:  
                indices\_usados[i] = random\_centroides  
                **for** j **in** range(num\_caracteristicas):  
                    centroides[i][j] = data[random\_centroides][j]  
                **break**  
  
    *# Iteraciones para calcular los clústeres y actualizar los centroides*  
    **for** iteraciones **in** range(epocas):  
        clouster\_asignados = [0] \* num\_datos  
        **for** i **in** range(num\_datos):  
            distancias = [distancia\_ecluidiana([centroide], data[i])[0] **for** centroide **in** centroides]  
            clouster\_asignados[i] = np.argmin(distancias)  
  
        *# Actualizar centroides*  
        Nuevos\_Centroides = [[0] \* num\_caracteristicas **for** \_ **in** range(k)]  
        puntos\_por\_clouster = [0] \* k  
        **for** i **in** range(num\_datos):  
            clouster = clouster\_asignados[i]  
            puntos\_por\_clouster[clouster] += 1  
            **for** j **in** range(num\_caracteristicas):  
                Nuevos\_Centroides[clouster][j] += data[i][j]  
  
        **for** clouster\_index **in** range(k):  
            **if** puntos\_por\_clouster[clouster\_index] > 0:  
                **for** j **in** range(num\_caracteristicas):  
                    Nuevos\_Centroides[clouster\_index][j] /= puntos\_por\_clouster[clouster\_index]  
            **else**:  
                **for** j **in** range(num\_caracteristicas):  
                    Nuevos\_Centroides[clouster\_index][j] = centroides[clouster\_index][j]  
  
        *# Verificar convergencia*  
        variacion = **True**  
        epsilon = 1e-10  
        **for** i **in** range(k):  
            **for** j **in** range(num\_caracteristicas):  
                **if** abs(Nuevos\_Centroides[i][j] - centroides[i][j]) > epsilon:  
                    variacion = **False**  
                    **break**  
            **if** **not** variacion:  
                **break**  
        **if** variacion:  
            **break**  
  
        centroides = Nuevos\_Centroides  
  
    *# Agrupamiento final*  
    clousters= [[] **for** \_ **in** range(k)]  
    **for** i **in** range(num\_datos):  
        clousters[clouster\_asignados[i]] = clousters[clouster\_asignados[i]] + [data[i]]  
     
    **return** centroides, clousters  
  
**def** graficar\_puntos(data, clousters, centroides):  
    colors = [**'red'**, **'blue'**, **'green'**, **'purple'**, **'orange'**, **'cyan'**, **'brown'**, **'magenta'**]  
    **for** clouster\_index **in** range(len(clousters)):  
        x\_points = [clousters[clouster\_index][i][0] **for** i **in** range(len(clousters[clouster\_index]))]  
        y\_points = [clousters[clouster\_index][i][1] **for** i **in** range(len(clousters[clouster\_index]))]  
        plt.scatter(x\_points, y\_points, color=colors[clouster\_index % len(colors)], label=**f'Clúster {clouster\_index + 1}'**)  
    x\_centroides = [centroides[i][0] **for** i **in** range(len(centroides))]  
    y\_centroides = [centroides[i][1] **for** i **in** range(len(centroides))]  
    plt.scatter(x\_centroides, y\_centroides, color=**'black'**, marker=**'x'**, s=100, label=**'Centroides'**)  
    plt.xlabel(**"x-axis"**)  
    plt.ylabel(**"y-axis"**)  
    plt.title(**"K-means"**)  
    plt.legend()  
    plt.grid(**True**)  
    plt.show()  
  
*# Datos de prueba*  
data = [  
    *# Clase 1: Alrededor de (-50, -50)*  
    [random.uniform(-55, -35), random.uniform(-55, -35)] **for** \_ **in** range(20)  
] + [  
    *# Clase 2: Alrededor de (-20, 20)*  
    [random.uniform(-25, -5), random.uniform(15, 35)] **for** \_ **in** range(20)  
] + [  
    *# Clase 3: Alrededor de (0, -30)*  
    [random.uniform(-5, 15), random.uniform(-35, -15)] **for** \_ **in** range(20)  
] + [  
    *# Clase 4: Alrededor de (40, 40)*  
    [random.uniform(35, 55), random.uniform(35, 55)] **for** \_ **in** range(20)  
] + [  
    *# Clase 5: Alrededor de (60, -60)*  
    [random.uniform(55, 75), random.uniform(-65, -45)] **for** \_ **in** range(20)  
] + [  
    *# Clase 6: Alrededor de (-70, 70)*  
    [random.uniform(-75, -55), random.uniform(65, 85)] **for** \_ **in** range(20)  
] + [  
    *# Clase 7: Alrededor de (80, 10)*  
    [random.uniform(75, 95), random.uniform(5, 25)] **for** \_ **in** range(20)  
] + [  
    *# Clase 8: Alrededor de (-90, -90)*  
    [random.uniform(-95, -75), random.uniform(-95, -75)] **for** \_ **in** range(20)  
]  
  
k = 8  
epocas = 100  
centroides, clousters = Kmeans(data, k, epocas)  
graficar\_puntos(data, clousters, centroides)

**ALGORITMO KMEANS++**

**import** matplotlib.pyplot **as** plt  
**import** random  
**import** numpy **as** np  
  
**def** distancia\_ecluidiana(X, x\_test):  
    x\_test = np.array(x\_test)  
    d = (X - x\_test) \*\* 2  
    distancia = [np.sqrt(d[i][0] + d[i][1]) **for** i **in** range(len(d))]  
    **return** distancia  
  
**def** inicializar\_centroides(data, k):  
    data = np.array(data)  
    num\_datos = len(data)  
     
    *# Inicializar el array de centroides con tamaño k x num\_caracteristicas*  
    num\_caracteristicas = data.shape[1]  
    centroides = np.zeros((k, num\_caracteristicas))  
     
    *# Seleccionar el primer centroide al azar*  
    primer\_centroide\_idx = random.randint(0, num\_datos - 1)  
    centroides[0] = data[primer\_centroide\_idx]  
     
    **for** i **in** range(1, k):  
        *# Calcular las distancias mínimas al conjunto actual de centroides*  
        distancias\_minimas = np.zeros(len(data))  *# Crear un array inicializado con ceros*  
        **for** idx, punto **in** enumerate(data):  
            distancias = [np.linalg.norm(punto - centroides[j]) **for** j **in** range(i)]  
            distancias\_minimas[idx] = min(distancias)  *# Asignar el mínimo directamente al índice correspondiente*  
  
        *# Elegir el siguiente centroide con probabilidad proporcional al cuadrado de la distancia*  
        distancias\_cuadradas = distancias\_minimas \*\* 2  
        probabilidades = distancias\_cuadradas / distancias\_cuadradas.sum()  
        siguiente\_centroide\_idx = np.random.choice(range(len(data)), p=probabilidades)  
        centroides[i] = data[siguiente\_centroide\_idx]  
  
    **return** centroides  
  
**def** Kmeans(data, k, epocas):  
    data = np.array(data)  
    num\_datos, num\_caracteristicas = data.shape  
  
    *# Inicializar centroides usando K-Means++*  
    centroides = inicializar\_centroides(data, k)  
  
    *# Iteraciones para calcular los clústeres y actualizar los centroides*  
    **for** iteraciones **in** range(epocas):  
        clouster\_asignados = [0] \* num\_datos  
        **for** i **in** range(num\_datos):  
            distancias = [distancia\_ecluidiana([centroide], data[i])[0] **for** centroide **in** centroides]  
            clouster\_asignados[i] = np.argmin(distancias)  
  
        *# Actualizar centroides*  
        Nuevos\_Centroides = [[0] \* num\_caracteristicas **for** \_ **in** range(k)]  
        puntos\_por\_clouster = [0] \* k  
        **for** i **in** range(num\_datos):  
            clouster = clouster\_asignados[i]  
            puntos\_por\_clouster[clouster] += 1  
            **for** j **in** range(num\_caracteristicas):  
                Nuevos\_Centroides[clouster][j] += data[i][j]  
  
        **for** clouster\_index **in** range(k):  
            **if** puntos\_por\_clouster[clouster\_index] > 0:  
                **for** j **in** range(num\_caracteristicas):  
                    Nuevos\_Centroides[clouster\_index][j] /= puntos\_por\_clouster[clouster\_index]  
            **else**:  
                **for** j **in** range(num\_caracteristicas):  
                    Nuevos\_Centroides[clouster\_index][j] = centroides[clouster\_index][j]  
  
        *# Verificar convergencia*  
        variacion = **True**  
        epsilon = 1e-10  
        **for** i **in** range(k):  
            **for** j **in** range(num\_caracteristicas):  
                **if** abs(Nuevos\_Centroides[i][j] - centroides[i][j]) > epsilon:  
                    variacion = **False**  
                    **break**  
            **if** **not** variacion:  
                **break**  
        **if** variacion:  
            **break**  
  
        centroides = Nuevos\_Centroides  
  
    *# Agrupamiento final*  
    clousters = [[] **for** \_ **in** range(k)]  
    **for** i **in** range(num\_datos):  
        clousters[clouster\_asignados[i]] = clousters[clouster\_asignados[i]] + [data[i]]  
     
    **return** centroides, clousters  
  
**def** graficar\_puntos(data, clousters, centroides):  
    colors = [**'red'**, **'blue'**, **'green'**, **'purple'**, **'orange'**, **'cyan'**, **'brown'**, **'magenta'**]  
    **for** clouster\_index **in** range(len(clousters)):  
        x\_points = [clousters[clouster\_index][i][0] **for** i **in** range(len(clousters[clouster\_index]))]  
        y\_points = [clousters[clouster\_index][i][1] **for** i **in** range(len(clousters[clouster\_index]))]  
        plt.scatter(x\_points, y\_points, color=colors[clouster\_index % len(colors)], label=**f'Clúster {clouster\_index + 1}'**)  
    x\_centroides = [centroides[i][0] **for** i **in** range(len(centroides))]  
    y\_centroides = [centroides[i][1] **for** i **in** range(len(centroides))]  
    plt.scatter(x\_centroides, y\_centroides, color=**'black'**, marker=**'x'**, s=100, label=**'Centroides'**)  
    plt.xlabel(**"x-axis"**)  
    plt.ylabel(**"y-axis"**)  
    plt.title(**"K-means++"**)  
    plt.legend()  
    plt.grid(**True**)  
    plt.show()  
  
  
*# Datos de prueba dispersos*  
data = [  
    *# Clase 1: Alrededor de (-50, -50)*  
    [random.uniform(-55, -35), random.uniform(-55, -35)] **for** \_ **in** range(20)  
] + [  
    *# Clase 2: Alrededor de (-20, 20)*  
    [random.uniform(-25, -5), random.uniform(15, 35)] **for** \_ **in** range(20)  
] + [  
    *# Clase 3: Alrededor de (0, -30)*  
    [random.uniform(-5, 15), random.uniform(-35, -15)] **for** \_ **in** range(20)  
] + [  
    *# Clase 4: Alrededor de (40, 40)*  
    [random.uniform(35, 55), random.uniform(35, 55)] **for** \_ **in** range(20)  
] + [  
    *# Clase 5: Alrededor de (60, -60)*  
    [random.uniform(55, 75), random.uniform(-65, -45)] **for** \_ **in** range(20)  
] + [  
    *# Clase 6: Alrededor de (-70, 70)*  
    [random.uniform(-75, -55), random.uniform(65, 85)] **for** \_ **in** range(20)  
] + [  
    *# Clase 7: Alrededor de (80, 10)*  
    [random.uniform(75, 95), random.uniform(5, 25)] **for** \_ **in** range(20)  
] + [  
    *# Clase 8: Alrededor de (-90, -90)*  
    [random.uniform(-95, -75), random.uniform(-95, -75)] **for** \_ **in** range(20)  
]  
  
k = 8  
epocas = 100  
centroides, clousters = Kmeans(data, k, epocas)  
graficar\_puntos(data, clousters, centroides)

**ALGORITMO KMEANS++ PARA PROBLEMÁTICA DEL TITANIC**

**import** pandas **as** pd  
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**from** mpl\_toolkits.mplot3d **import** Axes3D  
  
**def** entrenamiento\_MSV(X, Y):  
    numero\_muestras, numero\_caracteristicas = X.shape  
  
    *# Inicialización parámetros externos*  
    epocas = 1000  
    lr = 0.01  
    lamda = 1 / epocas  
  
    *# Inicialización de parámetros internos*  
    w = np.zeros(numero\_caracteristicas) + 0.1  
    b = 0.1  
  
    **for** epoca **in** range(epocas):  
        **for** i, x **in** enumerate(X):  
            condicion\_margen = Y[i] \* (np.dot(x, w) + b) >= 1  
            **if** condicion\_margen:  
                w = w - lr \* (2 \* lamda \* w)  
            **else**:  
                w = w - lr \* (2 \* lamda \* w - np.dot(Y[i], x))  
                b = b - lr \* lamda \* Y[i]  
  
    tolerancia = 1  
    vectores\_soporte\_indices = [  
        i **for** i, x **in** enumerate(X)  
        **if** abs(Y[i] \* (np.dot(x, w) + b) - 1) <= tolerancia  
    ]  
    vectores\_soporte = X[vectores\_soporte\_indices]  
    clases\_vectores\_soporte = Y[vectores\_soporte\_indices]  
  
    **return** w, b, vectores\_soporte, vectores\_soporte\_indices, clases\_vectores\_soporte  
  
**def** prediccion\_MSV(X\_test, w, b):  
    **return** np.sign(np.dot(X\_test, w) + b)  
  
*# Cargar los datos*  
ruta\_archivo = **'./train.csv'**  
datos = pd.read\_csv(ruta\_archivo)  
  
Y = datos[**'Survived'**].tolist()  
vivos = [y **for** y **in** Y **if** y == 1][:30]  
muertos = [y **for** y **in** Y **if** y == 0][:30]  
Y\_fixed = np.array(vivos + muertos)  
  
indices\_vivos = [i **for** i, y **in** enumerate(Y) **if** y == 1][:30]  
indices\_muertos = [i **for** i, y **in** enumerate(Y) **if** y == 0][:30]  
indices\_filtrados = indices\_vivos + indices\_muertos  
  
SibSp = datos[**'SibSp'**].tolist()  
Age = datos[**'Age'**].tolist()  
Pclass = datos[**'Pclass'**].tolist()  
  
moda\_sibsp = max(set(SibSp), key=SibSp.count)  
SibSp = [moda\_sibsp **if** pd.isna(s) **else** s **for** s **in** SibSp]  
SibSp = [moda\_sibsp **if** s < 0 **or** **not** isinstance(s, int) **else** s **for** s **in** SibSp]  
SibSp = [int(s) **for** s **in** SibSp]  
  
Age = [np.nan **if** pd.isna(a) **else** a **for** a **in** Age]  
promedio\_edad = np.nanmean(Age)  
Age = [promedio\_edad **if** np.isnan(a) **else** a **for** a **in** Age]  
  
moda\_clase = max(set(Pclass), key=Pclass.count)  
Pclass = [moda\_clase **if** pd.isna(c) **else** c **for** c **in** Pclass]  
Pclass = [int(c) **for** c **in** Pclass]  
  
X = np.array(list(zip(Pclass, Age, SibSp)))  
X\_fixed = X[indices\_filtrados]  
  
w, b, vectores\_soporte, vectores\_soporte\_indices, clases\_vectores\_soporte = entrenamiento\_MSV(X\_fixed, Y\_fixed)  
  
nuevo\_punto = []  
nuevo\_punto.append(int(input(**"Ingrese la clase de pasajero (Pclass, 1-3): "**)))  
nuevo\_punto.append(float(input(**"Ingrese la edad del pasajero: "**)))  
nuevo\_punto.append(int(input(**"Ingrese el número de hermanos/esposos (SibSp): "**)))  
nuevo\_punto = np.array(nuevo\_punto)  
clase\_predicha = prediccion\_MSV(nuevo\_punto, w, b)  
  
print(**f"El nuevo punto pertenece a la clase: {'Sobrevivió' if clase\_predicha == 1 else 'No sobrevivió'}"**)  
  
print(**"Vectores de soporte:"**)  
**for** i, vector **in** enumerate(vectores\_soporte):  
    print(**f"Vector: {vector}, Clase: {clases\_vectores\_soporte[i]}"**)  
  
fig = plt.figure()  
ax = fig.add\_subplot(111, projection=**'3d'**)  
ax.scatter(X\_fixed[:, 0], X\_fixed[:, 1], X\_fixed[:, 2], c=Y\_fixed, cmap=plt.cm.Paired, label=**'Datos'**)  
  
ax.scatter(X\_fixed[vectores\_soporte\_indices, 0],  
           X\_fixed[vectores\_soporte\_indices, 1],  
           X\_fixed[vectores\_soporte\_indices, 2],  
           s=150, facecolors=**'yellow'**, edgecolors=**'red'**, linewidths=2, label=**'Vectores de Soporte'**)  
  
ax.scatter(nuevo\_punto[0], nuevo\_punto[1], nuevo\_punto[2],  
           color=**'green'**, marker=**'o'**, s=150, label=**'Nuevo punto'**)  
  
*# Hiperplano y márgenes*  
xx, yy = np.meshgrid(np.linspace(X\_fixed[:, 0].min(), X\_fixed[:, 0].max(), 10),  
                     np.linspace(X\_fixed[:, 1].min(), X\_fixed[:, 1].max(), 10))  
zz = (-w[0] \* xx - w[1] \* yy - b) / w[2]  
ax.plot\_surface(xx, yy, zz, alpha=0.3, color=**'blue'**, label=**'Hiperplano'**)  
  
ax.set\_xlabel(**'Pclass'**)  
ax.set\_ylabel(**'Age'**)  
ax.set\_zlabel(**'SibSp'**)  
plt.title(**'Máquinas de Soporte Vectorial (MSV)'**)  
plt.legend()  
plt.show()

#### Funcionamiento

**ALGORITMO KMEANS**

Los resultados obtenidos tras la implementación del algoritmo K-means, considerando 8 clases distintas, cada una compuesta por 20 puntos y utilizando un valor de K=8, se presentan en la figura siguiente. Es importante analizar la distribución de los centroides, ya que su inicialización aleatoria puede conducir a la formación de clústeres de manera subóptima.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Clasificación del algoritmo para un K=8. Fuente: Elaboración propia.

Con el propósito de realizar múltiples pruebas con el algoritmo, se muestra su ejecución para un K=5 y un K=3. Los resultados se muestran a continuación.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Clasificación del algoritmo para un K=5. Fuente: Elaboración propia.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Clasificación del algoritmo para un K=3. Fuente: Elaboración propia.

**ALGORITMO KMEANS++**

En nuestro caso, optamos por implementar el algoritmo K-means++ para lograr una clasificación más precisa de cada una de las clases creadas. Asimismo, realizamos pruebas utilizando diferentes valores de K con el objetivo de analizar el comportamiento del algoritmo en distintos escenarios. En este caso, es crucial observar cómo el algoritmo realiza la clasificación de manera adecuada, asignando un clúster distinto a cada grupo creado.

Como se menciona en la introducción, este algoritmo selecciona el siguiente centroide de los puntos de datos de modo que la probabilidad de elegir un punto como centroide sea directamente proporcional a su distancia desde el centroide más cercano previamente elegido (es decir, el punto que tenga la distancia máxima desde el centroide más cercano tenga más probabilidades de ser seleccionado a continuación como centroide).

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Clasificación del algoritmo para un K=8. Fuente: Elaboración propia.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Clasificación del algoritmo para un K=5. Fuente: Elaboración propia.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Clasificación del algoritmo para un K=3. Fuente: Elaboración propia.

**ALGORITMO KMEANS++ PARA PROBLEMÁTICA DEL TITANIC**

Para resolver este ejercicio, se empleó el algoritmo K-means++, basándose en la implementación realizada en prácticas anteriores con el archivo CSV del Titanic. En este archivo, los datos fueron procesados y depurados adecuadamente para su uso. Además, se solicitó al usuario ingresar una clase, edad y sexo, permitiendo que el algoritmo determine si el nuevo punto ingresado pertenece a la categoría de sobrevivientes o fallecidos. En nuestro caso, se presenta únicamente uno de los dos clústeres, indicando la cantidad de datos asignados durante el entrenamiento.

Fue fundamental considerar una grafica tridimensional para mostrar las características y la clasificación de forma adecuada, de esta manera es posible interpretar la relación entre las variables de entrada (como clase, edad y sexo) de manera más clara y comprensible.

En las siguientes figuras se presenta de manera gráfica y en la terminal el resultado de la clasificación realizada, mostrando un ejemplo de una persona que sobrevivió y otro de una que falleció.

Interfaz de usuario gráfica, Texto

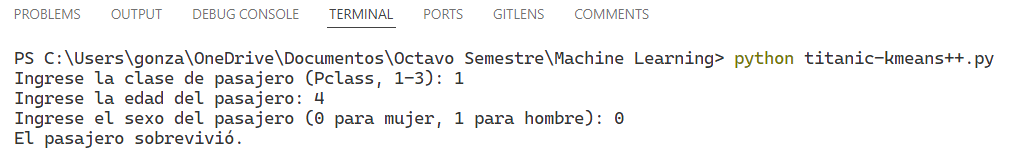
Descripción generada automáticamente

**Figura.** Resultado en terminal de un pasajero fallecido. Fuente: Elaboración propia.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Resultado en gráfica 3D de un pasajero fallecido. Fuente: Elaboración propia.



**Figura.** Resultado en terminal de un pasajero que sobrevivió. Fuente: Elaboración propia.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Resultado en gráfica 3D de un pasajero que sobrevivió. Fuente: Elaboración propia.

## CONCLUSIÓN

En el caso del algoritmo K-means, tanto en su versión estándar como en la versión K-means++, la visualización de los datos juega un papel crucial para que el programador pueda determinar el número adecuado de centroides (K). Una elección incorrecta de este valor puede afectar significativamente la calidad de la clasificación, incluso en la versión mejorada del algoritmo, K-means++.

Un ejemplo claro de esta situación se presenta cuando se selecciona un K=3. Aunque el algoritmo intenta realizar una clasificación óptima con esta cantidad de centroides, el resultado puede no ser el más adecuado para los datos disponibles. En este caso, un análisis visual previo podría indicar la necesidad de aumentar el número de centroides para reflejar de manera más precisa la distribución y estructura de los datos. Esto resalta la importancia de combinar el análisis visual con la lógica del algoritmo para maximizar la eficiencia de la clasificación y evitar posibles conflictos derivados de una mala parametrización.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Ejemplo de una mala inicialización de centroides. Fuente: Elaboración propia.

Para este caso lo adecuado sería aumentar la cantidad de K a 8 como en el caso óptimo que lo utilizamos en la parte del desarrollo, esto se infiere desde el análisis de la forma en que se encuentran dispersos los datos.

Logotipo

Descripción generada automáticamenteImagen que contiene señal, firmar, parada, tráfico

Descripción generada automáticamente

INSTITUTO POLITÉCTICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

**UNIDAD DE APRENDIZAJE:**

MACHINE LEARNING

**PRÁCTICA 6:** MÁQUINAS DE SOPORTE VECTORIAL

**INTEGRANTES:**

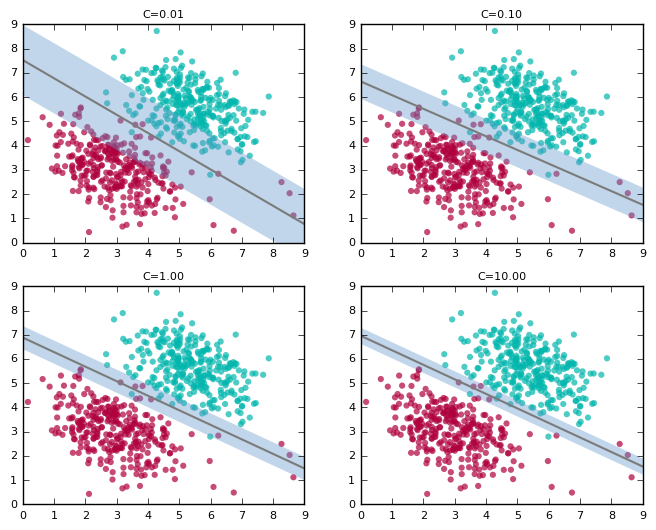
Hernández Hernández Roberto Isaac

Gonzalez Llamosas Noe Ramses

**PROFESOR:**

Ortiz Castillo Marco Antonio

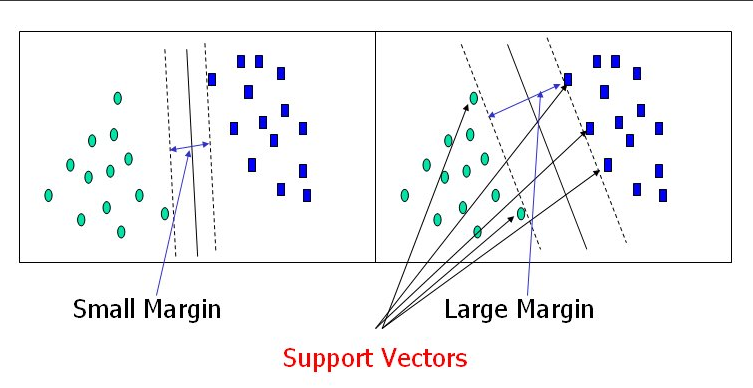
**FECHA DE ENTREGA:** 15/01/2025



# PRÁCTICA: SUPPORT VECTOR MACHINE

## INTRODUCCIÓN

La máquina de vectores de soporte (SVM) es un algoritmo de clasificación y regresión que utiliza la teoría de aprendizaje de las máquinas para maximizar la precisión de las predicciones sin ajustar excesivamente los datos. SVM utiliza una transformación no lineal opcional de los datos de entrenamiento, seguida de la búsqueda de ecuaciones de regresión en los datos transformados para separar las clases (para objetivos categóricos) o ajustar el objetivo (para los objetivos continuos). La implementación de SVM de Oracle permite que se generen modelos mediante el uso de los dos kernels disponibles: lineal o gaussiano. El kernel lineal omite la transformación no lineal de una vez, de tal forma que el modelo resultante sea, en esencia, un modelo de regresión [2].

Es un algoritmo de aprendizaje automático supervisado que se puede utilizar para problemas de clasificación o regresión. Pero generalmente se usa para clasificar. Dadas 2 o más clases de datos etiquetadas, actúa como un clasificador discriminativo, definido formalmente por un hiperplano óptimo que separa todas las clases. Los nuevos ejemplos que luego se mapean en ese mismo espacio se pueden clasificar según el lado de la brecha en que se encuentran. Los vectores de soporte son los puntos de datos más cercanos al hiperplano, los puntos de un conjunto de datos que, si se eliminan, alterarían la posición del hiperplano en división. Debido a esto, pueden considerarse los elementos críticos de un conjunto de datos, son los que nos ayudan a construir nuestra SVM [3]. En la siguiente figura se muestra la representación de vectores de soporte en SVM.

**Figura.** Representación de vectores de soporte. Fuente: [3].

La geometría nos dice que un hiperplano es un subespacio de una dimensión menos que su espacio ambiental. Por ejemplo, un hiperplano de un espacio n-dimensional es un subconjunto plano con dimensión . Por su naturaleza, separa el espacio en dos medios espacios [3].

### Funcionamiento de las SVM

Los conceptos básicos de Support Vector Machines y su funcionamiento se comprenden mejor con un ejemplo sencillo. Imaginemos que tenemos dos etiquetas: *rojos* y *azules*, y nuestros datos tiene dos características: *x* e *y*. Queremos un clasificador que, dado un par de coordenadas *(x, y)*, dé como resultado si es *rojo* o *azul*. Trazamos nuestros datos de entrenamiento ya etiquetados en un avión:

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

**Figura.** Datos etiquetados. Fuente: [3].

Gráfico, Gráfico de dispersión

Descripción generada automáticamenteUna máquina de vectores de soporte toma estos puntos de datos y genera el hiperplano (que en dos dimensiones es simplemente una línea) que separa mejor las etiquetas. Esta línea es el **límite de decisión**: todo lo que caiga a un lado lo clasificaremos como *azul* y todo lo que caiga al otro lado como *rojo*.

**Figura.** Hiperplano. Fuente: [3].

## DESARROLLO

* Para comprender el funcionamiento de este algoritmo, tomar 3 características del archivo CSV del Titanic (Edad, Clase, SipSp) y aplicando MSV realizar la clasificación con un entrenamiento equilibrado (30 vivos y 30 muertos).
* Retomar los patrones ruidosos (20 por clase) (2 clases), de ruido. Aplicar SVD, entrenar el modelo MSV y considerando 4 patrones nuevos (2 para cada clase) realizar la clasificación.

#### Códigos

**ALGORITMO SVM CON PROBLEMÁTICA DEL TITANIC**

**import** pandas **as** pd  
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**from** mpl\_toolkits.mplot3d **import** Axes3D  
  
**def** entrenamiento\_MSV(X, Y):  
    numero\_muestras, numero\_caracteristicas = X.shape  
  
    *# Inicialización parámetros externos*  
    epocas = 1000  
    lr = 0.01  
    lamda = 1 / epocas  
  
    *# Inicialización de parámetros internos*  
    w = np.zeros(numero\_caracteristicas) + 0.1  
    b = 0.1  
  
    **for** epoca **in** range(epocas):  
        **for** i, x **in** enumerate(X):  
            condicion\_margen = Y[i] \* (np.dot(x, w) + b) >= 1  
            **if** condicion\_margen:  
                w = w - lr \* (2 \* lamda \* w)  
            **else**:  
                w = w - lr \* (2 \* lamda \* w - np.dot(Y[i], x))  
                b = b - lr \* lamda \* Y[i]  
  
    tolerancia = 0.9  
    vectores\_soporte\_indices = [  
        i **for** i, x **in** enumerate(X)  
        **if** abs(Y[i] \* (np.dot(x, w) + b) - 1) <= tolerancia  
    ]  
    vectores\_soporte = X[vectores\_soporte\_indices]  
  
    **return** w, b, vectores\_soporte, vectores\_soporte\_indices  
  
**def** prediccion\_MSV(X\_test, w, b):  
    **return** np.sign(np.dot(X\_test, w) + b)  
  
*# Cargar los datos*  
ruta\_archivo = **'./train.csv'**  
datos = pd.read\_csv(ruta\_archivo)  
  
Y = datos[**'Survived'**].tolist()  
vivos = [y **for** y **in** Y **if** y == 1][:30]  
muertos = [y **for** y **in** Y **if** y == 0][:30]  
Y\_fixed = np.array(vivos + muertos)  
  
indices\_vivos = [i **for** i, y **in** enumerate(Y) **if** y == 1][:30]  
indices\_muertos = [i **for** i, y **in** enumerate(Y) **if** y == 0][:30]  
indices\_filtrados = indices\_vivos + indices\_muertos  
  
SibSp = datos[**'SibSp'**].tolist()  
Age = datos[**'Age'**].tolist()  
Pclass = datos[**'Pclass'**].tolist()  
  
moda\_sibsp = max(set(SibSp), key=SibSp.count)  
SibSp = [moda\_sibsp **if** pd.isna(s) **else** s **for** s **in** SibSp]  
SibSp = [moda\_sibsp **if** s < 0 **or** **not** isinstance(s, int) **else** s **for** s **in** SibSp]  
SibSp = [int(s) **for** s **in** SibSp]  
  
Age = [np.nan **if** pd.isna(a) **else** a **for** a **in** Age]  
promedio\_edad = np.nanmean(Age)  
Age = [promedio\_edad **if** np.isnan(a) **else** a **for** a **in** Age]  
  
moda\_clase = max(set(Pclass), key=Pclass.count)  
Pclass = [moda\_clase **if** pd.isna(c) **else** c **for** c **in** Pclass]  
Pclass = [int(c) **for** c **in** Pclass]  
  
X = np.array(list(zip(Pclass, Age, SibSp)))  
X\_fixed = X[indices\_filtrados]  
  
w, b, vectores\_soporte, vectores\_soporte\_indices = entrenamiento\_MSV(X\_fixed, Y\_fixed)  
  
*# Solicitar un nuevo punto al usuario*  
nuevo\_punto = []  
nuevo\_punto.append(int(input(**"Ingrese la clase de pasajero (Pclass, 1-3): "**)))  
nuevo\_punto.append(float(input(**"Ingrese la edad del pasajero: "**)))  
nuevo\_punto.append(int(input(**"Ingrese el número de hermanos/esposos (SibSp): "**)))  
  
nuevo\_punto = np.array(nuevo\_punto)  
clase\_predicha = prediccion\_MSV(nuevo\_punto, w, b)  
  
*# Mostrar en terminal*  
resultado = **"Sobrevivió"** **if** clase\_predicha == 1 **else** **"No sobrevivió"**  
print(**f"El nuevo punto pertenece a la clase: {resultado}"**)  
  
*# Gráfica tridimensional*  
fig = plt.figure(figsize=(10, 8))  
ax = fig.add\_subplot(111, projection=**'3d'**)  
  
*# Graficar los datos*  
ax.scatter(X\_fixed[:, 0], X\_fixed[:, 1], X\_fixed[:, 2], c=Y\_fixed, cmap=plt.cm.Paired, label=**'Datos'**)  
  
*# Graficar vectores de soporte*  
ax.scatter(X\_fixed[vectores\_soporte\_indices, 0],  
           X\_fixed[vectores\_soporte\_indices, 1],  
           X\_fixed[vectores\_soporte\_indices, 2],  
           s=150, facecolors=**'yellow'**, edgecolors=**'red'**, linewidths=2, label=**'Vectores de Soporte'**)  
  
*# Graficar el nuevo punto*  
ax.scatter(nuevo\_punto[0], nuevo\_punto[1], nuevo\_punto[2],  
           color=**'green'**, marker=**'o'**, s=150, label=**'Nuevo punto'**)  
  
*# Generar un rango para los planos*  
x = np.linspace(X\_fixed[:, 0].min(), X\_fixed[:, 0].max(), 10)  
y = np.linspace(X\_fixed[:, 1].min(), X\_fixed[:, 1].max(), 10)  
x, y = np.meshgrid(x, y)  
  
*# Hiperplano: z = -(w[0]\*x + w[1]\*y + b) / w[2]*  
z = -(w[0] \* x + w[1] \* y + b) / w[2]  
  
*# Márgenes: z = -(w[0]\*x + w[1]\*y + b ± 1) / w[2]*  
z\_margen\_superior = -(w[0] \* x + w[1] \* y + (b + 1)) / w[2]  
z\_margen\_inferior = -(w[0] \* x + w[1] \* y + (b - 1)) / w[2]  
  
*# Graficar el hiperplano y los márgenes*  
ax.plot\_surface(x, y, z, alpha=0.3, color=**'blue'**, label=**'Hiperplano'**)  
ax.plot\_surface(x, y, z\_margen\_superior, alpha=0.2, color=**'green'**, label=**'Margen Superior'**)  
ax.plot\_surface(x, y, z\_margen\_inferior, alpha=0.2, color=**'red'**, label=**'Margen Inferior'**)  
  
ax.set\_xlabel(**'Pclass'**)  
ax.set\_ylabel(**'Age'**)  
ax.set\_zlabel(**'SibSp'**)  
plt.title(**'Máquinas de Soporte Vectorial (MSV)'**)  
plt.legend()  
plt.show()

**ALGORITMO SVM PÁTRONES DE IMÁGENES**

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
  
*# Entrenamiento SVM*  
**def** entrenamiento\_MSV(X, Y):  
    numero\_muestras, numero\_caracteristicas = X.shape  
  
    *# Inicialización parámetros externos*  
    epocas = 1000  
    lr = 0.0001  
    lamda = 1 / epocas  
  
    *# Inicialización de parámetros internos*  
    w = np.zeros(numero\_caracteristicas) + 0.1  
    b = 0.1  
  
    **for** epoca **in** range(epocas):  
        **for** i, x **in** enumerate(X):  
            condicion\_margen = Y[i] \* (np.dot(x, w) + b) >= 1  
            **if** condicion\_margen:  
                *# Actualización mínima para alejar w del margen*  
                w = w - lr \* (2 \* lamda \* w)  
            **else**:  
                w = w - lr \* (2 \* lamda \* w - np.dot(Y[i], x))  
                b = b - lr \* lamda \* Y[i]  
  
    *# Identificar vectores de soporte considerando ambos márgenes*  
    tolerancia = .00001  
    vectores\_soporte\_indices = [  
        i **for** i, x **in** enumerate(X)  
        **if** abs(Y[i] \* (np.dot(x, w) + b) - 1) <= tolerancia  
    ]  
    vectores\_soporte = X[vectores\_soporte\_indices]  
  
    **return** w, b, vectores\_soporte, vectores\_soporte\_indices  
  
*# Predicción SVM*  
**def** prediccion\_MSV(X\_test, w, b):  
    **return** np.sign(np.dot(X\_test, w) + b)  
  
*# Función para visualizar patrones*  
**def** visualize\_patterns(patterns, title):  
    num\_patterns = len(patterns)  
    fig, axs = plt.subplots(1, num\_patterns, figsize=(12, 6))  
    **if** num\_patterns == 1:  
        axs = [axs]  
    **for** i **in** range(num\_patterns):  
        axs[i].imshow(patterns[i], cmap=**'gray'**)  
        axs[i].axis(**'off'**)  
    plt.suptitle(title)  
    plt.show()  
  
*# Definición manual de los patrones base*  
base\_T = np.zeros((10, 10))  
base\_T[1, 2:8] = 1  
base\_T[2:9, 5] = 1  
  
base\_J = np.zeros((10, 10))  
base\_J[1:9, 5] = 1  
base\_J[8, 3:6] = 1  
base\_J[7:9, 3] = 1  
  
*# Generación manual de variaciones de los patrones*  
patterns\_T = [base\_T + np.random.uniform(-0.05, 0.05, base\_T.shape) **for** \_ **in** range(20)]  
patterns\_J = [base\_J + np.random.uniform(-0.05, 0.05, base\_J.shape) **for** \_ **in** range(20)]  
  
*# Generación de patrones ruidosos adicionales*  
noisy\_patterns\_T = [  
    base\_T + np.random.uniform(-0.05, 0.05, base\_T.shape) **for** \_ **in** range(3)  
]  
noisy\_patterns\_J = [  
    base\_J + np.random.uniform(-0.05, 0.05, base\_J.shape) **for** \_ **in** range(3)  
]  
  
*# Aplicar SVD a todos los patrones*  
dimensiones = 5  
X = []  
  
**for** pattern **in** patterns\_T + patterns\_J:  
    U, S, Vt = np.linalg.svd(pattern)  
    X.append((U[:, :dimensiones] @ np.diag(S[:dimensiones])).flatten())  
  
X = np.array(X)  
Y = np.array([-1] \* 20 + [1] \* 20)  *# Clases: -1 para T, 1 para J*  
  
*# Representaciones de los patrones ruidosos*  
U\_T, S\_T, \_ = np.linalg.svd(noisy\_patterns\_T[0])  
test\_T = (U\_T[:, :dimensiones] @ np.diag(S\_T[:dimensiones])).flatten()  
  
U\_J, S\_J, \_ = np.linalg.svd(noisy\_patterns\_J[0])  
test\_J = (U\_J[:, :dimensiones] @ np.diag(S\_J[:dimensiones])).flatten()  
  
*# Entrenamiento con SVM*  
w, b, vectores\_soporte, vectores\_soporte\_indices = entrenamiento\_MSV(X, Y)  
  
*# Predicción*  
clase\_T = prediccion\_MSV(test\_T, w, b)  
clase\_J = prediccion\_MSV(test\_J, w, b)  
  
print(**"Clase para el patrón T (ruido):"**, **"T"** **if** clase\_T == -1 **else** **"J"**)  
print(**"Clase para el patrón J (ruido):"**, **"T"** **if** clase\_J == -1 **else** **"J"**)  
  
*# Visualización*  
visualize\_patterns([base\_T], **"Patrón Base T"**)  
visualize\_patterns([base\_J], **"Patrón Base J"**)  
visualize\_patterns(patterns\_T, **"Variaciones de Patrón T"**)  
visualize\_patterns(patterns\_J, **"Variaciones de Patrón J"**)  
visualize\_patterns([noisy\_patterns\_T[0]], **f"Patrón Ruido T - Clasificado como {'T' if clase\_T == -1 else 'J'}"**)  
visualize\_patterns([noisy\_patterns\_J[0]], **f"Patrón Ruido J - Clasificado como {'T' if clase\_J == -1 else 'J'}"**)  
visualize\_patterns([noisy\_patterns\_T[1]], **f"Patrón Ruido T - Clasificado como {'T' if clase\_T == -1 else 'J'}"**)  
visualize\_patterns([noisy\_patterns\_J[1]], **f"Patrón Ruido J - Clasificado como {'T' if clase\_J == -1 else 'J'}"**)  
visualize\_patterns([noisy\_patterns\_T[2]], **f"Patrón Ruido T - Clasificado como {'T' if clase\_T == -1 else 'J'}"**)  
visualize\_patterns([noisy\_patterns\_J[2]], **f"Patrón Ruido J - Clasificado como {'T' if clase\_J == -1 else 'J'}"**)

#### Funcionamiento

**ALGORITMO SVM CON PROBLEMÁTICA DEL TITANIC**

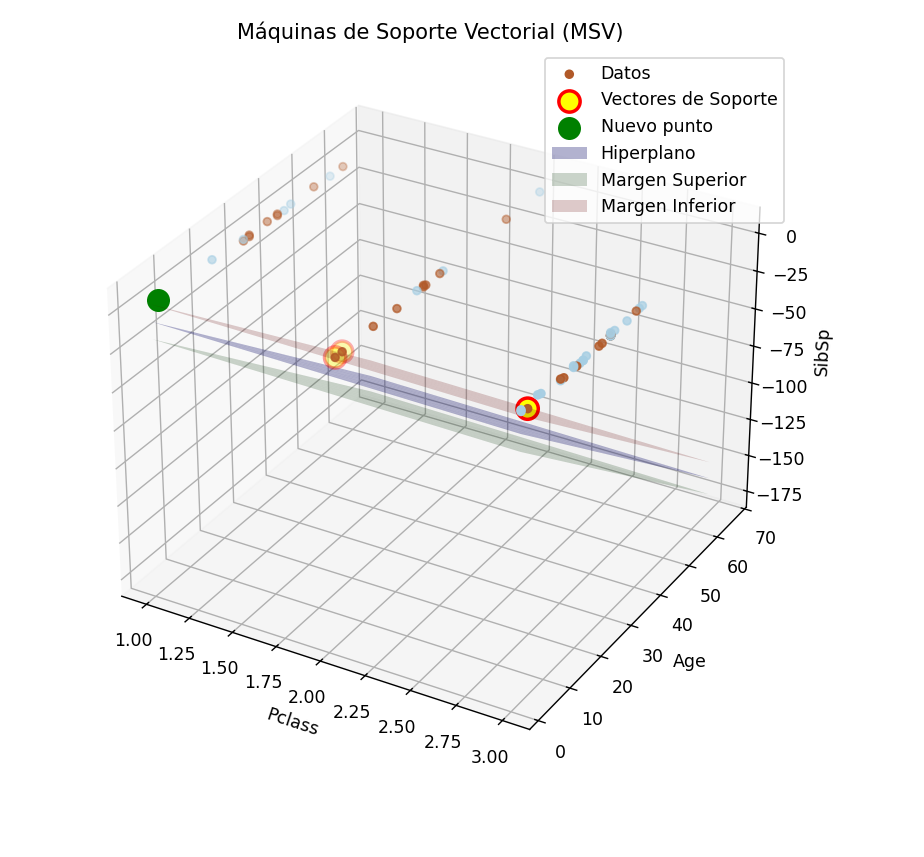
De manera similar al caso del Titanic con la implementación del algoritmo K-means++, se solicita al usuario ingresar un nuevo dato para clasificar, y el sistema imprime en la terminal si la persona sobrevivió o falleció, basándose en el entrenamiento previo.

Para el caso de las Máquinas de Vectores de Soporte (SVM), se llevó a cabo un entrenamiento equilibrado utilizando un conjunto de datos compuesto por 30 personas fallecidas y 30 sobrevivientes. Este equilibrio fue fundamental para garantizar una clasificación adecuada, ya que este algoritmo se basa en la separación de valores binarios. Además, fue necesario emplear una gráfica tridimensional para visualizar y analizar mejor los resultados de la clasificación.

Texto

Descripción generada automáticamente

**Figura.** Resultado en terminal de una persona que sobrevivió. Fuente: Elaboración propia.



**Figura.** Resultado en gráfica 3D de una persona que sobrevivió. Fuente: Elaboración propia.

**ALGORITMO SVM PÁTRONES DE IMÁGENES**

En este ejercicio, trabajamos con dos patrones de datos que forman las letras "T" y "J". Generamos diversas variaciones de estos patrones con el propósito de entrenar el modelo y así clasificar correctamente dos nuevos patrones para cada letra. Para esta tarea, empleamos Máquinas de Vectores de Soporte (SVM), lo que nos permitió determinar a qué clase pertenece cada uno de los nuevos patrones propuestos, basándonos en las características aprendidas durante el entrenamiento.

En las siguientes figuras se muestra la manera la clasificación realizada por el algoritmo con los nuevos patrones implementados.

Imagen que contiene Logotipo

Descripción generada automáticamenteTexto, Logotipo

Descripción generada automáticamente

Código QR

Descripción generada automáticamenteCódigo QR

Descripción generada automáticamente

## CONCLUSIÓN

En el caso del algoritmo SVM, es fundamental que el programador analice cuidadosamente los datos y las características del conjunto para seleccionar los parámetros adecuados, como el kernel y los valores de regularización. Una elección incorrecta de estos parámetros puede comprometer significativamente la capacidad del algoritmo para clasificar los datos de manera precisa.

Por ejemplo, si los datos no son linealmente separables y se selecciona un kernel lineal en lugar de uno no lineal, como el RBF o el polinomial, el modelo podría fallar en capturar las relaciones complejas presentes en los datos. Este escenario destaca la importancia de una exploración visual y un análisis detallado de los datos antes de entrenar el modelo.

Al igual que con otros algoritmos, el éxito de SVM depende tanto de las decisiones tomadas durante la configuración como de la calidad del conjunto de datos, lo que subraya la relevancia de combinar el conocimiento técnico con una comprensión profunda de los datos para garantizar resultados óptimos.