Universidad Autónoma de Querétaro

Facultad de Ingeniería
Maestría en Inteligencia Artificial
Machine Learning
Sheila Leyva López
Cecilia Gabriela Rodríguez Flores

Algoritmos de agrupamiento de datos: K-Means++ vs BIRCH.

09 de diciembre del 2022

Objetivo

Implementar y comparar dos algoritmos de agrupamiento diferentes, K-means++ y BIRCH, utilizando como datos a clasificar la base de enfermedad cardíaca.

Introducción

Los problemas de agrupamiento probablemente son uno de los temas más estudiados dentro del aprendizaje máquina y la minería de datos. El agrupamiento de datos tiene múltiples dominios de aplicación como en datos multimedia, de texto, redes sociales, datos biológicos y más [1,2].

En este trabajo se aplica dicha técnica de agrupamiento a la base de datos *Indicadores personales clave de enfermedad cardíaca*. El agrupamiento de la ya mencionada base de datos se realizará con dos algoritmos distintos, K-means ++ y el algoritmo BIRCH. En el presente trabajo se desarrollan las funciones, en lenguaje de Python, necesarias para el agrupamiento de cualquier base de datos, sin embargo, toma como referencia la base de datos de enfermedad cardíaca.

Marco Teórico

El agrupamiento o clustering (en inglés) es una herramienta, específica del aprendizaje no supervisado, encargada de agrupar un conjunto de objetos sin etiqueta en clases o clústeres que son altamente similares entre sí, es decir, comparten propiedades y/o características, mientras que son escasamente diferente al resto de objetos, dado que no tienen propiedades y/o características similares [3][4]. Básicamente, se desconoce si los datos ocultan patrones, así que el algoritmo se encarga de encontrarlos si es que los hay[5].

El agrupamiento tiene una gran cantidad de aplicaciones en distintos ámbitos de la vida, desde segmentación de mercado hasta en imágenes médicas, así como en análisis de redes sociales, agrupación de resultados de búsqueda, segmentación de imágenes, motores de búsqueda y más [3].

Existen distintos tipos de datos lo cual conlleva a que existan diferentes tipos de algoritmos, por ejemplo, aquellos basados en la densidad de los datos, en la distribución o los basados en centroides y jerarquías[5].

K-means ++

El algoritmo de k-means es la técnica más popular, la cual se basa en centroides. Sin embargo, presenta una desventaja, depende en gran medida de la inicialización de los centroides[6]. Es decir, si el centroide se inicializa en un punto alejado podría terminar sin puntos asociados a él o en caso contrario podría terminar con más de un clúster asociado, esto y otros factores pueden ocasionar un clustering deficiente.

Debido a la desventaja que presenta la técnica Kmeans, surge K-means++, que intenta solucionar la situación asegurando que los centroides se inicialicen de mejor manera, de tal forma que se mejore la calidad del clustering; esencialmente esta es la única diferencia entre k-means y K-means ++, el resto del método es idéntico[6].

Principalmente, la inicialización consiste en seleccionar de forma aleatoria el primer centroide los puntos de datos, para posteriormente, calcular la distancia de cada punto de datos al centroide más cercano elegido en el paso uno. Después, se selecciona el siguiente centroide de los puntos de datos, de tal manera que el punto que se desea elegir como centroide tiene la distancia máxima medida desde el centroide más cercano y tiene la mayor probabilidad de ser elegido, y así, sucesivamente se repiten los dos pasos anteriores hasta que se cumplan para todos los k-ésimos centroides[7].

$$J = \sum_{k \in K} \sum_{i} z_k^i |x_i - \mu_k|^2$$

Donde:
xi: representa al objeto.
μk: media del cluster

A continuación, se enumeran algunas limitaciones de la agrupación de esta técnica de agrupamiento.

- El resultado está muy influenciado por la entrada original, es decir, por el número de agrupaciones.
- En algunos casos, las agrupaciones muestran vistas espaciales complejas, por lo cual la esta técnica no es buena elección para estos casos.

Por otra parte, se presentan algunas de las desventajas de este algoritmo de agrupamiento:

- El algoritmo exige la especificación inferida del número de agrupaciones.
- El algoritmo no funciona con conjuntos de datos no lineales y no es capaz de tratar datos ruidosos y valores atípicos.
- No es directamente aplicable a datos categóricos
- La distancia euclidiana puede ponderar de forma desigual los factores subyacentes.
- El algoritmo no es variante de la transformación no lineal, es decir, proporciona resultados diferentes con distintas representaciones de los datos.

BIRCH

Los algoritmos de agrupamiento, como el agrupamiento de K-means, no realizan el agrupamiento de manera muy eficiente, lo cual dificulta el procesamiento de grandes conjuntos de datos debido a la cantidad limitada de recursos (como memoria o una CPU más lenta). Por lo tanto, los algoritmos de agrupamiento regulares no escalan bien en términos de tiempo de ejecución y calidad a medida que aumenta el tamaño del conjunto de datos. Aquí es donde entra en juego la agrupación en clústeres BIRCH.

Agrupación y reducción iterativa equilibrada mediante jerarquías (BIRCH, por sus siglas en inglés) es un algoritmo de agrupamiento que puede agrupar grandes conjuntos de datos generando primero un resumen pequeño y compacto del gran conjunto de datos que conserva la mayor cantidad de información posible. Este resumen más pequeño posteriormente se agrupa en clústeres en lugar de agrupar el conjunto de datos más grande.

BIRCH se utiliza a menudo para complementar otros algoritmos de agrupación mediante la creación de un resumen del conjunto de datos que ahora puede utilizar el otro algoritmo de agrupación. Sin embargo, este algoritmo tiene un gran inconveniente ya que solo puede procesar atributos métricos. Un atributo métrico es cualquier atributo cuyos valores pueden representarse en el espacio euclidiano, es decir, no deben estar presentes atributos categóricos.

Antes de implementar BIRCH, debemos comprender dos términos importantes: Función de agrupamiento (CF) y Función de agrupamiento de árboles (TCF):

BIRCH resume grandes conjuntos de datos en regiones más pequeñas y densas denominadas entradas de función de agrupamiento (CF). Formalmente, una entrada de característica de agrupación se define como un triple ordenado (N, LS, SS) donde 'N' es el número de puntos de datos en el grupo, 'LS' es la suma lineal de los puntos de datos y 'SS' es la suma al cuadrado de los puntos de datos en el grupo. Es posible que una entrada CF esté compuesta por otras entradas CF.

El árbol CF es la representación compacta real de la que hemos estado hablando hasta ahora. Un árbol CF es un árbol en el que cada nodo hoja contiene un subclúster. Cada entrada en un árbol

CF contiene un puntero a un nodo secundario y una entrada CF formada por la suma de las entradas CF en los nodos secundarios. Hay un número máximo de entradas en cada nodo hoja, el cual se denomina umbral.

A continuación, se presentan los parámetros necesarios para realizar la implementación del algoritmo BIRCH:

- Umbral: el umbral es el número máximo de puntos de datos que puede contener un subclúster en el nodo hoja del árbol CF.
- Branching_factor : este parámetro especifica el número máximo de subclústeres de CF en cada nodo (nodo interno).
- n_clusters : el número de clústeres que se devolverán después de que se complete todo el algoritmo BIRCH, es decir, el número de clústeres después del último paso de agrupación. Si se establece en Ninguno, no se realiza el paso de agrupación final y se devuelven las agrupaciones intermedias.

Ventajas

- Ahorra memoria, todas las muestras están en el disco, CF Tree solo almacena los nodos CF y los punteros correspondientes.
- La velocidad de agrupación es rápida, y solo se necesita un escaneo del conjunto de entrenamiento para construir el CF Tree, y la adición, eliminación y modificación del Árbol CF son muy rápidas.
- Se pueden identificar puntos de ruido y se puede realizar un preprocesamiento de clasificación preliminar en el conjunto de datos.

Desventajas

- Dado que CF Tree tiene un límite en la cantidad de CF por nodo, el resultado de la agrupación puede ser diferente de la distribución de categorías real.
- El efecto de agrupación de datos no es bueno en características de alta dimensión. En este momento, puede elegir Mini Batch K-Means.

- Si el grupo de distribución del conjunto de datos no es similar a una hiperesfera o no es convexo, el efecto de agrupación no es bueno.

Etapas en la implementación de modelos

Es importante definir conceptos básicos del aprendizaje máquina que permitan entender el contexto en el que se trabaja, como son las etapas fundamentales para la implementación de técnicas de aprendizaje máquina: Preparación de datos, la cual es una etapa fundamental en cualquier proyecto, debido a que por medio de ella se pueden inicializar correctamente los datos para su posterior procesamiento y análisis; Analizar, en esta sección se utilizaran (qué datos se van a utilizar) o dividirá los datos ya preparados para el siguiente paso, para este trabajo se optó por crear subconjuntos de datos, en los cuales el 80% de los datos se considera para el entrenamiento, mientras que un 20% corresponde a pruebas del modelo de agrupamiento.

Métrica

La manera más simple de evaluar un modelo con características nominales y discretas es por medio de diferentes métricas, como es la métrica de exactitud, la cual es la relación entre los simples correctamente clasificados y el número total de simples en el conjunto de datos de evaluación. Esta métrica es conocida por ser engañosa en el caso de diferentes proporciones de clases, ya que asignar simplemente todos los simples a la clase predominante es una forma fácil de lograr una alta precisión [3].

$$ACC = \frac{\#\ correctly\ classified\ samples}{\#\ all\ samples} = \frac{TP+TN}{TP+FP+TN+FN} \quad (1)$$

Materiales y métodos

Herramientas utilizadas

- Google Collaboratory
- Conjunto de datos: *Indicadores personales clave de enfermedad cardíaca*

- AMD Ryzen 9 5900HS with Radeon Graphics 3.30 GHz
- RAM 16.0 GB
- 64-bit operating system, x64-based processor

Conjunto de datos

En este trabajo, se utilizará el conjunto de datos: *Indicadores personales clave de enfermedad cardíaca*, datos provenientes de 400,000 adultos, obtenidos durante la encuesta anual 2020 de los Centros para el Control y prevención de Enfermedades (CDC, por sus siglas en inglés) pertenecientes al departamento de salud y servicios humanos en los Estados Unidos. Originalmente el conjunto de datos contenía alrededor de 300 atributos, sin embargo, se redujo a solo 18 variables, los cuales son los que se encuentran disponibles públicamente en la plataforma Kaggle [6].

Casi la mitad de los estadounidenses (47%), incluyendo afroamericanos, indios americanos, nativos de Alaska y blancos; tienen al menos de 1 a 3 factores de riesgo de padecer alguna enfermedad cardíaca. A continuación, se agrega una breve descripción de los atributos incluidos en este conjunto de datos:

- HeartDisease: (atributo de decisión): personas encuestadas que informaron alguna vez haber padecido alguna enfermedad coronaria (CHD, por sus siglas en inglés) o infarto al miocardio(IM, por sus siglas en inglés).
- BMI: Índice de Masa Corporal.
- Smoking: personas encuestadas que han fumado al menos 100 cigarros en su vida entera.
- AlcoholDrinking: corresponde a hombres adultos que beben más de 14 tragos por semana y mujeres adultas que beben más de 7 tragos por semana.
- Stroke: responde a la pregunta: ¿alguna vez le dijeron o usted tuvo un derrame cerebral?
- PhysicalHealth: incluyendo enfermedades y lesiones físicas, responde a la pregunta: ¿durante cuántos días en los últimos 30 días su salud física no fue buena? (de 0 a 30 días)
- MentalHealth: ¿durante cuántos días en los últimos 30 días su salud mental no fue buena? (de 0 a 30 días).

- DiffWalking: responde a ¿tiene serias dificultades para caminar o subir escaleras?
- Sex: hombre o mujer.
- AgeCategory: 14 rangos de edad.
- Race: valor de raza / etnicidad imputada.
- Diabetic: responde a ¿alguna vez ha sido diagnosticada con diabetes?
- PhysiclActivity: adultos que informaron haber realizado actividad física o ejercicio en los últimos 30 días, no incluyendo su trabajo habitual.
- GenHealth: responde a ¿cómo calificarías tu salud en general?
- SleepTime: responde a un promedio de horas que duerme, en un periodo de 24 horas, la persona encuestada.
- Asthma: responde a ¿alguna vez ha sido diagnosticado con asma?

Métodos

En esta sección se presentan las funciones de los algoritmos de agrupamiento propuestos, en donde se incluye de igual forma las funciones para la preparación de los datos y la función para la división de entrenamiento y prueba.

Primeramente, para la realización de esta tarea, fue necesario importar las librerías de Pandas, numpy, etc., a fin de poder hacer uso de la base de datos propuesta.

Posteriormente, se optó por crear diversas funciones a fin de distribuir las tareas de una manera más eficiente como se muestra a continuación:

- Función valores_faltantes: obtiene el porcentaje total de los valores faltantes, así como el porcentaje de valores faltantes para ca uno de los atributos que comprende la base de datos en cuestión. Esta función requiere como datos de entrada únicamente la base de datos.

```python
def valores_faltantes(dataset):
    missing_values_count = dataset.isnull().sum()
    total_missing = missing_values_count.sum()
    #Porcentaje de datos faltantes
    total_missing_porcent = total_missing/(np.product(dataset.shape))*100
    print('Porcentaje total de valores faltantes:',total_missing_porcent,'%')
    print('')
    print('Porcentaje de valores faltantes de cada atributo:')
    for col in dataset.columns:
        VP_missing = np.mean(dataset[col].isnull())
        print('{} - {}%'.format(col,round(VP_missing*100)))
```

Ilustración 1. Función para calcular valores faltantes.

- Función norm_min_max: realiza la normalización de la base de datos. Esta función requiere como datos de entrada la base de datos en cuestión.

```python
def norm_min_max(datos):
    lim_sup = []
    lim_inf = []
    rangoDatos =[]
    maxNorm = 1
    minNorm = 0
    rango = maxNorm - minNorm
    for i in range (0,datos.columns.size):
        lim_sup.append(datos.iloc[:,i].max())
        lim_inf.append(datos.iloc[:,i].min())
        rangoDatos.append(lim_sup[i] - lim_inf[i])
    nombres = datos.columns.values.tolist()
    datosNorm = pd.DataFrame(columns = nombres)

    for j in range(len(datos.columns)):
        varNorm = []
        var = datos.iloc[:,j]
        for i in range(len(datos)):
            D = var[i] - lim_inf[j]
            DPct = D/rangoDatos[j]
            dNorm = rango*DPct
            varNorm.append(minNorm+dNorm)
        datosNorm.iloc[:,j] = varNorm
    datos = datosNorm
    return datos
```

Ilustración 2. Función para normalizar los datos.

- Función matriz_cov: se encarga de obtener la matriz de covarianza de los elementos de la base de datos en cuestión.

```python
def matriz_cov(data):
    atributos = data.columns
    n = len(atributos)
    m = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            X = data[atributos[i]]
            Y = data[atributos[j]]
            m[i][j] = (((X-X.mean())*(Y-Y.mean())).sum())/(len(X)-1)
    return m
```

Ilustración 3. Función para calcular la matriz de covarianza.

- Función PCA: brinda los porcentajes de cada uno de los atributos de acuerdo con su relevancia dentro de la base de datos, a fin de discernir los atributos con mayor peso.

```
def PCA(datos,col_decision):
    datos1 = datos.drop([col_decision],axis=1) #Eliminando el atributo de decisión
    #Ajustar los datos restando la media a cada atributo
    datos_A = pd.DataFrame(columns=datos1.columns,index=range(len(datos1)))
    for i in datos_A.columns:
        datos_A[i] = datos1[i] - datos1[i].mean()
    #datos_A
    matrix = matriz_cov(datos_A)
    #sns.heatmap(matrix)
    L,V =  np.linalg.eig(matrix)
    #Obtener el porcentaje de covarianza de cada uno de los atributos
    total = L.sum()
    p = (L/total)*100
    pca =[]
    columnas1 = datos_A.columns.values
    for index, row in enumerate(p):
        print(columnas1[index] + ':',row)
```

Ilustración 4. Función de PCA.

- Función method_8020: realiza la separación del conjunto de datos de acuerdo con el porcentaje de 80% para el conjunto de entrenamiento y 20% para el conjunto de prueba. Esta función requiere como entradas los datos y las etiquetas.

```
def method_8020(x,y):
    train_x = x[0 : int(len(x)*0.8)]
    train_y = y[0 : int(len(y)*0.8)]
    test_x = x[int(len(x)*0.8) : ]
    test_y = y[int(len(y)*0.8) : ]
    return train_x, train_y, test_x, test_y
```

Ilustración 5. Función para dividir datos.

- Función fit: realiza el entrenamiento del algoritmo de k-means ++. Esta función requiere como entrada la base de datos en cuestión.

```
def fit(self, dataset):
    self.x_data = dataset.iloc[:, 0]
    self.y_data = dataset.iloc[:, 1]
    self.X = dataset.iloc[:, [0, 1]]  # not use feature labels
    self.m = self.X.shape[0]  # number of training examples
    self.n = self.X.shape[1]  # number of features.
    self.initial_centroids = []

    first_cen = self.get_random_centroid()
    initial_centroids = self.select_the_others_centroid(first_cen)

    self.plot_initial_centroids(initial_centroids)
    c = self.clustering(initial_centroids)
    return c
```

Ilustración 6. Función para ajustar el modelo.

- Función get_random_centroid: realiza la obtención aleatoria del primer centroide.

```
def get_random_centroid(self):
    return np.random.randint(len(self.x_data))
```

Ilustración 7. Función para obtener centroide.

- Función select_the_others_centroid: se encarga de obtener los demás centroides de acuerdo con el primer centroide obtenido. Esta función requiere como entrada el centroide inicial.

```
def select_the_others_centroid(self, first_cen):
    self.initial_centroids.append((self.x_data[first_cen], self.y_data[first_cen]))
    for i in range(self.n_cluster - 1):
        dis_max = 0
        index_max = 0
        for j in range(len(self.x_data)):
            if j != first_cen and (self.x_data[j], self.y_data[j]) not in self.initial_centroids:
                dis_temp = 0
                for k in self.initial_centroids:
                    pt1 = k
                    pt2 = (self.x_data[j], self.y_data[j])
                    dis_temp += self.euclidean_distance(pt1, pt2)
                if dis_temp > dis_max:
                    dis_max = dis_temp
                    index_max = j
        self.initial_centroids.append((self.x_data[index_max], self.y_data[index_max]))
    #print(self.initial_centroids)
    return np.array(self.initial_centroids)
```

Ilustración 8. Función para obtener centroides.

- Función clustering: realiza la agrupación de los datos de acuerdo con las posiciones de los centroides en cuestión, la cual se va a detener cuando las posiciones de los centroides no varíen. Esta función requiere como entrada los centroides.

```
def clustering(self, centroids):

    old_centroids = np.zeros(centroids.shape)
    stopping_criteria = 0.0001
    self.iterating_count = 0
    self.objective_func_values = []

    while self.euclidean_distance(old_centroids, centroids) >= stopping_criteria:
        clusters = np.zeros(len(self.X))
        C = []
        # Assigning each value to its closest cluster
        for i in range(self.m):
            distances = []
            for j in range(len(centroids)):
                distances.append(self.euclidean_distance(self.X.iloc[i, :], centroids[j]))
            cluster = np.argmin(distances)
            clusters[i] = cluster

        # Storing the old centroid values to compare centroid moves
        old_centroids = centroids.copy()

        # Finding the new centroids
        for i in range(self.n_cluster):
            points = [self.X.iloc[j, :] for j in range(len(self.X)) if clusters[j] == i]
            centroids[i] = np.mean(points, axis=0)

        # calculate objective function value for current cluster centroids
        self.objective_func_values.append([self.iterating_count, self.objective_func_calculate(clusters, centroids)])
        self.plot_centroids(centroids, clusters)
        self.iterating_count += 1

    self.plot_objective_function_values()
    return centroids
```

Ilustración 9. Función para clasificar/agrupar los datos.

- Función predict: se encarga de realizar la etapa de prueba del algoritmo de k-means++, en donde se obtendrá la asignación de cada uno de los datos de acuerdo con la cercanía entre los agrupamientos realizados. Esta función requiere como entradas los centroides y el conjunto de datos de prueba.

```
def predict(self, centroids,X):
    self.D = X.iloc[:, [0, 1]]  # not use feature labels
    self.b = self.D.shape[0]  # number of training examples
    clusters = np.zeros(len(self.D))
    C = []
    # Assigning each value to its closest cluster
    for i in range(self.b):
        distances = []
        for j in range(len(centroids)):
            distances.append(self.euclidean_distance(self.D.iloc[i, :], centroids[j]))
        cluster = np.argmin(distances)
        clusters[i] = cluster

    # calculate objective function value for current cluster centroids
    #C = self.plot_centroids(centroids, clusters)
    return clusters
```

Ilustración 10. Función para la predicción.

- Función euclidean_distance: se encarga de obtener la distancia entre cada uno de los datos. Esta función requiere como entrada los datos.

```
def euclidean_distance(self, a, b):
    return np.sqrt(np.sum((np.array(a) - np.array(b))**2))
```

Ilustración 11. Función para calcular la distancia euclidiana.

Las siguientes funciones corresponden al algoritmo BIRCH:

- Función objective_func_calculate: se encarga de calcular la suma de las distancias de los centroides a fin de determinar las variaciones entre cada iteración. Esta función requiere como entrada los clústeres y los centroides.

```
def objective_func_calculate(self, clusters, centroids):
    """Calcular el valor de la función objetivo para los centroides actuales"""
    distances_from_centroids = []
    for i in range(self.n_cluster):
        points = np.array([self.X.iloc[j, :] for j in range(len(self.X)) if clusters[j] == i])
        for k in range(len(points)):
            distances_from_centroids.append(self.euclidean_distance(points[k, :], centroids[i]))
    return sum(distances_from_centroids)
```

Ilustración 12. Función para calcular el valor de la función objetivo para los centroides actuales.

- Función split_node: se encarga de dividir el nodo si no hay lugar para un nuevo subclúster en el nodo. Así bien, las propiedades de los subclústeres y nodos vacíos se actualizan

según la distancia más cercana entre los subclústeres y el par de subclústeres distantes. Esta función requiere como entradas threshold y branching factor.

```
def split_node(node, threshold, branching_factor):
    new_subcluster1 = CF_Subcluster()
    new_subcluster2 = CF_Subcluster()
    new_node1 = CF_Node(threshold=threshold,branching_factor=branching_factor,is_leaf=node.is_leaf,n_features=node.n_feature
    new_node2 = CF_Node(threshold=threshold,branching_factor=branching_factor,is_leaf=node.is_leaf,n_features=node.n_feature
    new_subcluster1.child = new_node1
    new_subcluster2.child = new_node2
    if node.is_leaf:
        if node.prev_leaf is not None:
            node.prev_leaf.next_leaf = new_node1
        new_node1.prev_leaf = node.prev_leaf
        new_node1.next_leaf = new_node2
        new_node2.prev_leaf = new_node1
        new_node2.next_leaf = node.next_leaf
        if node.next_leaf is not None:
            node.next_leaf.prev_leaf = new_node2
    dist = euclidean_distances(node.centroids,node.centroids,squared=True)
    #print('dist',dist)
    n_clusters = dist.shape[0]
    farthest_idx = np.unravel_index(dist.argmax(), (n_clusters, n_clusters))
    #print('farthest_idx',farthest_idx)
    node1_dist, node2_dist = dist[(farthest_idx,)]
    #print('node1_dist',node1_dist)
    #print('node2_dist',node2_dist)
    node1_closer = node1_dist < node2_dist
    for idx, subcluster in enumerate(node.subclusters):
        if node1_closer[idx]:
            new_node1.append_subcluster(subcluster)
            new_subcluster1.update(subcluster)
        else:
            new_node2.append_subcluster(subcluster)
            new_subcluster2.update(subcluster)
    return new_subcluster1, new_subcluster2
```

Ilustración 13. Función para hacer la separación de nodos.

- Función update: realiza modificaciones en el subclúster, centroid y sq_norm. Esta función requiere como entrada el subclúster.

```
def update(self, subcluster):
    self.n_samples += subcluster.n_samples
    self.linear_sum += subcluster.linear_sum
    self.squared_sum += subcluster.squared_sum
    self.centroid = self.linear_sum / self.n_samples
    self.sq_norm = np.dot(self.centroid, self.centroid)
```

Ilustración 14. Función para actualizar los grupos de datos.

- Función merge_subcluster: se encarga de comprobar si un clúster puede ser fusionado. Esa función requiere como entrada nominee_cluster y threshold.

```
def merge_subcluster(self, nominee_cluster, threshold):
    new_ss = self.squared_sum + nominee_cluster.squared_sum
    new_ls = self.linear_sum + nominee_cluster.linear_sum
    new_n = self.n_samples + nominee_cluster.n_samples
    new_centroid = (1 / new_n) * new_ls
    new_sq_norm = np.dot(new_centroid, new_centroid)
    sq_radius = new_ss / new_n - new_sq_norm
    if (sq_radius <= threshold ** 2):
        (self.n_samples,self.linear_sum,self.squared_sum,self.centroid,self.sq_norm,) = (new_n, new_ls, new_ss, new_centroi
        return True
    return False
```

Ilustración 15. Función para fusionar subclústeres.

- Función append_subcluster: se encarga de añadir el subclúster en cuestión al conjunto de subclústeres. Esta función requiere como entrada el subclúster.

```
def append_subcluster(self, subcluster):
    n_samples = len(self.subclusters)
    self.subclusters.append(subcluster)
    self.init_centroids[n_samples] = subcluster.centroid
    self.init_sq_norm[n_samples] = subcluster.sq_norm
    self.centroids = self.init_centroids[:n_samples+1,:]
    self.squared_norm = self.init_sq_norm[:n_samples+1]
```

Ilustración 16. Función para agregar nuevos subclústeres.

Ilustración 17. Proceso de adhesión de subclúster.

- Función update_split_subclusters: se encarga de eliminar un subclúster de un nodo y lo actualiza con los subclústeres divididos. Esta función requiere como entrada el subclúster, el nuevo primer subclúster y el nuevo segundo subclúster.

```
def update_split_subclusters(self, subcluster, new_subcluster1, new_subcluster2):
    ind = self.subclusters.index(subcluster)
    self.subclusters[ind] = new_subcluster1
    self.init_centroids[ind] = new_subcluster1.centroid
    self.init_sq_norm[ind] = new_subcluster1.sq_norm
    self.append_subcluster(new_subcluster2)
```

Ilustración 18. Función para actualizar la división de subclústeres.

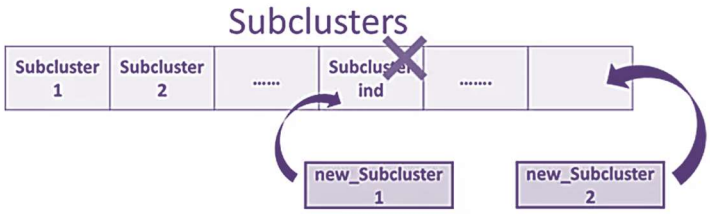


Ilustración 19. Proceso de división y actualización de subclústeres.

- Función insert_cf_subcluster: se encarga de inserta un nuevo subclúster en el nodo. Esta función requiere como entrada subclúster.

```
def insert_cf_subcluster(self, subcluster):
    if not self.subclusters:
        self.append_subcluster(subcluster)
        return False
    threshold = self.threshold
    branching_factor = self.branching_factor
    dist_matrix = np.dot(self.centroids, subcluster.centroid)
    dist_matrix *= -2.0
    dist_matrix += self.squared_norm
    closest_index = np.argmin(dist_matrix)
    closest_subcluster = self.subclusters[closest_index]
    if closest_subcluster.child is not None:
        split_child = closest_subcluster.child.insert_cf_subcluster(subcluster)
        if not split_child:
            closest_subcluster.update(subcluster)
            self.init_centroids[closest_index] = self.subclusters[closest_index].centroid
            self.init_sq_norm[closest_index] = self.subclusters[closest_index].sq_norm
            return False
        else:
            new_subcluster1, new_subcluster2 = split_node(closest_subcluster.child,threshold,branching_factor)
            self.update_split_subclusters(closest_subcluster,new_subcluster1,new_subcluster2)
            if len(self.subclusters) > self.branching_factor:
                return True
            return False
    else:
        merged = closest_subcluster.merge_subcluster(subcluster, self.threshold)
        if merged:
            self.init_centroids[closest_index] = closest_subcluster.centroid
            self.init_sq_norm[closest_index] = closest_subcluster.sq_norm
            return False
        elif len(self.subclusters) < self.branching_factor:
            self.append_subcluster(subcluster)
            return False
        else:
            self.append_subcluster(subcluster)
            return True
```

Ilustración 20. Función encargada de agregar un nuevo subclúster al nodo.

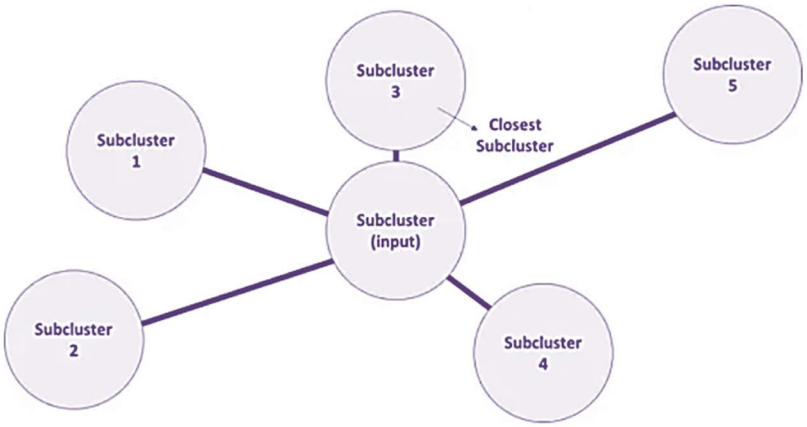


Ilustración 21. Detección del subclúster más cercano.

- Función fit: se encarga de construir un árbol CF para los datos de entrada. Esta función requiere como entrada la base de datos en cuestión.

```
def fit(self,X):
    threshold = self.threshold
    branching_factor = self.branching_factor
    X = check_array(X, accept_sparse='csr', copy=True)
    n_samples, n_features = X.shape
    ##CF
    self.root = CF_Node(threshold=threshold,branching_factor=branching_factor,is_leaf=True,n_features=n_features)
    self.dummy_leaf = CF_Node(threshold=threshold,branching_factor=branching_factor,is_leaf=True,n_features=n_features)
    self.dummy_leaf.next_leaf = self.root
    self.root.prev_leaf = self.dummy_leaf
    iter_func = iter(X)
    for sample in iter_func:
        subcluster = CF_Subcluster(linear_sum=sample)
        split = self.root.insert_cf_subcluster(subcluster)
        if split:
            new_subcluster1, new_subcluster2 = split_node(self.root, threshold, branching_factor)
            del self.root
            self.root = CF_Node(threshold=threshold,branching_factor=branching_factor,is_leaf=False,n_features=n_features,)
            self.root.append_subcluster(new_subcluster1)
            self.root.append_subcluster(new_subcluster2)
    centroids = np.concatenate([leaf.centroids for leaf in self.get_leaves()])
    self.subcluster_centers = centroids
    self.global_clustering(X)
    return self
```

Ilustración 22. Función para el entrenamiento del algoritmo.

- Función get_leaves: se encarga de recuperar las hojas del nodo CF.

```
def get_leaves(self):
    leaf_ptr = self.dummy_leaf.next_leaf
    leaves = []
    while leaf_ptr is not None:
        leaves.append(leaf_ptr)
        leaf_ptr = leaf_ptr.next_leaf
    return leaves
```

Ilustración 23. Función para construir las hojas del nodo CF.

- Función global_clustering: realiza la agrupación global de los subclústeres obtenidos tras el ajuste. Esta función requiere como entrada los datos en cuestión.

```
def global_clustering(self, X):
    clusterer = self.n_clusters
    centroids = self.subcluster_centers
    compute_labels = (X is not None) and self.compute_labels

    if isinstance(clusterer, int):
        clusterer = AgglomerativeClustering(n_clusters=self.n_clusters)
    self.subcluster_norms = row_norms(self.subcluster_centers, squared=True)
    self.subcluster_labels = clusterer.fit_predict(self.subcluster_centers)
```

Ilustración 24. Función para agrupar los subclústeres.

- Función predict: se encarga de realiza la etapa de prueba del algoritmo de Birch, en donde se obtendrá la asignación de cada uno de los datos de acuerdo con la cercanía entre los

agrupamientos realizados. Esta función requiere como entrada el conjunto de datos de prueba.

```
def predict(self,X):
    dist = euclidean_distances(X,self.subcluster_centers,squared=True)
    argmin = np.argmin(dist,axis=1)
    return self.subcluster_labels[argmin]
```

Ilustración 25. Función de predicción.

- Función accuracy: se encarga de brindar el orden de los grupos, la métrica de exactitud entre las predicciones hechas por el algoritmo y el conjunto de prueba 'y'; y el número de aciertos obtenidos. Esta función requiere como entradas las predicciones obtenidas y los valores del conjunto de prueba 'y'.

```
def accuracy(y,predictions):
    y = np.array(y)
    allPermutations = np.array(list(permutations(np.unique(y))))
    acc = []
    for perm in allPermutations:
        classes = np.arange(0,y.shape[0])
        for index in range(classes.shape[0]):
            classes[index]=np.where(y[index]== perm)[0][0]
        acc.append(np.sum(classes==predictions))
    acc = np.array(acc)
    bestAccIndex = np.where(max(acc)==acc)[0][0]
    return dict(zip(np.arange(0,allPermutations.shape[1]),allPermutations[bestAccIndex])), acc[bestAccIndex]
```

Ilustración 26. Función para calcular la exactitud.

Diagrama de metodología

La metodología se dividió en preparación y tratamiento de los datos, codificación del algoritmo K-Means++ y elaboración del método BIRCH:

- Recolección: Definir y cargar la base de datos a utilizar.
- Preparar: Convertir las características lingüísticas a valores categóricos, comprobar la no existencia de datos faltantes, identificación de outliers, así como la normalización de los datos.
- Analizar: Conocer la distribución de la información por cada atributo, reducción de dimensionalidad, división de los conjuntos de entrenamiento y prueba del modelo, asignando el 80% de ellos para el entrenamiento y el 20 % restante a la etapa de prueba.

- Etapa de entrenamiento y prueba: Selección e implementación de los algoritmos de agrupamiento. Así bien, se calcula la métrica de exactitud para conocer el desempeño obtenido.

A continuación, se muestra el diagrama de flujo que representa el proceso de desarrollo de la presente tarea, el cual fue implementado en un programa basado en lenguaje Python.

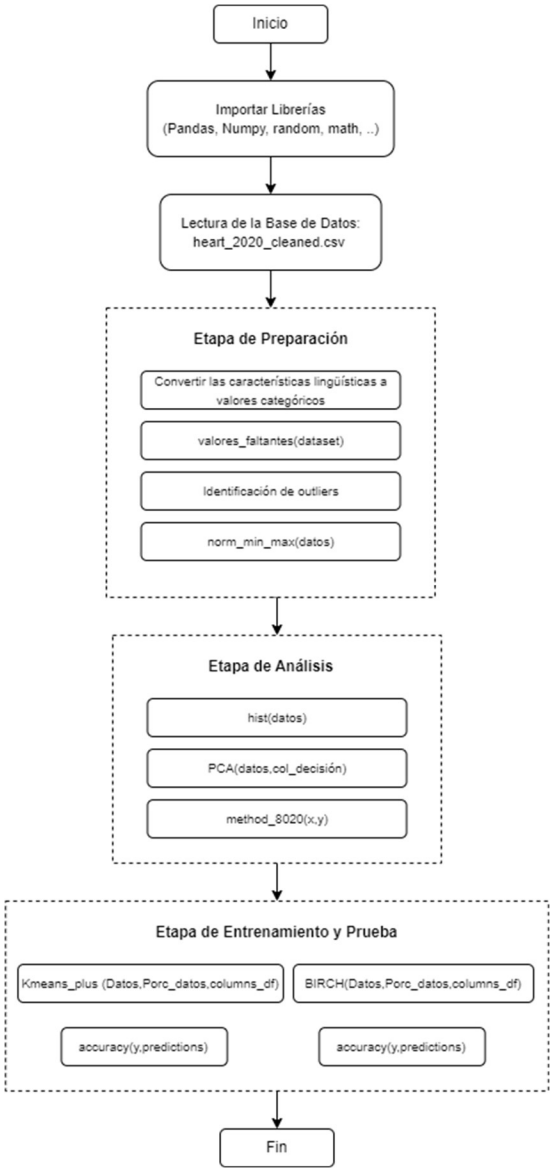
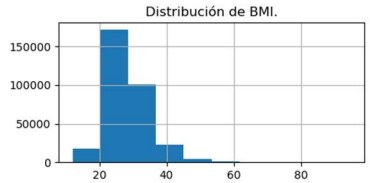
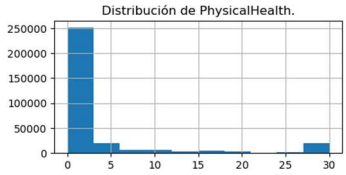


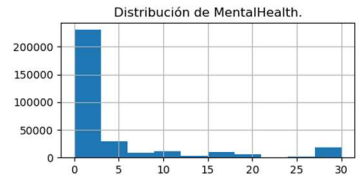
Ilustración 15. Diagrama de flujo para el análisis del conjunto de datos.

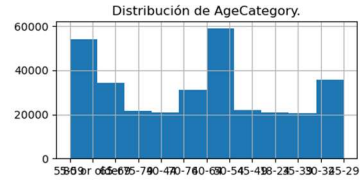
Resultados y discusión

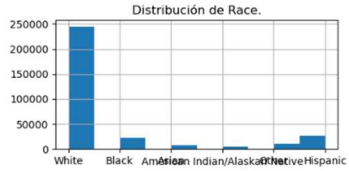
A continuación, se presentan las distribuciones de los atributos de la base de datos en cuestión, después de haberle realizado un subsampling del 5%, al cual se le efectuó una preparación, la cual comprende los siguientes puntos: la conversión de características lingüísticas a valores categóricos, la comprobación de la no existencia de datos faltantes, la identificación de outliers y la normalización de los datos.

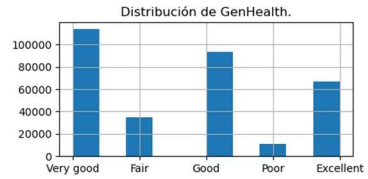


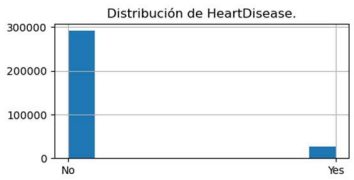


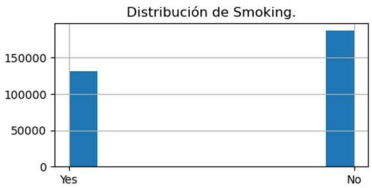


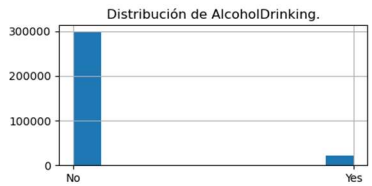


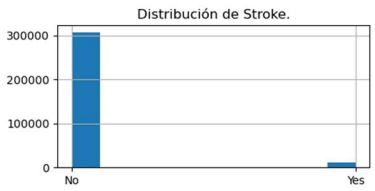


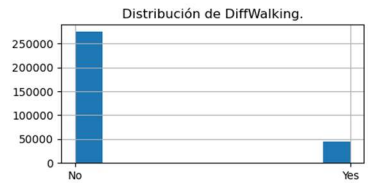


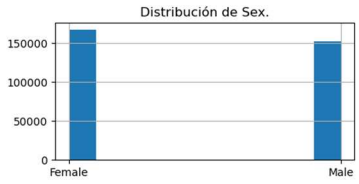


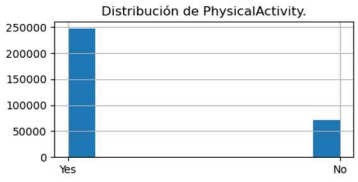












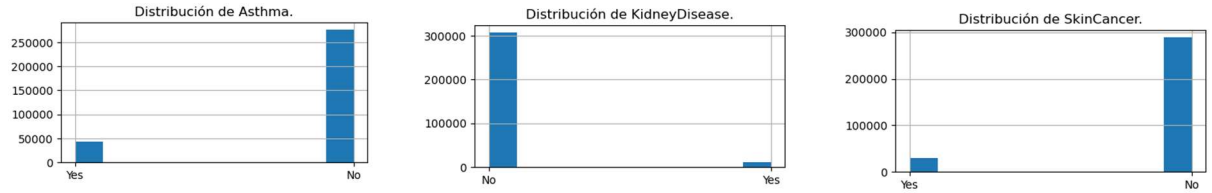


Ilustración 27. Distribución de los atributos de la base de datos.

A partir de haber implementado la técnica de análisis de componentes principales (PCA) en la base de datos en cuestión, se logró reducir la cantidad de atributos de 18 a 12, entre los atributos que contenían la mayor información se encuentran: BMI, PhysicalHealth, MentalHealth, AgeCategory, Race, Diabetic, Smoking, AlcoholDrinking, Stroke, DiffWalking y PhysicalActivity. Así bien, para los siguientes pasos se incluye a estos atributos seleccionados el atributo de decisión HeartDisease.

Antes de proseguir con la implementación de algoritmos de agrupamiento será necesario dividir el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba. En la Tabla 1 se presentan el total de los datos para cada uno de los conjuntos, en donde se utilizó la división 80/20.

Tabla 1. Conjunto de Entrenamiento y Prueba.

	Conjunto de Entrenamiento	Conjunto de Prueba
Total de datos	12792	3198

Algoritmos de Agrupamiento

En la Ilustración se puede visualizar la gráfica correspondiente a los atributos de BMI y PhysicalHealth, los cuales por medio de la técnica de análisis de componentes principales (PCA) se pudo determinar que eran los atributos con una mayor relevancia del conjunto de datos en cuestión.

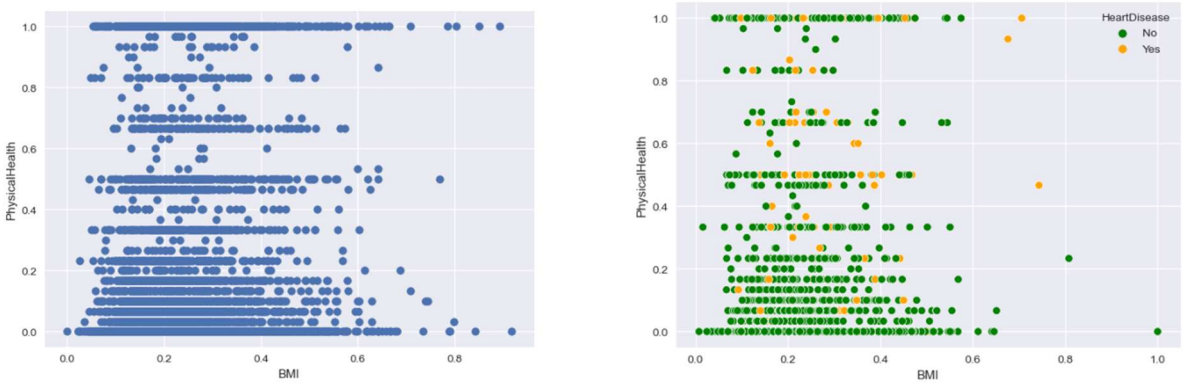


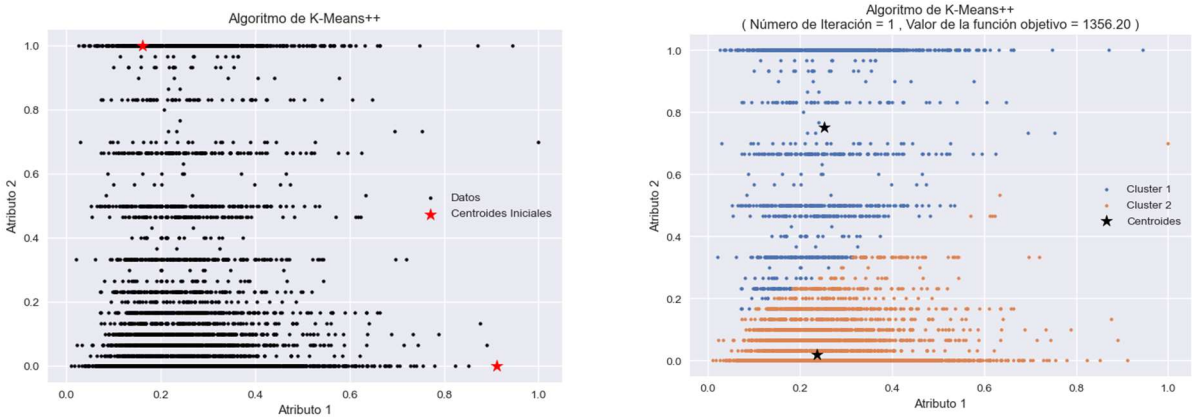
Ilustración 28. . Distribución de los atributos BMI y PhysicalHealth de la base de datos con y sin tomar en cuenta el atributo de decisión HeartDisease.

Así mismo, se presentan los resultados obtenidos de cada una de las técnicas de agrupamiento abordado en este trabajo.

Algoritmo K-Means ++

A continuación, se pueden observar los resultados que el algoritmo de K-Means ++ brindó, en donde se utilizaron los atributos con mayor relevancia en la base de datos: BMI y PhysicalHealth para su implementación.

En la Ilustración 29 se puede observar cómo se van reubicando los centroides, así como la reasignación de los datos con respecto a las nuevas ubicaciones de los centroides. Así bien, el tiempo de ejecución de la etapa de entrenamiento fue 12.3234744072 segundos.



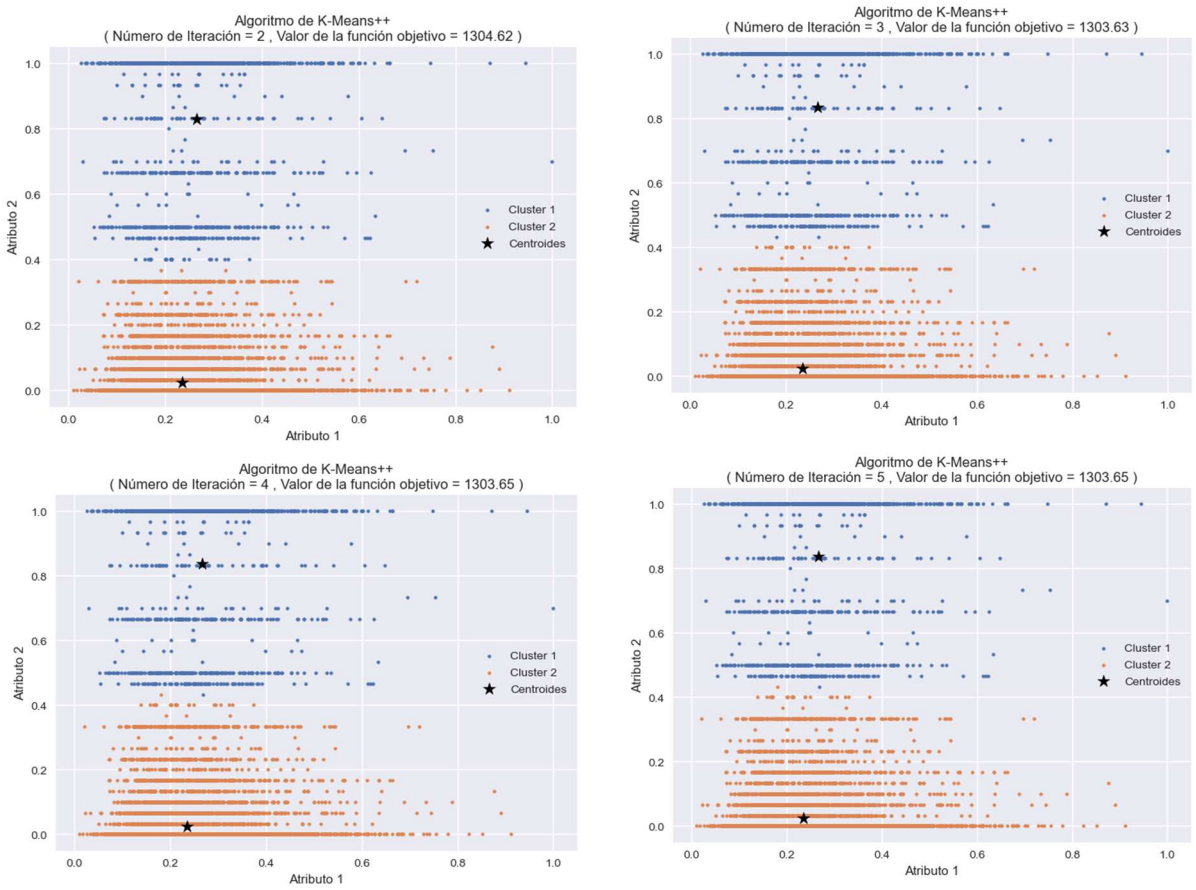


Ilustración 29.Implementación del algoritmo de K-Means ++.

En la Ilustración 30 se pueden observar las variaciones en la función objetivo entre cada iteración, en donde a partir de la iteración 3 el valor de la función objetivo varia por milésimas con respecto a las iteraciones 4 y 5.

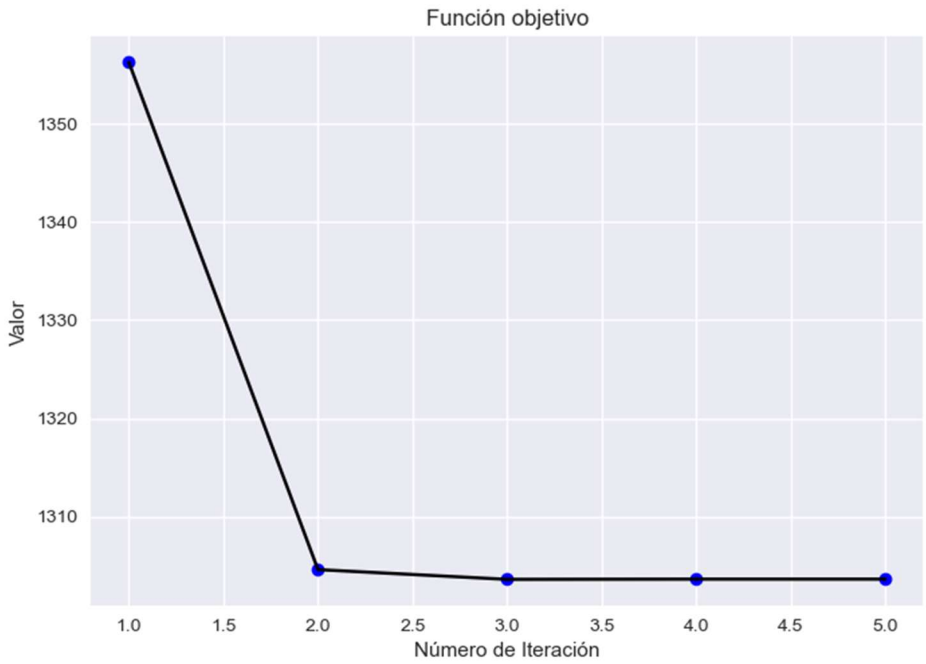


Ilustración 30. Comportamiento del modelo con cada iteración: gráfica de codo.

En la Ilustración se puede observar del lado derecho el resultado final posterior a la implementación de la técnica de K-Means++, en donde cada una de las agrupaciones corresponde a la predicción propuesta con respecto al atributo de decisión. Sin embargo, dichas agrupaciones no son del todo parecidas a las originales.

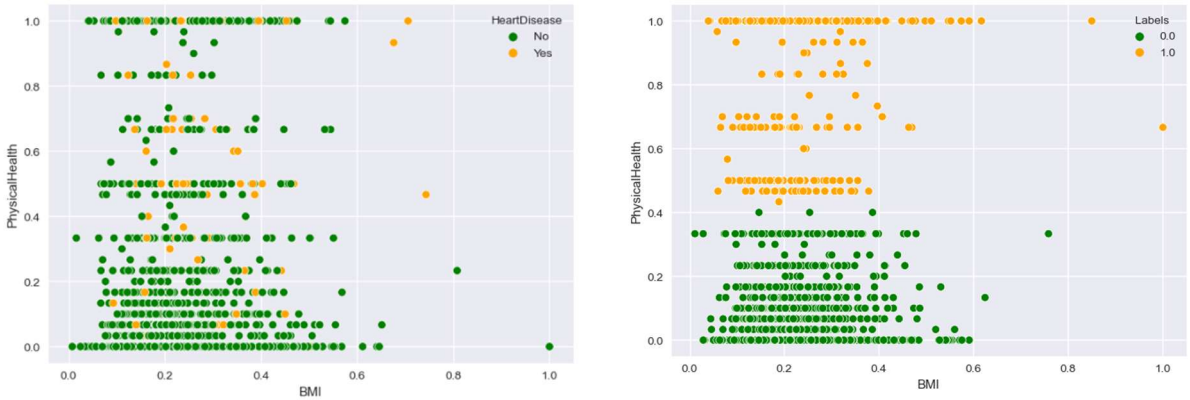


Ilustración 31. Comparación de la clasificación brindada del atributo de decisión de la base de datos y las agrupaciones propuestas por el algoritmo de K-Means++.

En la Tabla 2 se presenta los resultados obtenidos en la etapa de prueba al utilizar la métrica de exactitud.

Tabla 2. Resultados en la etapa de Prueba del algoritmo de k-means ++.

Orden de los grupos:	Aciertos:	Porcentaje [%]:
0: 'No', 1: 'Yes'	2734	85.49093183239525

Algoritmo BIRCH

A continuación, se pueden observar los resultados que la técnica de agrupamiento BIRCH brindo, en donde se probaron con diferentes valores para los parámetros de Threshold y Branching factor, a fin de observar el comportamiento que el algoritmo presentaba.

Threshold = 0.5, Branching factor = 50, n_clusters = 2:

En la Ilustración 32 se puede observar del lado derecho el resultado obtenido posterior a la implementación del algoritmo de BIRCH, con un Threshold de 0.5, Branching factor de 50 y n_cluster de 2, en donde se puede observar que la clasificación propuesta por el algoritmo es similar a la clasificación original. Así bien, el tiempo de ejecución de la etapa de entrenamiento fue 0.1989953518 segundos.

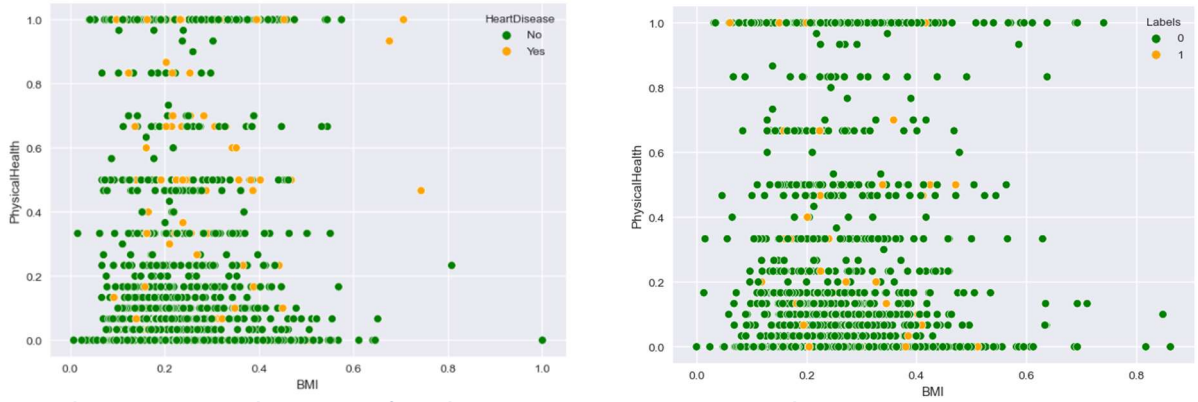


Ilustración 32. Comparación de la clasificación brindada del atributo de decisión de la base de datos y las agrupaciones propuestas por el algoritmo de Birch.

En la Tabla 3 se presentan los resultados obtenidos en la etapa de prueba al utilizar la métrica de exactitud.

Tabla3. Resultados en la etapa de Prueba.

Orden de los grupos:	Aciertos:	Porcentaje:
0: 'No', 1: 'Yes'	2888	90.30644152595372 %

Threshold = 0.3, Branching factor = 50, n_clusters = 2

En la Ilustración se puede observar del lado derecho el resultado obtenido posterior a la implementación del algoritmo de BIRCH, con un Threshold de 0.3, Branching factor de 50 y n_cluster de 2; en donde al comparar el resultado actual con el anterior se observa que la clasificación brindo una mayor numero de etiquetas con padecido de enfermedad en el corazón. Así bien, el tiempo de ejecución de la etapa de entrenamiento fue 0.1825356483 segundos.

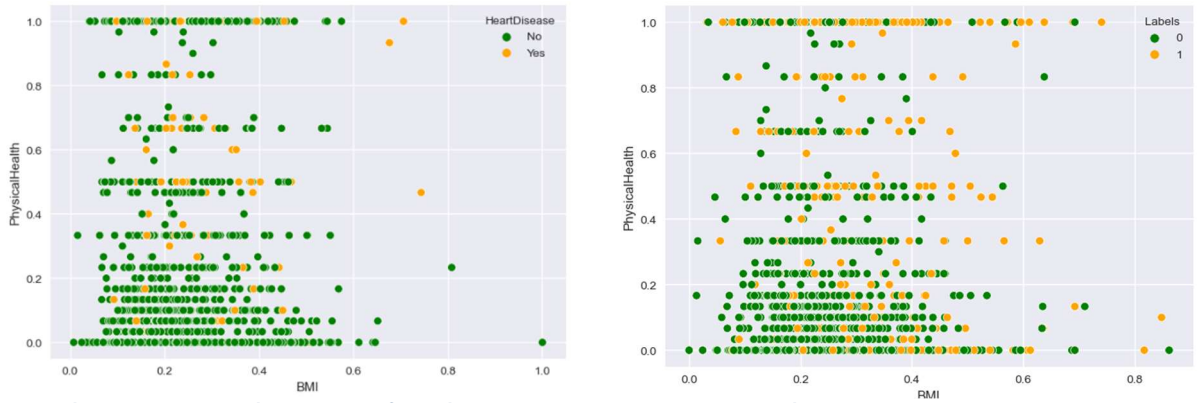


Ilustración 33. Comparación de la clasificación brindada del atributo de decisión de la base de datos y las agrupaciones propuestas por el algoritmo de Birch.

En la Tabla 4 se presenta los resultados obtenidos en la etapa de prueba al utilizar la métrica de exactitud.

Tabla 4. Resultados en la etapa de Prueba.

Orden de los grupos:	Aciertos:	Porcentaje:
0: 'No', 1: 'Yes'	2679	83.77110694183865 %

Threshold = 0.7, Branching factor = 50, n_clusters = 2

En la Ilustración se puede observar del lado derecho el resultado obtenido posterior a la implementación del algoritmo de BIRCH, con un Threshold de 0.7, Branching factor de 50 y n_cluster de 2; el cual al compararlo con los resultados anteriores brinda un resultado más parecido con la clasificación original. Así bien, el tiempo de ejecución de la etapa de entrenamiento fue 0.1653952599 segundos.

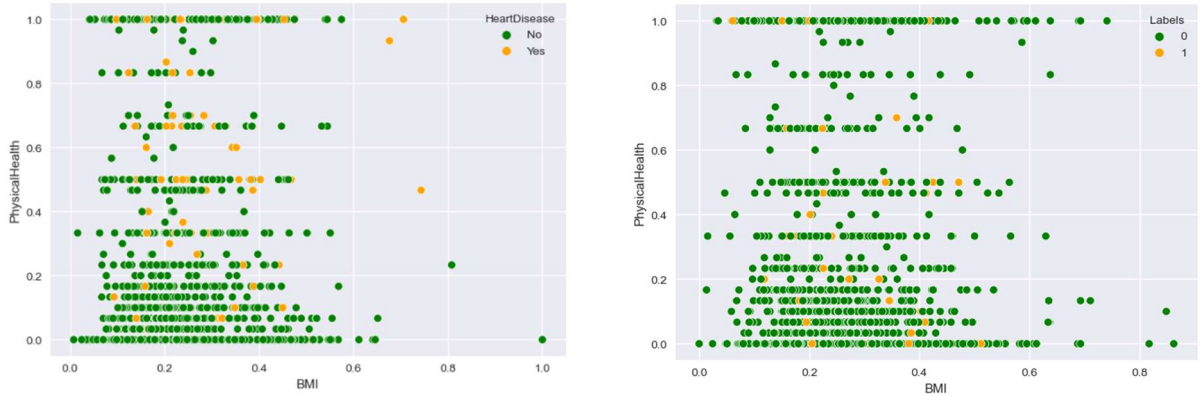


Ilustración 34. Comparación de la clasificación brindada del atributo de decisión de la base de datos y las agrupaciones propuestas por el algoritmo de Birch.

En la Tabla 5 se presenta los resultados obtenidos en la etapa de prueba al utilizar la métrica de exactitud.

Tabla 5. Resultados en la etapa de Prueba.

Orden de los grupos:	Aciertos:	Porcentaje:
0: 'No', 1: 'Yes'	2889	90.33771106941839 %

Threshold = 0.5, Branching factor = 30, n_clusters = 2

En la Ilustración se puede observar del lado derecho el resultado obtenido posterior a la implementación del algoritmo de BIRCH, con un Threshold de 0.5, Branching factor de 30 y n_cluster de 2; el cual sigue presentando una distribución algo parecida a la original. Así bien, el tiempo de ejecución de la etapa de entrenamiento fue 0.1809995174 segundos.

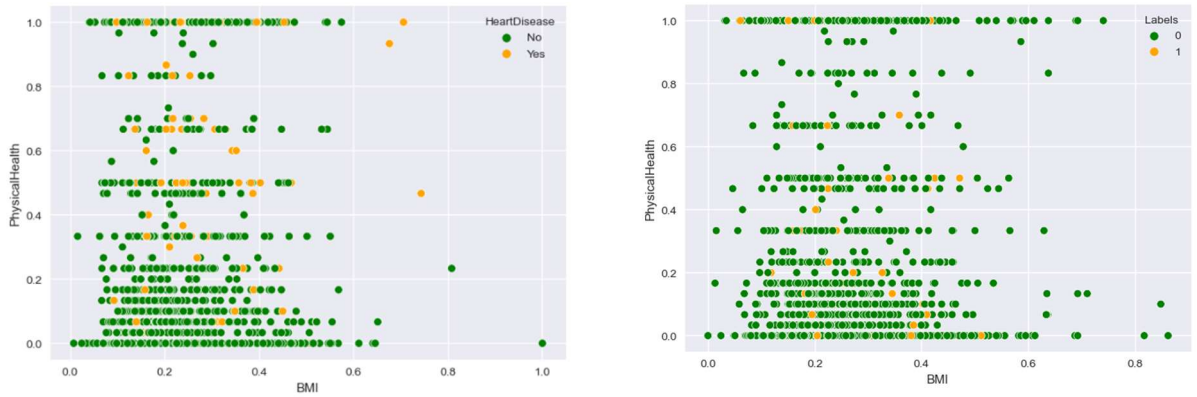


Ilustración 35. Comparación de la clasificación brindada del atributo de decisión de la base de datos y las agrupaciones propuestas por el algoritmo de Birch.

En la Tabla 6 se presenta los resultados obtenidos en la etapa de prueba al utilizar la métrica de exactitud. Al comparar el porcentaje obtenido con el anterior se puede observar que tuvo una variación de centésimas.

Tabla 6. Resultados en la etapa de Prueba.

Orden de los grupos:	Aciertos:	Porcentaje:
0: 'No', 1: 'Yes'	2890	90.36898061288305 %

Threshold = 0.5, Branching factor = 70, n_clusters = 2

En la Ilustración se puede observar del lado derecho el resultado obtenido posterior a la implementación del algoritmo de BIRCH, con un Threshold de 0.5, Branching factor de 70 y n_cluster de 2; el cual brindo un mayor porcentaje de la etiqueta de no contar con alguna

enfermedad en el corazón. Así bien, el tiempo de ejecución de la etapa de entrenamiento fue 0.1796615124 segundos.

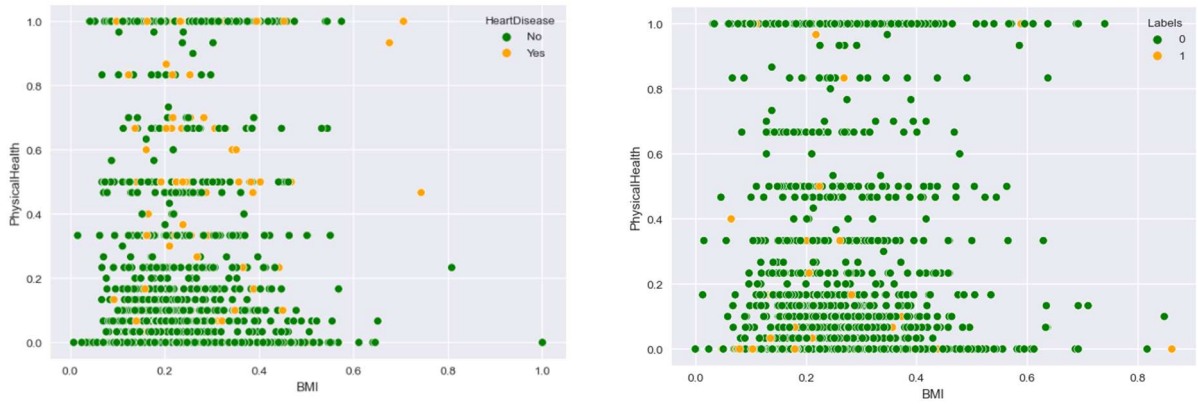


Ilustración 36. Comparación de la clasificación brindada del atributo de decisión de la base de datos y las agrupaciones propuestas por el algoritmo de Birch.

En la Tabla 7 se presenta los resultados obtenidos en la etapa de prueba al utilizar la métrica de exactitud.

Tabla 7. Resultados en la etapa de Prueba.

Orden de los grupos:	Aciertos:	Porcentaje:
0: 'No', 1: 'Yes'	2724	85.17823639774859 %

En la siguiente tabla se presenta la comparación entre los resultados obtenidos al modificar los parámetros del Threshold y Branching factor, en donde es posible observar los porcentajes obtenidos son mayores al 80%. Así bien, el tiempo de entrenamiento para la cantidad de datos que se están utilizado es realmente destacable.

Tabla 8. Resultados en la etapa de Prueba del algoritmo de Birch.

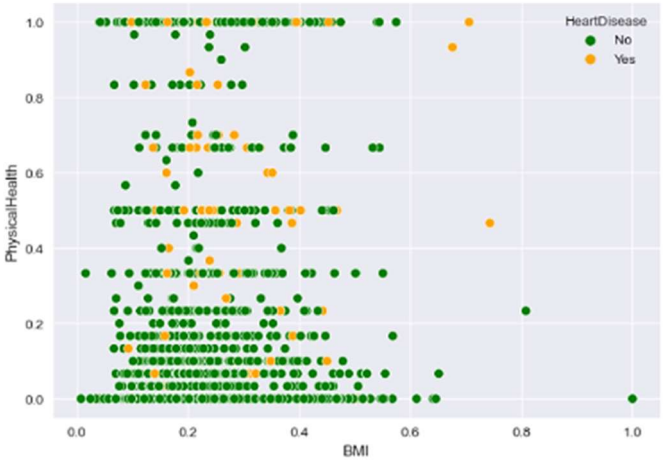
Parámetros	Orden de los grupos	Aciertos	Porcentaje	Tiempo
Threshold = 0.5, Branching factor = 50, n_clusters = 2	0: 'No', 1: 'Yes'	2888	90.30644 %	0.1989953518 segundos
Threshold = 0.3, Branching factor = 50, n_clusters = 2	0: 'No', 1: 'Yes'	2679	83.77110 %	0.1825356483 segundos.
Threshold = 0.7, Branching factor = 50, n_clusters = 2	0: 'No', 1: 'Yes'	2889	90.33771%	0.1653952599 segundos
Threshold = 0.5, Branching factor = 30, n_clusters = 2	0: 'No', 1: 'Yes'	2890	90.36898 %	0.1809995174 segundos
Threshold = 0.5, Branching factor = 70, n_clusters = 2	0: 'No', 1: 'Yes'	2724	85.17823 %	0.1796615124 segundos.

En la Tabla 9, se presenta la comparación de los resultados obtenidos entre los dos algoritmos de agrupamiento implementados para este trabajo, en donde es posible observar que el algoritmo de Birch presento mejores resultados tanto en el porcentaje de exactitud como en el tiempo de ejecución en la etapa de entrenamiento.

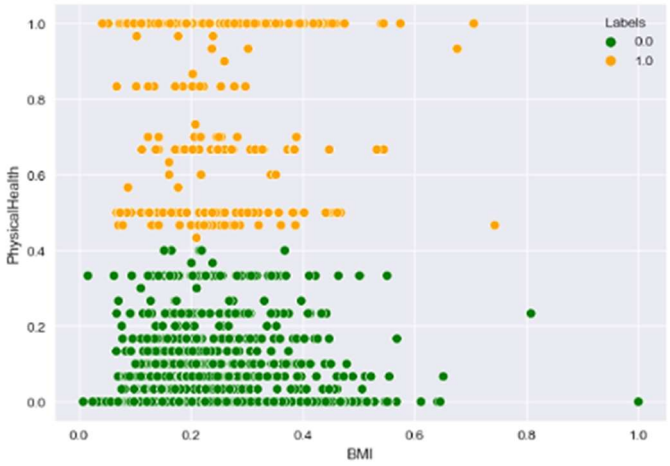
Tabla 9. Resultados en la etapa de Prueba del algoritmo de K-Means++ y Birch.

Algoritmo de Agrupamiento	Orden de los grupos	Aciertos	Porcentaje	Tiempo de Ejecución en la etapa de entrenamiento
K-Means ++	0: 'No', 1: 'Yes'	2734	85.49093 %	12.3234744072 segundos
Birch	0: 'No', 1: 'Yes'	2890	90.36898 %	0.1809995174 segundos

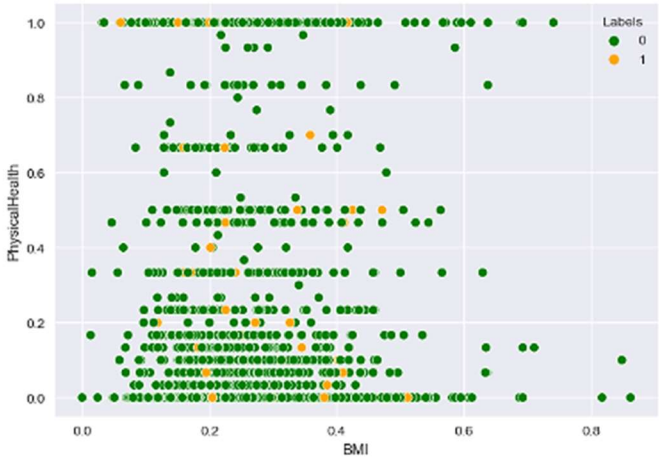
Por otra parte, se muestra en la siguiente ilustración comparación entre las dos técnicas implementados: K-means++ y BIRCH.



Distribución Original



Distribución con K-means++



Distribución con BIRCH

Por otra parte, los resultados presentados en la ilustración anterior demuestran que el algoritmo de BIRCH brindo un mejor agrupamiento de los datos comparado con la técnica de K-Means++. La razón por la que se realiza la comparación entre estos dos métodos de agrupación y el atributo de decisión HeartDisease es debido a que se busca mantener una agrupación muy parecida a la original. Sin embargo, al tratarse de métodos no supervisados es común que no se pueda realizar este tipo de comparación debido a que la implementación de estos algoritmos suele enfocarse a bases de datos sin etiquetar.

Conclusiones

En este programa generalizado inicialmente se generaron funciones para preparar la base de datos en cuestión a fin de poder implementar posteriormente dos métodos de agrupamiento, esto con el propósito de identificar los puntos fuertes de cada uno de ellos.

Comenzando con la técnica de K-Means ++, como se mencionó anteriormente es uno de los algoritmos de aprendizaje automático no supervisado más sencillos y populares. Sin embargo, como se pudo comprobar al realizar su correspondiente implementación los agrupamientos brindados no son tan parecidos a la clasificación original, así bien el tiempo de ejecución fue considerablemente rápido tomando en cuenta la cantidad de instancias a consideradas.

Por otro lado, el algoritmo de BIRCH es un algoritmo de agrupamiento eficiente para muchos grandes bases de datos, ya que encuentra un buen agrupamiento con un solo escaneo. Así bien, con respecto al conjunto de datos propuesto para esta actividad este algoritmo brindó un mejor desempeño con respecto al método de K-Means++. Además, su predicción fue muy similar al atributo de decisión de la base de datos en cuestión.

Es importante destacar la importancia de comprender los datos antes de adoptar alguna técnica de agrupamiento que se adapte bien a un conjunto de datos determinado para el problema en cuestión, ya que si se hace un análisis con respecto al algoritmo a utilizar puede brindar un ahorro el tiempo, así como la obtención de resultados más precisos.

Referencias

[1]“8 algoritmos de agrupación en clústeres en el aprendizaje automático que todos los científicos de datos deben conocer.” [Online]. Available: https://www.freecodecamp.org/espanol/news/8-algoritmos-de-agrupacion-en-clusteres-en-el-aprendizaje-automatico-que-todos-los-cientificos-de-datos-deben-conocer/. [Accessed: 28-Mar-2022].

[2]“Algoritmos de Agrupamiento - 🤖 Aprende IA.” [Online]. Available: https://aprendeia.com/algoritmos-de-clustering-agrupamiento-aprendizaje-no-supervisado/. [Accessed: 28-Mar-2022].

[3] M. Antonio and A. Fernández, *Inteligencia artificial para programadores con prisa by Marco Antonio Aceves Fernández - Books on Google Play*. Universo de Letras. [Online]. Available: https://play.google.com/store/books/details/Inteligencia_artificial_para_programadores_con_pri?id=ieFYEAAAQBAJ&hl=en_US&gl=US

[4] L. Laura-Ochoa, “Evaluation of Classification Algorithms using Cross Validation,” Ind. Innov. Infrastruct. Sustain. Cities Communities, pp. 24–26, 2019, doi: 10.18687/LACCEI2019.1.1.471.].

[5] Ramadhani, F., Muhammad, Z., Suwila,S. (2020). Improve BIRCH algorithm for big data clustering. IOP Conf. Ser.: Mater. Sci. Eng. from https://iopscience.iop.org/article/10.1088/1757-899X/725/1/012090/pdf

[6] T. Zhang, R. Ramakrishnan and M. Livny, “BIRCH: an efficient data clustering method for very large databases” in ACM Sigmod Record, ACM, vol. 25, pp. 103–114.

[7]“ML | K-means++ Algorithm - GeeksforGeeks.” [Online]. Available: https://www.geeksforgeeks.org/ml-k-means-algorithm/. [Accessed: 28-Mar-2022].