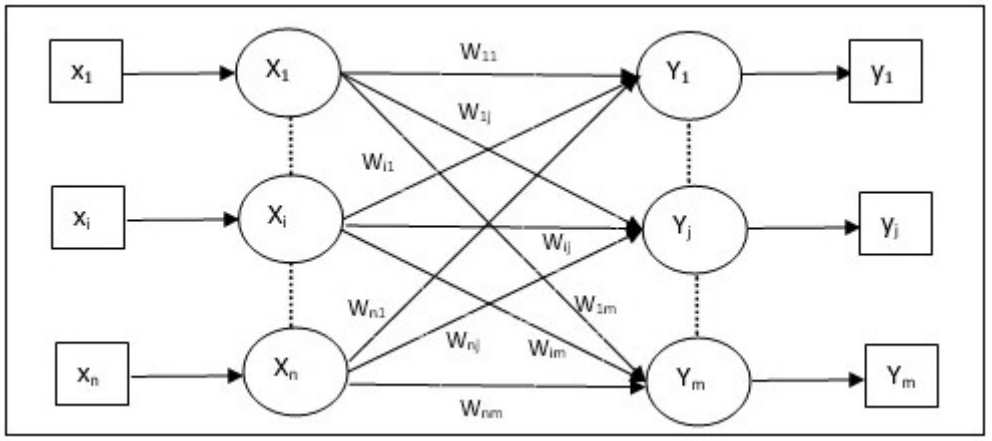
Mapas Autoorganizativos (SOM) para Clustering

- Entendiendo el algoritmo SOMKohonen
- Principales componentes y lógica del código
- Visualización del entrenamiento y datos de prueba



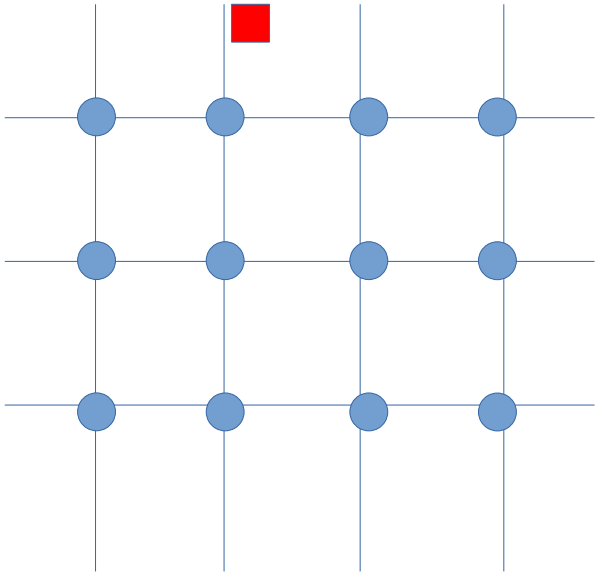
¿Qué es un Mapa Autoorganizado (SOM)?

- Un algoritmo de aprendizaje no supervisado para clustering y visualización.
- Proyecta datos de alta dimensión en una rejilla de menor dimensión.
- Aprende mediante aprendizaje competitivo – las neuronas compiten para representar los datos.

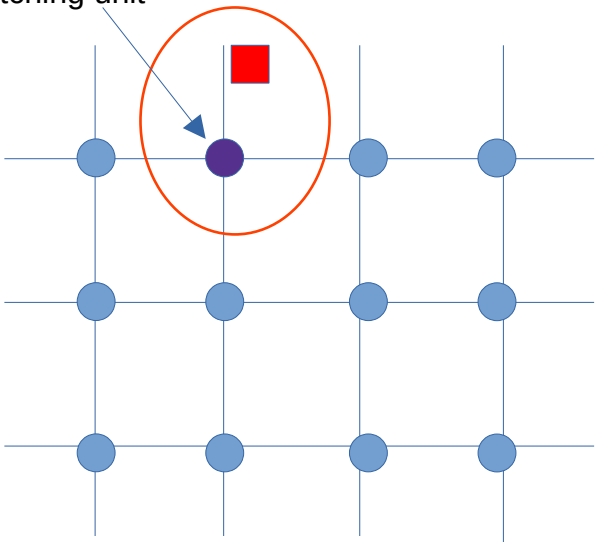
Aplicaciones

- Clustering y reconocimiento de patrones
- Reducción de dimensionalidad
- Extracción de características

Entendiendo la Mejor Unidad de Coincidencia (BMU) en SOM de Forma Intuitiva

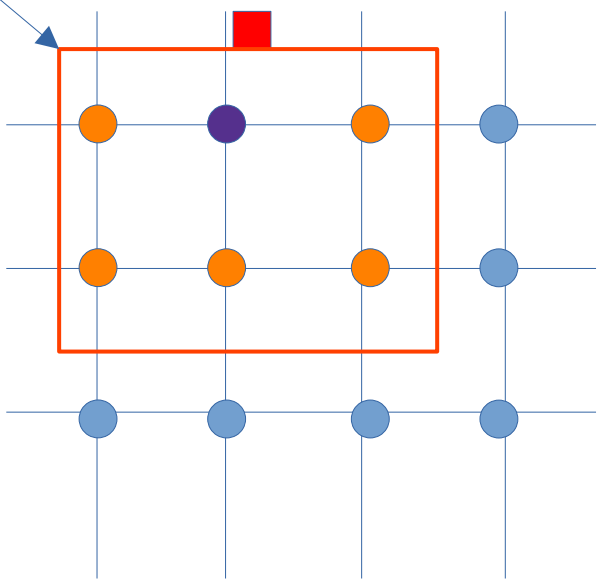


BMU- Best matching unit



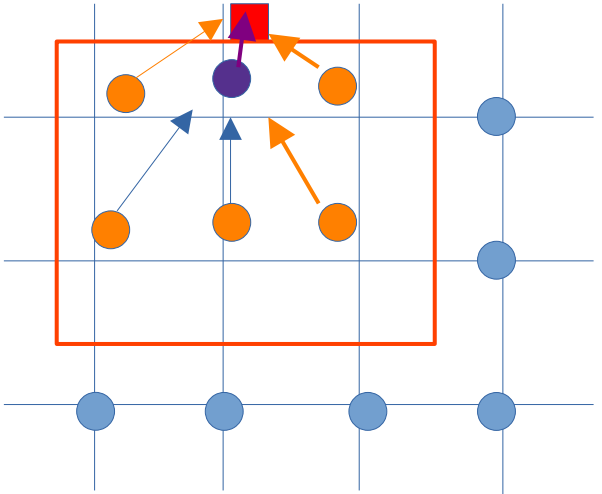
$$BMU = \arg\min_i \|x - w_i\|$$

Neighborhood Neurons



La BMU es la mejor porque:

- Es la más parecida al punto de datos en términos de similitud.
- Su vector de pesos es el más cercano al dato de entrada.
- El proceso de aprendizaje ajusta la BMU (y sus vecinos) para que se parezcan más a los datos, refinando el mapa.



The update step:

Clase: SOMKohonen: Define la rejilla SOM y la lógica de entrenamiento.
Métodos:

- `train(data, epochs)`: Aprende los patrones de los datos.
- `_find_bmu(x)`: Encuentra la unidad de mejor coincidencia (BMU).
- `_update_weights(x, bmu_row, bmu_col)`: Ajusta los pesos de las neuronas.
- `plot_clusters(data, test_data)`: Visualiza los clusters y datos de prueba.

```python
class SOMKohonen:
    """
    A simple Self-Organizing Map (Kohonen Map) for clustering.
    This class supports a 2D grid of neurons, each with a weight vector in
'input_dim'.
    """

    def __init__(
        self,
        map_size=(10, 10),    # shape of the SOM grid
        input_dim=2,            # dimension of input data
        learning_rate=0.5,
        sigma=3.0,               # initial neighborhood radius
        lr_decay=0.99,          # learning rate decay per epoch
        sigma_decay=0.99,      # sigma decay per epoch
        seed=None
    ):
        if seed is not None:
            np.random.seed(seed)
            random.seed(seed)

        self.map_size = map_size
        self.input_dim = input_dim
        self.learning_rate = learning_rate
        self.sigma = sigma
        self.lr_decay = lr_decay
        self.sigma_decay = sigma_decay

        # Initialize SOM weights: shape = [rows, cols, input_dim]
        self.weights = np.random.rand(map_size[0], map_size[1], input_dim)
```

- Rejilla SOM: 10×10 neuronas
- Cada neurona tiene un vector de pesos con la misma dimensión que los datos
- Pesos inicializados aleatoriamente para el clustering

```
# Initialize SOM weights: shape = [rows, cols, input_dim]
self.weights = np.random.rand(map_size[0], map_size[1], input_dim)
```

esto significa que cada neurona (nodo) en la rejilla 2D tiene un vector de pesos, y estos valores son asignados aleatoriamente al inicio.

- La rejilla SOM en sí misma es 2D → Tiene una estructura organizada (por ejemplo, una malla de 10×10).
- Cada neurona en la rejilla está asociada con un vector de pesos de dimensión input_dim.
- Estos pesos existen en el mismo espacio que los datos de entrada (no en la rejilla 2D, sino en el espacio de características).

Ejemplo: Si input_dim = 3 (Agrupamiento de colores RGB)

- Cada neurona en una rejilla 10×10 tiene un vector de 3 dimensiones (R,G,B) que representa un color:

```
Posición en la Rejilla (2D)  →  Vector de Pesos (Espacio de Color RGB)
------------------------------------------------------------------
(0,0)  →  [0.8, 0.2, 0.3]   (Color marrón)
(0,1)  →  [0.1, 0.9, 0.2]   (Color verdoso)
(1,0)  →  [0.5, 0.5, 0.5]   (Color gris)
(9,9)  →  [0.9, 0.1, 0.7]   (Color rosado)
```

Aunque las neuronas están organizadas en 2D, sus vectores de pesos existen en el espacio de características (en este caso, RGB).

Proceso de Entrenamiento:

1. Barajar los datos en cada época
2. Encontrar la Mejor Unidad de Coincidencia (BMU) para cada muestra
3. Actualizar los pesos de la BMU y sus vecinos
4. Reducir la tasa de aprendizaje y el radio de vecindad

```
def train(self, data, epochs=100):
      """
      Train the SOM using the given data for the specified number of epochs.
      """
      data = np.array(data)
      for epoch in range(epochs):
            np.random.shuffle(data)
            for x in data:
                  bmu_row, bmu_col = self._find_bmu(x)
                  self._update_weights(x, bmu_row, bmu_col)
            # Decay learning rate, sigma
            self.learning_rate *= self.lr_decay
            self.sigma *= self.sigma_decay
```

- SOM aprende los patrones de los datos progresivamente
- La influencia de la vecindad disminuye con el tiempo

Encontrar la Mejor Unidad de Coincidencia (BMU)
¿Cómo encuentra el SOM la neurona más cercana?

- Calcula la distancia euclidiana entre la entrada y todas las neuronas.
- La neurona con la menor distancia es la BMU.

```
def _find_bmu(self, x):
    """
    Find the Best Matching Unit (BMU) for input x.
    Returns (row, col) of the BMU in the grid.
    """
    diff = self.weights - x
    dist_sq = np.sum(diff**2, axis=2)
    bmu_idx = np.unravel_index(np.argmin(dist_sq), dist_sq.shape)
    return bmu_idx
```

Actualización de Pesos

Los pesos se actualizan según una función de vecindad

- Se actualizan la BMU y sus vecinos.
- Se usa una función gaussiana para determinar la influencia.

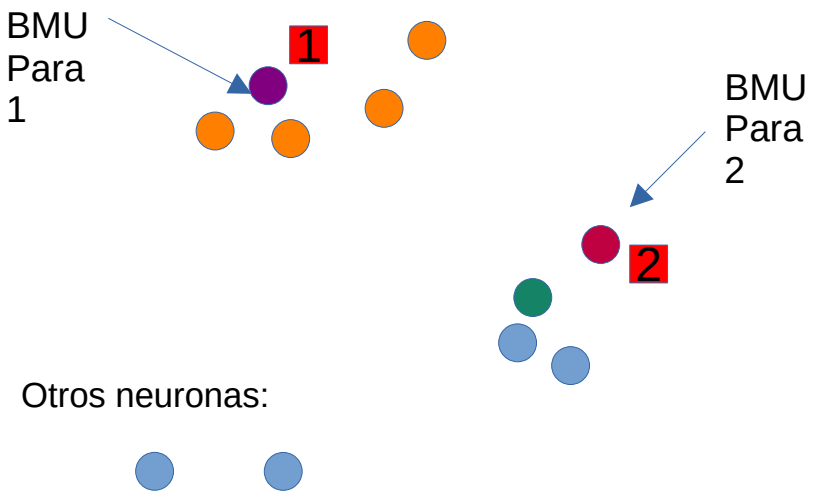
```
def _update_weights(self, x, bmu_row, bmu_col):
     """
     Update weights of BMU and its neighbors using Gaussian neighborhood function.
     """
     rows, cols, _ = self.weights.shape
     for i in range(rows):
          for j in range(cols):
               dist_sq = (i - bmu_row)**2 + (j - bmu_col)**2
               neigh_strength = np.exp(-dist_sq / (2.0 * (self.sigma**2)))
               self.weights[i, j, :] += self.learning_rate * neigh_strength
                                                    * (x - self.weights[i, j, :])
```

La BMU se actualiza más, los vecinos menos
La influencia de la vecindad disminuye con la distancia

Cada punto de datos se asigna a una neurona BMU

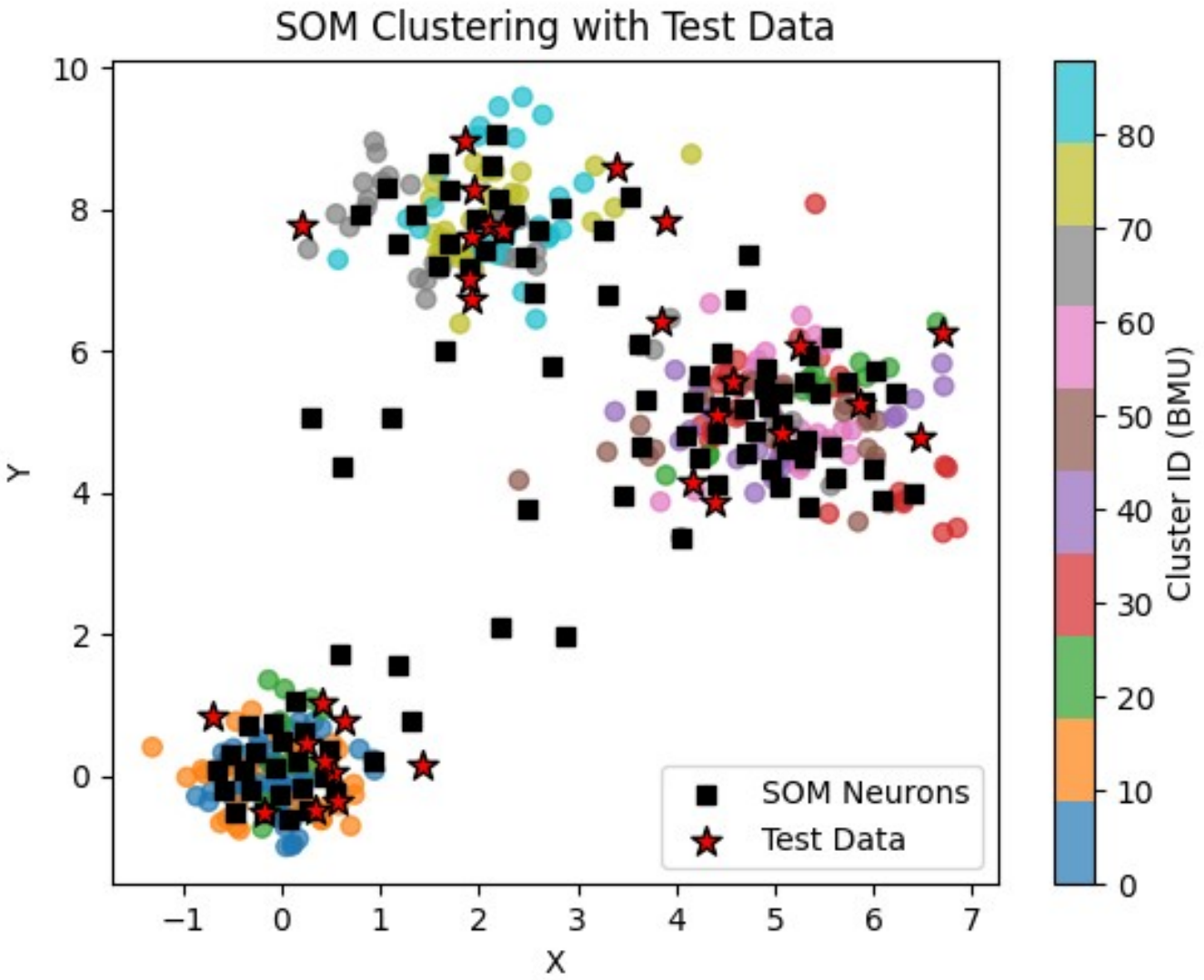
```
def get_cluster_assignments(self, data):
      """
      Find the BMU for each data point and return a list of (bmu_row, bmu_col).
      """
      assignments = []
      for x in data:
            bmu_row, bmu_col = self._find_bmu(x)
            assignments.append((bmu_row, bmu_col))
      return assignments
```

- Devuelve la ubicación BMU de cada punto de datos
- Agrupa puntos de datos similares juntos



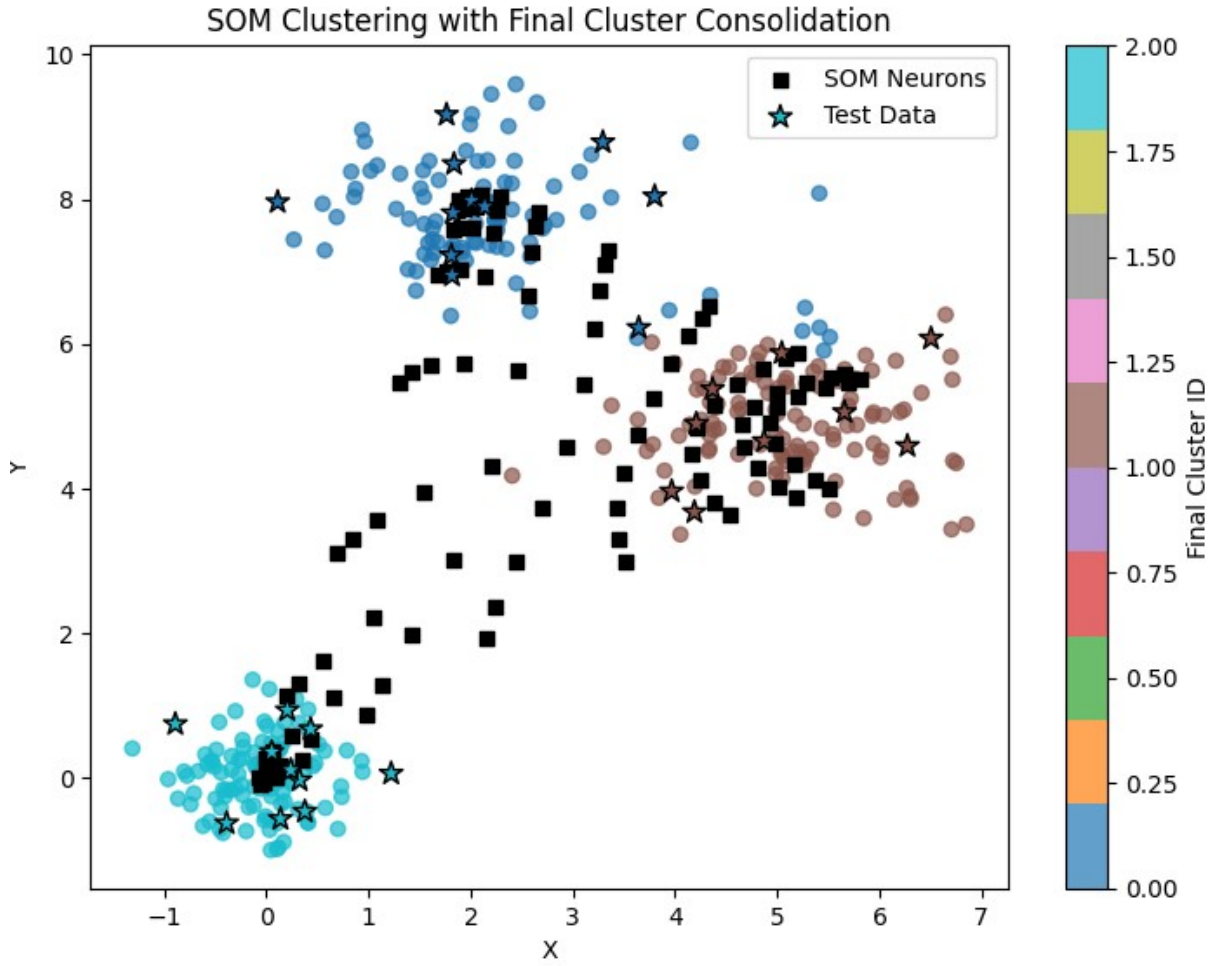
The update step:

Comportamiento del SOM



- *Múltiples BMUs por Cluster:* La cuadrícula del SOM tiene muchas neuronas, y varios BMUs pueden existir dentro del mismo cluster real.
- *Asignaciones Fragmentadas:* Los puntos de prueba se asignan a muchos BMUs diferentes en la misma región, en lugar de consolidarse.
- *Necesidad de Clustering Final:* El SOM se comporta más como un mapeo detallado que como un algoritmo de clustering, requiriendo post-procesamiento.

Adiciones de K-Means en el Post-Procesamiento



- *Consolidación de BMUs:* Después del entrenamiento del SOM, K-Means agrupa los BMUs en un número fijo de clusters finales (por ejemplo, 3).
- *Asignación Estable de Clusters:* Cada BMU se asigna a un cluster final, asegurando coherencia en el etiquetado.
- *Manejo de Nuevos BMUs:* Los BMUs de datos de prueba que no aparecieron en el entrenamiento se asignan al cluster de BMU más cercano existente.

Optimización por Enjambre de Partículas (PSO) para Clustering

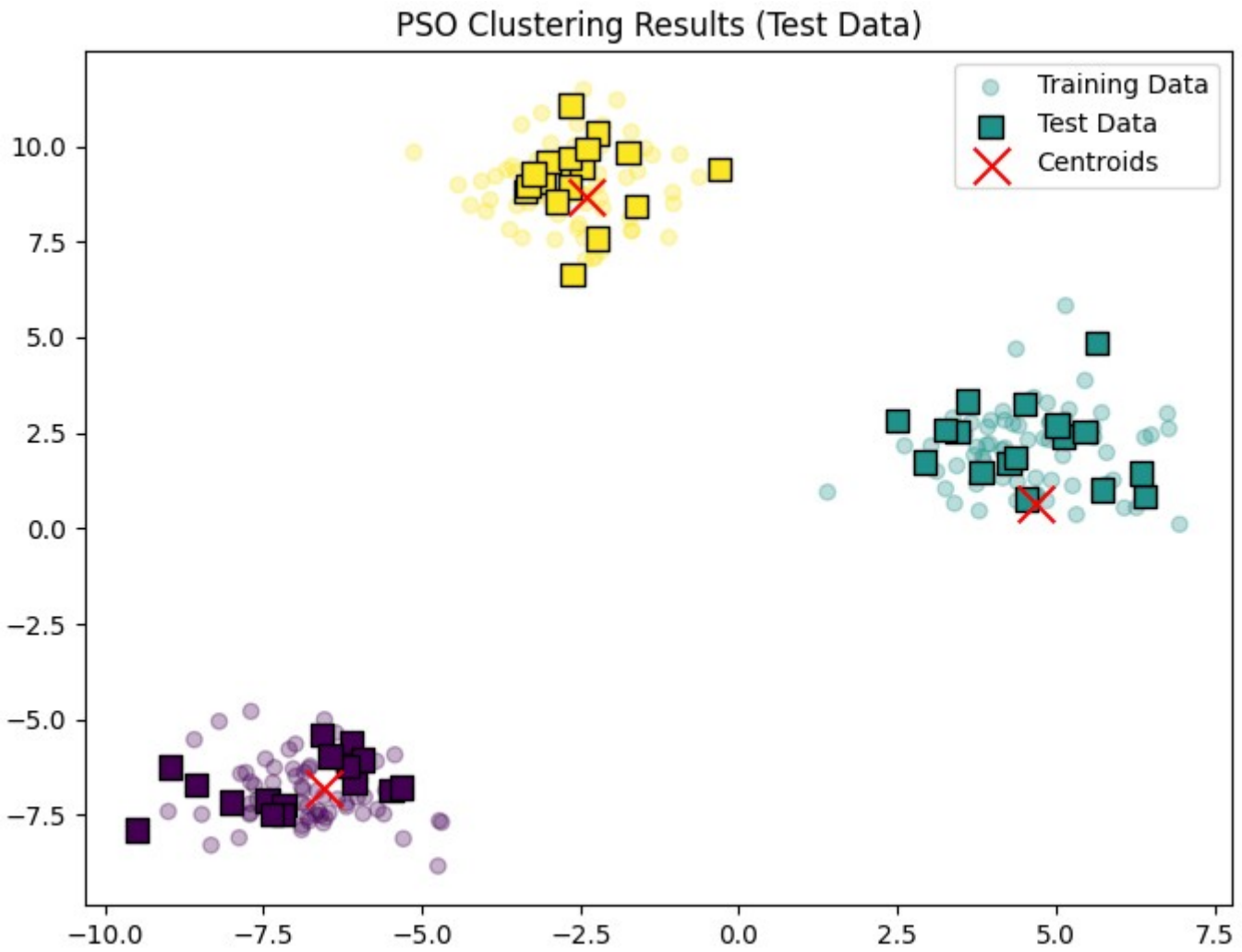
Descripción del Algoritmo

- Inspirado en la inteligencia de enjambre, PSO optimiza los centroides de clusters ajustando iterativamente las posiciones de las partículas.
- Cada partícula representa una posible solución de clustering, moviéndose según inercia, mejor posición personal y mejor posición global.
- Las partículas actualizan sus posiciones para minimizar las distancias intra-cluster, obteniendo centroides óptimos.
- Los clusters finales se asignan según la proximidad a los centroides optimizados.

```
# PSO for clustering
class PSO:
      def __init__(self, num_clusters, data,
                   num_particles=30,
                   max_iters=100):
            self.num_clusters = num_clusters
            self.data = data
            self.num_particles = num_particles
            self.max_iters = max_iters
            self.global_best_position = None
            self.global_best_error = np.inf
            self.particles = [Particle(num_clusters, data)
          for _ in range(num_particles)]
```

Qué Hace Este Código

- Genera datos sintéticos (blobs gaussianos) para clustering.
- Utiliza PSO para encontrar los centroides óptimos, minimizando la distancia intra-cluster.
- Asigna muestras de prueba a clusters según los centroides entrenados.
- Evalúa la calidad del clustering con puntuación de pureza.
- Grafica los resultados de entrenamiento y prueba con colores de cluster correctos.


PSO Clustering Results (Test Data)
Training Data
Test Data
Centroids
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5

Optimización por Colonia de Hormigas para Clustering

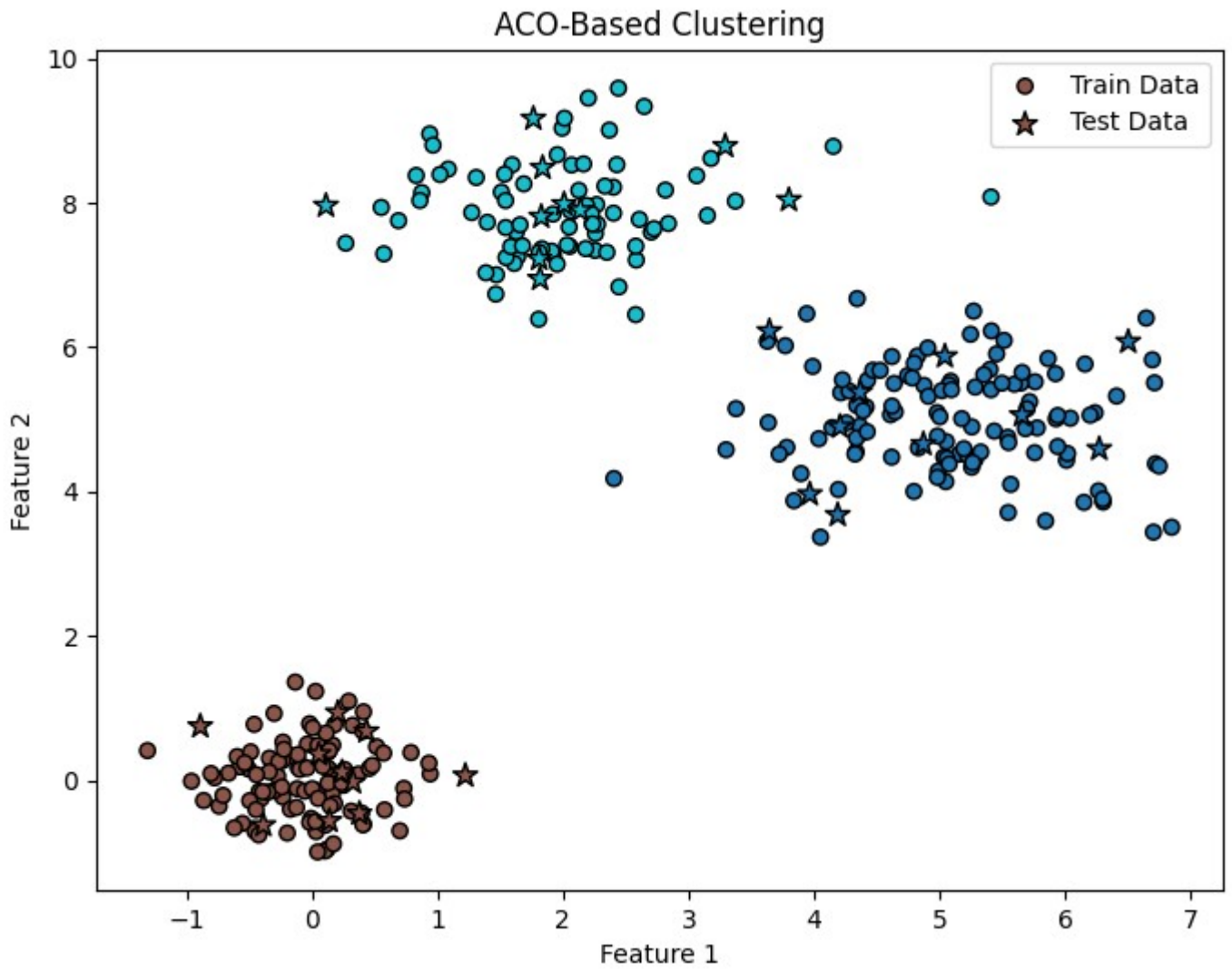
Descripción del Algoritmo

- Basado en el comportamiento de búsqueda de alimento de las hormigas.
- Las hormigas recorren los puntos de datos, reforzando conexiones entre puntos similares.
- Las trayectorias con feromonas más fuertes forman clusters naturales con el tiempo.
- Los clusters finales se extraen mediante Clustering Aglomerativo sobre las feromonas.

Qué Hace Este Código

- Genera datos sintéticos (blobs gaussianos) para clustering.
- Utiliza ACO para agrupar datos, reforzando caminos entre puntos similares.
- Asigna muestras de prueba a clusters según conexiones de feromonas.
- Evalúa la calidad del clustering con puntuación de pureza.
- Grafica los resultados de entrenamiento y prueba con colores correctos de cluster.

```python
class AntColonyClustering:
    """
    Ant Colony Optimization for Clustering.
    Ants reinforce paths between similar points, forming clusters.
    """

    def __init__(
        self,
        distance_matrix,
        num_ants=10,
        alpha=1.0,                  # Importance of pheromone
        beta=2.0,                   # Importance of similarity
        evaporation_rate=0.5,   # Pheromone evaporation
        pheromone_constant=100,# Q in ACO
        generations=50,
        num_clusters=3,
        seed=None
    ):
        if seed is not None:
            random.seed(seed)
            np.random.seed(seed)

        self.distance_matrix = np.array(distance_matrix)
        self.num_nodes = self.distance_matrix.shape[0]
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.Q = pheromone_constant
        self.generations = generations
        self.num_clusters = num_clusters

        # Initialize pheromone trails
        self.pheromones = np.ones((self.num_nodes, self.num_nodes)) * 0.1

        # Heuristic (1/distance), except diagonal = 0
        self.desirability = 1.0 / (self.distance_matrix + 1e-9)
        np.fill_diagonal(self.desirability, 0.0)
         # Will store final cluster assignments
        self.cluster_labels = None
```


ACO-Based Clustering
Train Data
Test Data
Feature 1
Feature 2
10
8
6
4
2
0
−1
0
1
2
3
4
5
6
7

Optimización por Colonia de Abejas (BCO) para Clustering

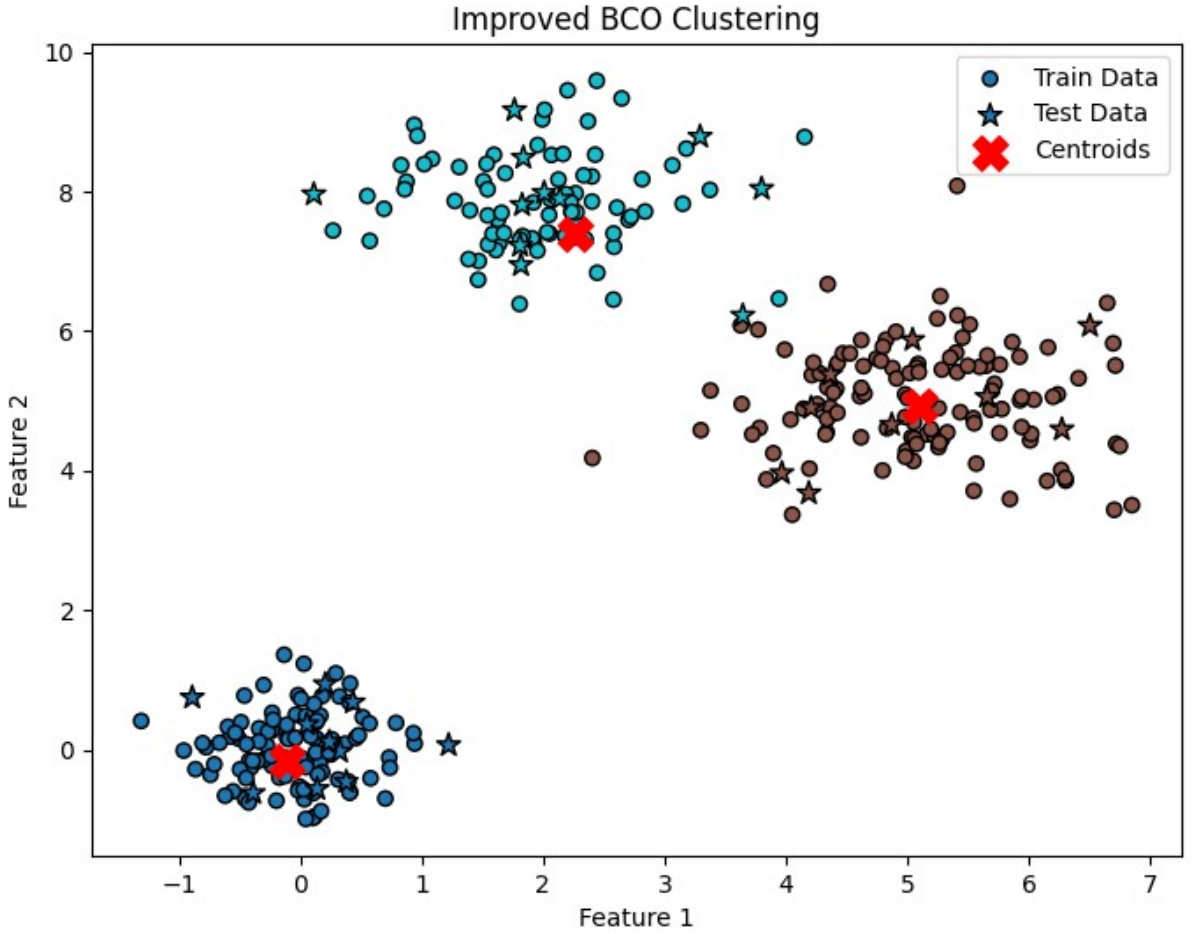
Descripción del Algoritmo

- Inspirado en el comportamiento de búsqueda de alimento de las abejas.
- Abejas obreras refinan los centroides basándose en soluciones cercanas.
- Abejas observadoras seleccionan los mejores centroides según la aptitud.
- Abejas exploradoras buscan soluciones completamente nuevas al azar.

Características Clave

- Utiliza BCO para optimizar los centroides de clusters iterativamente.
- Asigna datos de prueba usando los centroides más cercanos.
- Evalúa la calidad del clustering con puntuación de pureza.
- Grafica los datos de entrenamiento y prueba con asignaciones de cluster.

```python
class BeeColonyClustering:
    """
    Improved Bee Colony Optimization (BCO) for Clustering with:
      - Bounding cluster centers to data range
      - Final K-Means on top solutions to reduce fragmentation
      - Refined local search to converge smoothly
    """

    def __init__(
        self,
        data,
        num_clusters=3,
        num_bees=30,
        num_employed=15,
        num_scouts=5,
        generations=50,
        top_solutions=5,   # number of best solutions to keep for final K-Means
        seed=None
    ):
        if seed is not None:
            np.random.seed(seed)
            random.seed(seed)

        self.data = np.array(data)
        self.num_samples, self.num_features = self.data.shape
        self.num_clusters = num_clusters

        self.num_bees = num_bees
        self.num_employed = num_employed
        self.num_scouts = num_scouts
        self.generations = generations
        self.top_solutions = top_solutions

        # Determine bounding box of data to prevent
        # centroids from drifting away
        self.data_min = self.data.min(axis=0)
        self.data_max = self.data.max(axis=0)

        # Initialize random positions for bee solutions
        #(shape: [num_bees, num_clusters, num_features])
        self.positions = np.random.uniform(
            low=self.data_min, high=self.data_max,
            size=(self.num_bees, self.num_clusters, self.num_features)
        )

        # Track best solution
        self.best_solution = None
        self.best_fitness = np.inf
```



Mejoras Clave en la Optimización de la Colonia de Abejas
Búsqueda Local Refinada

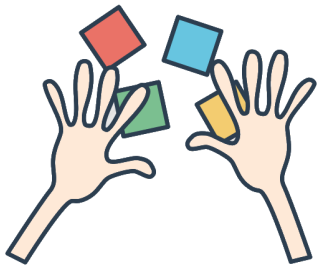
- Las abejas empleadas se mueven hacia la mejor solución en lugar de explorar aleatoriamente.
- Las abejas observadoras eligen entre todas las soluciones con probabilidad exp(-fitness)

Múltiples Soluciones para K-Means Final

- Se recopilan las N mejores soluciones (top_solutions=5) después de la optimización.
- K-Means sobre las mejores soluciones fusiona centroides duplicados y reduce la fragmentación.

Mayor Exploración de Exploradoras

- Las abejas exploradoras buscan cerca de la mejor solución pero con un rango más amplio (±1.0).
- Permite escapar de mínimos locales y mejorar la precisión de los centroides de los clusters.



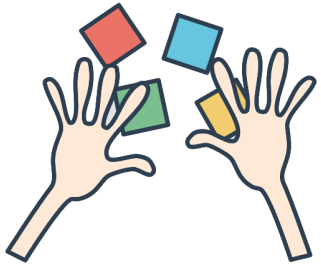
Manos-a-la-obra: #1

Revisión y Ejecución del Código

- En este ejercicio, vas a probar y analizar diferentes algoritmos de clustering:
  - Optimización por Enjambre de Partículas (PSO)
  - Optimización por Colonia de Hormigas (ACO)
  - Optimización por Colonia de Abejas (BCO)
  - Mapa Autoorganizado (SOM)

Qué hacer:

1. Ejecuta cada algoritmo y observa cómo agrupa los datos.
2. Revisa la estructura del código para comprender cómo funciona cada algoritmo.
3. Modifica parámetros clave (por ejemplo, número de clusters, generaciones, tamaño del enjambre/colonia) y analiza su efecto en el rendimiento del clustering.
4. Compara los resultados entre los algoritmos. Identifica cuál funciona mejor y comenta sus ventajas o desventajas.



Manos-a-la-obra: #2

Enunciado: En este ejercicio, seleccionarás dos de los siguientes algoritmos de clustering:

- Optimización por Enjambre de Partículas (PSO)
- Optimización por Colonia de Hormigas (ACO)
- Optimización por Colonia de Abejas (BCO)
- Mapa Autoorganizado de Kohonen (SOM)

Tarea:

1. Elige un conjunto de datos real de tu interés (por ejemplo, un dataset de UCI Machine Learning Repository o Kaggle).
2. Aplica los dos algoritmos seleccionados al dataset y realiza el clustering.
3. Analiza y compara los resultados:
   - ¿Los algoritmos identifican correctamente los grupos esperados?
   - ¿Cuál parece ser más eficiente o robusto?
   - ¿Existen diferencias en la distribución de los clusters?

Objetivos:

- Implementar y ejecutar los algoritmos en un conjunto de datos real.
- Comparar la calidad del clustering entre los dos métodos elegidos.
- Presentar los resultados en un breve informe o diapositivas, incluyendo:
  - Descripción de los datos y su preprocesamiento.
  - Cómo se asignan los puntos a los clusters en cada algoritmo.
  - Comparación visual (gráficos de los clusters).
  - Métrica de evaluación (ej. pureza, silhouette score, etc.).
  - Discusión de ventajas y desventajas de cada enfoque.