Proyecto algoritmo Mapper

Diego Villarreal De La Cerda - 173591
Análisis Topológico de Datos

19 de junio de 2025

1. Introdución

Cuando se inicia un proyecto de análisis de datos, la aproximación más común consiste en aplicar herramientas estadísticas con el fin de extraer información, inferir propiedades relevantes y, a partir de ello, proponer soluciones que resulten razonables para el caso en cuestión. Sin embargo, en numerosas ocasiones, los científicos de datos se enfrentan a conjuntos de datos cuya complejidad —especialmente en contextos de alta dimensionalidad dificulta significativamente su análisis mediante técnicas estadísticas convencionales. La visualización y comprensión de estos datos en espacios de múltiples dimensiones suele ser poco intuitiva o directamente inviable.

Ante esta problemática, surge un enfoque alternativo basado en el estudio de la topología de los datos. Este campo permite explorar la estructura subyacente de los conjuntos de datos más allá de sus características estadísticas superficiales. En particular, el algoritmo Mapper ha demostrado ser una herramienta poderosa para revelar relaciones complejas y patrones que no son evidentes a simple vista. Gracias a su capacidad para proyectar datos de alta dimensión en representaciones más manejables, Mapper facilita la toma de decisiones informadas y robustas en contextos donde otras técnicas fallan.

2. Implementación del Algoritmo

Si bien ya existen librerías implementadas en python que realizan el algoritmo mapper, proponemos una implementación propia de uso libre que se puede encontrar en GitHub o ir a la siguiente URL : `https://github.com/DiegoVilla03/Topological-Data-Analysis`

Dado un conjunto de datos X en $\mathbb{R}^{\times}$ al cual buscamos aplicar el algoritmo, el primer paso para esto es definir una función filtro, usualmente estas funciones suelen proyectar los

datos a dimensiones más baja. Entre las comúnmente utilizadas están el PCA (Principal Components analysis) o funciones conocidas como la norma euclidiana. Seguido de esta transformación se genera una cubierta para los datos y se genera un clustering local, con esos datos para después sobre las intersecciones de la cubierta y los clusters se forme el grafo del complejo simplicial.

```
def __init__(self, lens_func=None, n_intervals=10, overlap=0.1,
                 clusterer=None, nerve_order=2):

def fit(self, X):
        X = np.asarray(X)
        self.data = X
        self.lens_values = self.lens_func(X) if self.lens_func
           else X.mean(axis=1)
        self.cover = self.cover_generator.compute_cover(self.
           lens_values)
        self.graph = self.base.build_graph(X, self.cover)
        self.clusters_map = self.base.clusters_map.copy()
        raw_idx = nx.get_node_attributes(self.graph, 'indices')
        clusters_idx = {nid: set(vals) for nid, vals in raw_idx.
           items()}
        self.nerve = self.nerve_builder.compute_nerve(clusters_idx
           , max_order=self.nerve_order)
        return self
```

El fragmento de código mostrado muestra a grandes rasgos el flujo de acciones que recibe el algoritmo como una API de la libreria, vamos a explicar punto por punto sus componentes y la inicializacion

Para cada instancia del objeto Mapper se cargan los parámetros iniciales de este

```
def radial_lens(x):
    return np.linalg.norm(x,axis = 1)

mapper = Mapper(
    lens_func=radial_lens,
    n_intervals=4,
    overlap=0.3,
    clusterer=DBSCAN(eps=3, min_samples=3),
    nerve_order= 2
)
```

Esto genera la instancia que se va a utilizar para nuestros datos aplicando una función de filtro radial definida como la norma de los datos de X dividiendo en 4 intervalos con

un traslape de 0.3 aplicando un clustering local de DBSCAN y generando un nervio de dimensión máxima 2 o bien un 2-esqueleto

```
    def compute_cover(self, lens_values):
        """
        Retorna lista de (start, end, indices) basada en
           lens_values.
        """
        min_l, max_l = lens_values.min(), lens_values.max()
        length = max_l - min_l
        if length == 0:
            raise ValueError("Valores de lente id nticos; no se
               puede generar cubierta.")

        size = length / (self.n_intervals - (self.n_intervals - 1)
            * self.overlap)
        step = size * (1 - self.overlap)
        cover = []
        start = min_l
        for _ in range(self.n_intervals):
            end = start + size
            inds = np.where((lens_values >= start) & (lens_values
               <= end))[0]
            cover.append((start, end, inds))
            start += step
        return cover
```

Generamos la cubierta sobre los valores de los datos con la función filtro, se añade control de código en caso de que la función filtro haga que los datos, toma la coordenada máxima y mínima del espacio de los datos filtrados y calcula el tamaño de cada división retornando una lista de inicio y fin de coordenadas en forma de intervalos, cubos o hípercubos

```
def build_graph(self, X, cover):
        self.graph.clear()
        self.clusters_map.clear()
        node_counter = 0

        for start, end, inds in cover:
            if len(inds) == 0:
                continue
            subset = X[inds]
            new_nodes, node_counter = self._cluster_interval(
```

```
                subset, inds, node_counter)
            for nid, members in new_nodes:
                self.graph.add_node(nid, indices=members, interval
                    =(start, end))
                self.clusters_map[nid] = set(members)

        # Conectar nodos que compartan puntos
        for id1, set1 in self.clusters_map.items():
            for id2, set2 in self.clusters_map.items():
                if id1 < id2 and set1 & set2:
                    self.graph.add_edge(id1, id2)

        return self.graph
```

El código genera una clusterización local de los datos para después mediante las listas de intervalos generados por la función filtro conecta los nodos donde existan intersecciones , esta grafica unicamente soporta representaciones simpliciales de 0-simplejos, 1-simplejos y 2-simplejos, cualquier otra representación por razones dimensionales no es posible visualizar sin embargo se implementa el metodo de computar nervio

```
def compute_nerve(clusters_indices, max_order=2):
        clusters_sets = {nid: set(inds) for nid, inds in
            clusters_indices.items()}
        nerve = []
        nodes = list(clusters_sets.keys())
        for r in range(2, min(max_order, len(nodes)) + 1):
            for combo in itertools.combinations(nodes, r):
                inter = set.intersection(*(clusters_sets[n] for n
                    in combo))
                if inter:
                    nerve.append(combo)
        return nerve
```

De acuerdo con los índices de los clusters y las conexiones que se establecen entre ellos por la aplicación de el Algoritmo Mapper podemos computar el nervio de el complejo simplicial

Esto, sin embargo; resulta computacionalmente demandante en la medida que el orden del nervio aumenta y la geometria de los datos, aun así nos permite darnos una idea de la estructura que estos tienen al menos de manera numérica que posteriormente puede ser utilizada para análisis de homología simplicial.

Finalmente se crearon algoritmos de visualización para las distintas etapas de la ejecución del algoritmo , para más información visitar el repositorio de GitHub.

3. Aplicación a conjuntos de datos

Aplicaremos el algoritmo a un conjunto de datos de prueba para comprobar su eficacia y la congruencia de los resultados.

3.1. Conglomerados

Utilizando la función makeblobs de la librería sckit-learn generaremos un conjunto de datos de prueba en 2 dimensiones.

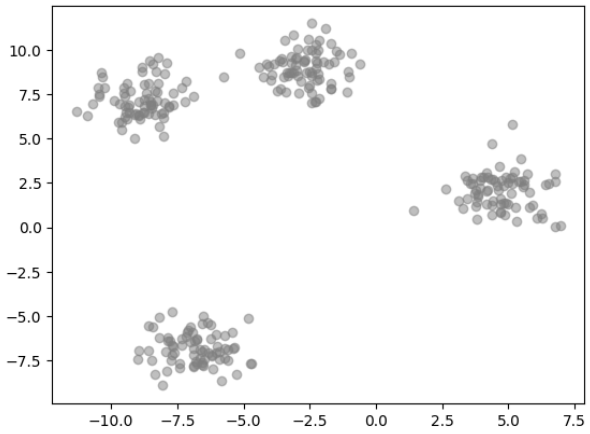

Figura 1: Ejemplo 1: datos conglomerados

en la figura 1 se muestra 4 nubes de puntos en un cuadrado de rango $[-10, 10] \times [-10, 10]$, seleccionamos como función filtro un PCA para reducir la dimensión de los puntos.

```
def lens_pca(X):
    return PCA(n_components=1).fit_transform(X).flatten()
```

De acuerdo con lo visto con la iniciación del algoritmo propuesto

```
mapper = Mapper(lens_func=lens_pca, n_intervals=10, overlap=0.2,
   clusterer=DBSCAN(eps=1.0))
mapper.fit(X)
```

Se separa en 10 intervalos con traslape de 0.2 utilizando un pre-clusterizador DBSCAN

En efecto el algoritmo logra identificar las 4 componentes de los datos originales como se ve en la figura 2 y además nos proporciona el nervio de la gráfica

$$\eta(K) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 01, 23, 45, 57, 68\}$$

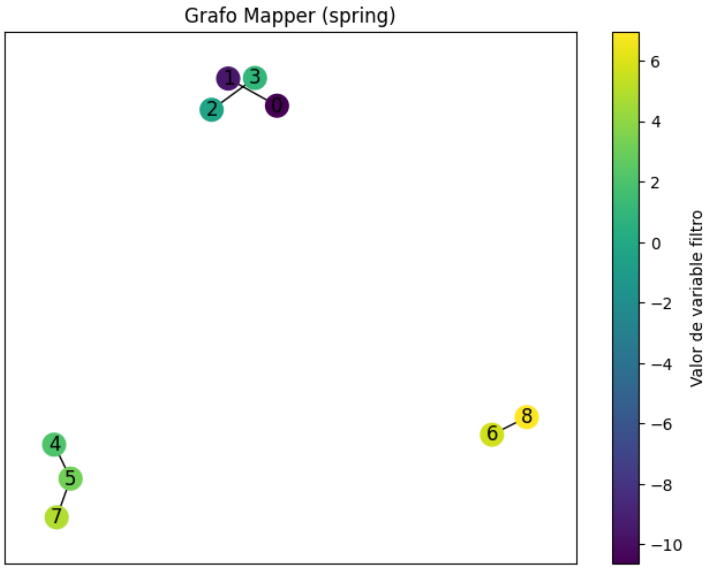

Figura 2: Grafo generado por Mapper Ejemplo 1

3.2. Anillos

Ahora se propone otra geometría de datos para evaluar el comportamiento del algoritmo, nuevamente con una función de Scikit-Learn usamos make-rings, para crear 2 anillos concéntricos de radio 1 y radio 2 (Véase figura 3), definimos 2 funciones filtro a través de la que vamos a analizar los datos.

```
def radial_lens(x):
    return np.linalg.norm(X, axis=1)

def simple_lens(x):
    return x[0:]
```

representado respectivamente las funciones

$$f(x, y) = \|(x, y)\| \qquad g(x, y) = x$$

Aplicamos primero el algoritmo mapper con la función filtro de la norma, es natural pensar que dada la forma de los datos el algoritmo pueda recuperar 2 componentes separadas

```
mapper = Mapper(
    lens_func=radial_lens,
    n_intervals=4,
    overlap=0.3,
    clusterer=DBSCAN(eps=3, min_samples=3),
    nerve_order= 2)
```

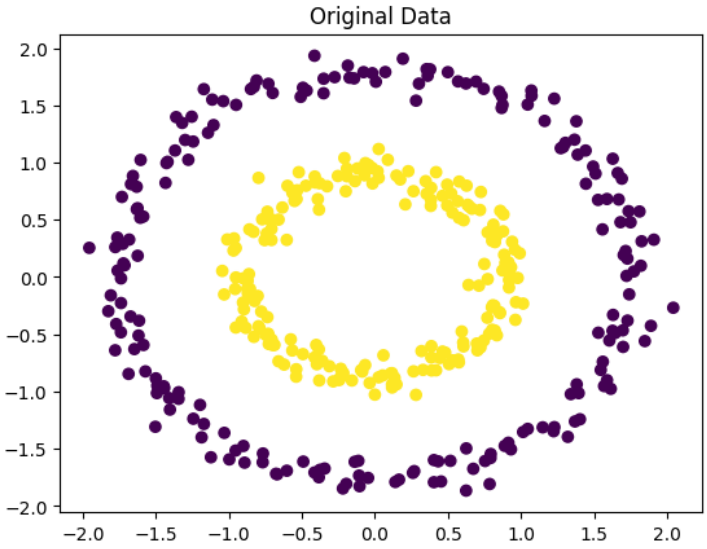

Figura 3: Ejemplo 2 datos en anillos

Sobre el ajuste de los parámetros decidimos dividir en 4 intervalos con traslape de 0.3, limitamos el orden del nervio a generar a lo más 1 simplejos, el grafo resultante son 2 componentes marcando que si hay una separación clara entre los círculos.

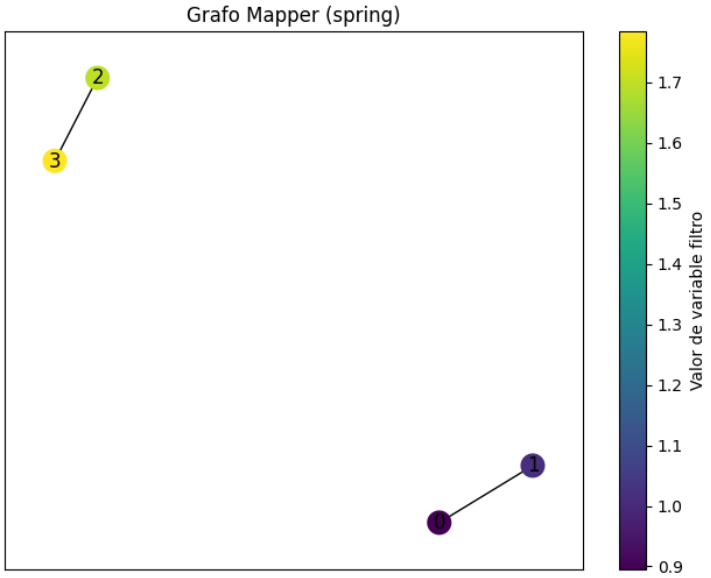

Figura 4: Grafo Mapper con función filtro f(x) para datos en anillos

Ahora con la función que proyecta sobre el eje x observamos un comportamiento distinto, empezando con los parámetros de inicio aquí se busca una cubierta más fina sobre la cual se analizan los datos

```
mapper = Mapper(
    lens_func=simple_lens,
    n_intervals=10,
```

```
    overlap=0.4,
    clusterer=DBSCAN(eps=0.3, min_samples=3),
    nerve_order= 4
```

El grafo resultante es una abstracción de la forma de ambos círculos separados formando 2 componentes de la gráfica, en la figura 5, se ve como uno de los componentes se deforma en una lemniscata sin embargo el nervio de la grafica nos dice que se trata de 2 ciclos.

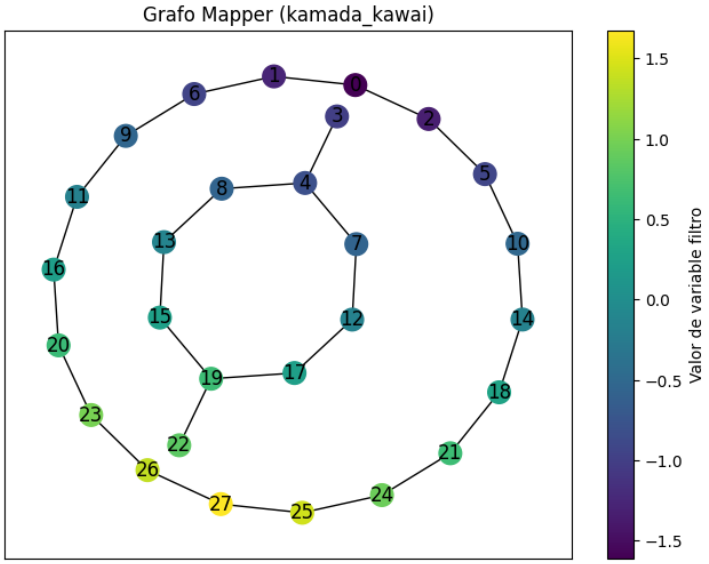

Figura 5: Grafo Mapper con función filtro g(x) para datos en anillos

3.3. Aplicación a conjunto de datos

Para esta prueba se seleccionó el conjunto de datos sobre calidad de vinos disponible en Kaggle. Este dataset describe propiedades como acidez, pH, contenido de azúcar y nivel de alcohol en distintas muestras, con el objetivo de clasificarlas según una medida de calidad o asignarles una calificación.

Como motivación para este análisis, proponemos una aplicación práctica: este tipo de problema puede encontrarse en cadenas de retail o entre distribuidores que buscan evaluar la calidad de sus productos y ofrecer un mejor servicio a sus clientes.

Iniciamos el análisis con la carga de la base de datos, que contiene 13 columnas, una de las cuales corresponde a la calidad asignada. Para efectos de este estudio, se omite dicha columna.

A través de un análisis descriptivo, se decidió eliminar la columna 'Density', ya que aporta poca información debido a su baja variabilidad, así como la columna 'id', que únicamente representa un identificador numérico.

Luego de transformar los datos mediante un proceso de normalización, se obtiene una

base de datos con 10 columnas numéricas. Sin embargo, trabajar con las 10 variables no resulta ideal, por lo que se propone el uso de Análisis de Componentes Principales (PCA) para reducir la dimensión del conjunto de datos.

Al incrementar el número de componentes principales, se observa un aumento en la varianza explicada por el modelo. Como punto de referencia, se consideran 3 y 5 componentes principales.

Con 3 componentes, los datos explican el 61 % de la varianza total; al usar 5 componentes, se alcanza el 80 %. No obstante, las componentes principales 4 y 5, de manera individual, aportan menos del 10 % de la varianza. Por esta razón, se recurre al análisis de la matriz de cargas por variable para interpretar mejor la contribución de cada una.

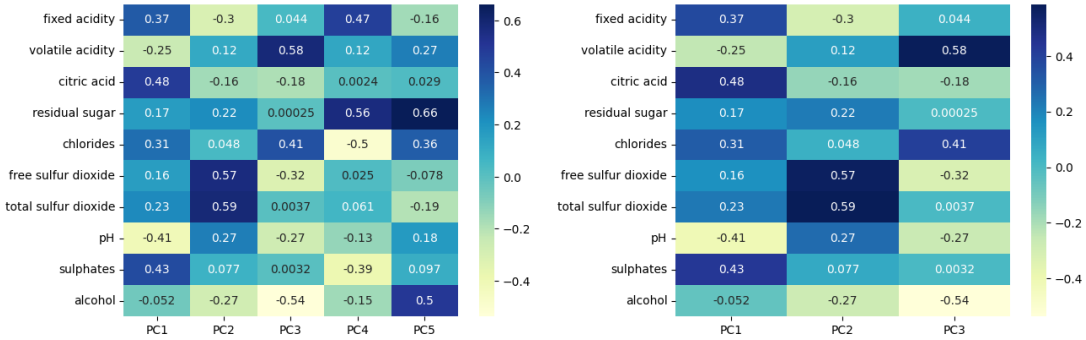

Figura 6: Matrices de carga para PCA con 5 y 3 componentes

En la Figura 6 podemos observar que con cinco componentes principales, cada variable es explicada en su mayoría por una componente específica. Esto sugiere que la transformación conserva buena parte de la estructura original del conjunto de datos. A continuación, analizamos el comportamiento del algoritmo Mapper bajo estos parámetros.

```
mapper = Mapper(
    lens_func=lens,
    n_intervals=12,
    overlap=0.5,
    clusterer=DBSCAN(eps=0.4, min_samples=4),
    nerve_order= 5)
mapper.fit(X_pca)
```

Definimos el conjunto de entrada como los datos transformados mediante PCA, utilizando como función filtro la proyección sobre la primera componente principal, dado que es la que mayor varianza captura. Sin embargo, esta configuración genera únicamente un conglomerado como podemos observar en la figura 7, el cual se manifiesta en forma de un grafo completo. Este grafo no proporciona información estructural significativa, ya que no permite distinguir subestructuras o patrones relevantes dentro de los datos.

Incluso al modificar parámetros clave como el número de intervalos, la cantidad de solapamiento o los criterios de agrupamiento, el resultado no mejora de manera sustancial.

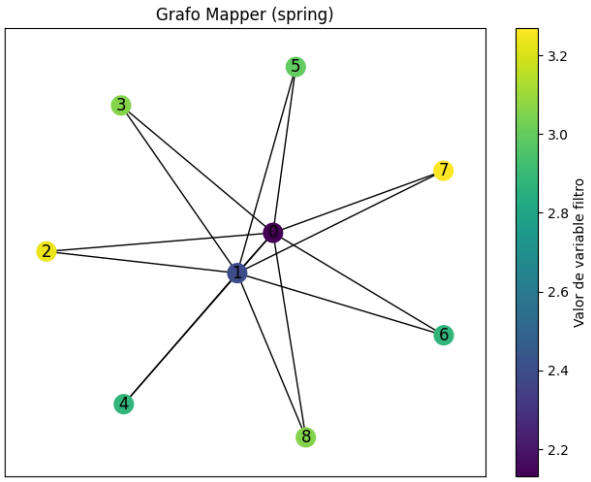

Figura 7: Grafo Mapper de datos con 5 componentes

Esto sugiere que, con esta configuración, ni la función filtro ni los datos de entrada son adecuados para obtener una representación topológica informativa. En algunos casos, el algoritmo Mapper incluso falla en devolver una solución viable, lo que refuerza la idea de que esta elección de parámetros no es óptima.

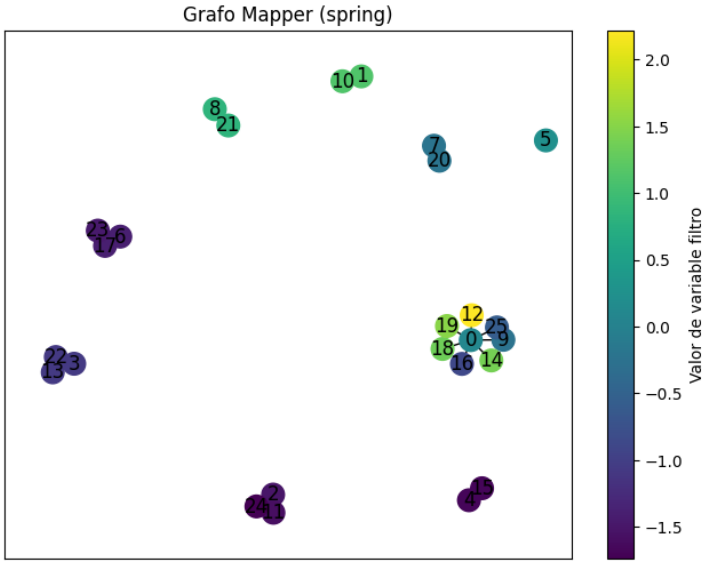

Figura 8: Grafo Mapper con 3 componentes

Cambiando de enfoque, decidimos mantener la misma función filtro la proyección sobre la primera componente principal, pero esta vez utilizando únicamente las tres primeras componentes principales como conjunto de entrada. Esta elección se basa en que dichas componentes explican aproximadamente el 61 % de la varianza total y, además, permiten una representación más compacta y manejable del espacio de características.

Al aplicar Mapper con esta nueva configuración, se observa en la figura 8 un compor-

tamiento diferente y más prometedor. La reducción a tres dimensiones parece preservar estructuras relevantes que se habían diluido al trabajar con cinco componentes.

En el grafo se diferencian entre sí, componentes bien separadas que podrían corresponder a distintas categorías de calidad del vino.

Esto resulta especialmente conveniente, ya que hemos encontrado una representación que parece capturar aspectos significativos de la estructura del conjunto de datos. Al volver al contexto original del problema, este análisis proporciona mayor visibilidad sobre cómo se distribuyen las muestras en el espacio de características transformado.

Sabemos de antemano que el conjunto de datos contempla seis categorías de calidad. En este caso, el algoritmo logra identificar diez agrupaciones bien definidas algunas con valores similares en , lo cual es destacable considerando que la muestra utilizada está fuertemente desbalanceada: dos de las categorías constituyen aproximadamente el 75 % del total. Esta desproporción puede dificultar el trabajo de algoritmos clásicos de clasificación o agrupamiento, especialmente aquellos que dependen de supuestos de homogeneidad o que se basan en criterios de minimización de distancias.

En contraste, la metodología basada en Mapper permite resaltar esta estructura desigual, capturando patrones relevantes que de otro modo podrían quedar ocultos. Esta propiedad refuerza el potencial del enfoque topológico como herramienta complementaria para el análisis exploratorio de datos complejos o desbalanceados.

4. Resultados y Conclusiones

El algoritmo Mapper mostró su utilidad como herramienta exploratoria para revelar estructura en conjuntos de datos, tanto sintéticos como reales. En los ejemplos generados, identificó correctamente patrones conocidos. Pero su valor real se hizo evidente al aplicarlo a los datos de calidad del vino.

Inicialmente, con cinco componentes principales y una función filtro basada en la primera componente, el grafo resultante no ofreció información útil: todos los datos colapsaron en un único clúster. A pesar de ajustar parámetros, el resultado no mejoró. Esto evidenció que no siempre más dimensión implica más información.

La reducción a tres componentes y el uso de la misma función filtro cambió por completo el panorama. El grafo generado reveló cinco agrupaciones bien diferenciadas, que pueden vincularse con categorías de calidad. Aunque el conjunto tiene seis clases conocidas, dos de ellas concentran la mayoría de los ejemplos. Este desbalance probablemente explica por qué solo se detectaron cinco grupos.

Este resultado es clave. Muchos algoritmos clásicos fallan al enfrentarse a datos desbalanceados o no lineales. Mapper, en cambio, expone la estructura de forma natural, sin asumir formas específicas ni depender de distancias.

En suma, Mapper no garantiza siempre una representación clara, pero con una se-

lección cuidadosa de componentes y parámetros, puede ofrecer perspectivas valiosas que otros métodos pasan por alto. Es una herramienta útil cuando lo que se busca no es una respuesta directa, sino entender mejor el espacio donde viven los datos.