1. Aplicar PCA y determinar la cantidad óptima de componentes

PCA - Explicación de su funcionamiento

PCA es una técnica estadística que se utiliza para reducir la dimensionalidad de los datos mientras se retiene la mayor parte de la variabilidad de los mismos. Se basa en una transformación lineal de las variables originales a un nuevo conjunto de variables llamadas **componentes principales**. Estas componentes principales son las direcciones de mayor varianza en los datos.

¿Cómo funciona PCA en álgebra lineal?

1. **Cálculo de la matriz de covarianza**: Se calcula cómo se correlacionan entre sí las diferentes características del dataset.
2. **Autovalores y autovectores**: Se calculan los autovalores y los autovectores de la matriz de covarianza. Los autovectores representan las nuevas direcciones (componentes principales), y los autovalores nos indican la magnitud de la varianza en cada dirección.
3. **Selección de los componentes**: Se seleccionan los componentes principales en función de la cantidad de varianza que explican. El objetivo es retener la mayor varianza posible con el menor número de componentes.

Ejemplo de PCA para la reducción de dimensiones:

Usaremos PCA para reducir las dimensiones del dataset sin considerar la columna objetivo, aplicando los 5 tamaños de componentes que mencionaste: 12, 10, 11, 9, 5, 3.

python

```
from sklearn.decomposition import PCAfrom sklearn.preprocessing
import StandardScalerimport pandas as pdimport numpy as np
# Supongamos que ya hemos cargado el dataset y preprocesado como en
los pasos anteriores# df = pd.read_csv('restaurant_dataset.csv')  #
Cambiar al nombre del archivo cargado
# Preprocesamiento: Rellenamos los valores faltantes y codificamos
las variables categóricas
df.fillna(df.median(), inplace=True)  # Rellenamos los valores nulos
con la mediana
df_encoded = pd.get_dummies(df, drop_first=True)  # Codificamos las
variables categóricas
# Separar las características (X) y la variable objetivo (y)
X = df_encoded.drop('target_column', axis=1)  # Reemplaza
'target_column' con la columna de clase
y = df_encoded['target_column']  # Reemplaza 'target_column' con la
columna de clase
# Escalar los datos
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
# Aplicar PCA para diferentes números de componentesfor n_components
in [12, 10, 11, 9, 5, 3]:
    pca = PCA(n_components=n_components)
    X_pca = pca.fit_transform(X_scaled)
    explained_variance = sum(pca.explained_variance_ratio_)
    print(f"Varianza explicada con {n_components} componentes:
{explained_variance:.4f}")
```

En este código:

- **StandardScaler** se usa para estandarizar las características (es importante porque PCA es sensible a las escalas de las variables).
- Luego se aplica **PCA** para diferentes números de componentes y se imprime la varianza explicada por cada número de componentes.

2. Aprendizaje no supervisado (sin considerar la columna de clase)

En este paso, realizamos un análisis no supervisado. Como no estamos utilizando la columna de clase (es decir, no estamos haciendo clasificación), utilizaremos un algoritmo de **clustering** como K-Means para agrupar los datos en clústeres sin tener en cuenta la variable objetivo.

```
python
from sklearn.cluster import KMeansimport matplotlib.pyplot as plt
# Aplicar KMeans sin considerar la columna de clase (sin 'y')
kmeans = KMeans(n_clusters=3, random_state=42)  # Elegimos 3 clusters
como ejemplo
clusters = kmeans.fit_predict(X_scaled)  # Agrupamos los datos
# Visualización de los clústeres
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters,
cmap='viridis')
plt.title("Clustering de los datos sin la columna 'y'")
plt.xlabel("Componente 1")
plt.ylabel("Componente 2")
plt.show()
# Visualizar los centros de los clusters
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:,
1], marker='X', color='red')
plt.show()
print("Asignaciones de clústeres:", clusters)
```

- Aquí estamos utilizando el algoritmo **K-Means** para dividir los datos en 3 clústeres.
- Después, mostramos la distribución de los clústeres y los centros de los mismos.

3. Resolución del problema de las N-Reinas usando un enfoque de Simulado Recocido

El problema de las N-Reinas consiste en colocar N reinas en un tablero de ajedrez de NxN de manera que no se amenacen entre sí. Esto significa que no pueden compartir la misma fila, columna o diagonal.

Para resolver este problema utilizando **simulado recocido**, el objetivo es encontrar una configuración válida de las reinas en el tablero minimizando el número de ataques. El algoritmo de **Simulated Annealing** es un enfoque probabilístico que se utiliza para encontrar una buena solución a problemas de optimización combinatoria como este.

```
python
import randomimport math
def generar_tablero(N):
    """Genera un tablero de NxN con N reinas al azar"""
    return [random.randint(0, N-1) for _ in range(N)]
def evaluar(tablero):
    """Evalúa cuántas reinas se atacan entre sí"""
    ataques = 0
    N = len(tablero)
    for i in range(N):
        for j in range(i + 1, N):
            if tablero[i] == tablero[j] or abs(tablero[i] -
tablero[j]) == abs(i - j):
                ataques += 1
    return ataques
def simulado_recocido(N, temperatura_inicial=1000, enfriamiento=0.99,
max_iter=1000):
    """Resuelve el problema de las N-reinas usando Simulated
Annealing"""
    tablero_actual = generar_tablero(N)
    ataque_actual = evaluar(tablero_actual)
    mejor_tablero = tablero_actual
    mejor_ataque = ataque_actual
    temperatura = temperatura_inicial

    for _ in range(max_iter):
        if ataque_actual == 0:
            break
        # Generar un vecino (tablero vecino)
        vecino = tablero_actual[:]
        i, j = random.sample(range(N), 2)
        vecino[i] = random.randint(0, N-1)  # Cambiar la posición de
la reina
```

```
        ataque_vecino = evaluar(vecino)

        # Calcular la probabilidad de aceptar el vecino
        if ataque_vecino < ataque_actual or random.random() <
math.exp((ataque_actual - ataque_vecino) / temperatura):
            tablero_actual = vecino
            ataque_actual = ataque_vecino

            # Si el vecino es mejor, actualizar la mejor solución
            if ataque_vecino < mejor_ataque:
                mejor_tablero = vecino
                mejor_ataque = ataque_vecino

        # Reducir la temperatura
        temperatura *= enfriamiento

    return mejor_tablero, mejor_ataque
# Ejemplo de uso:
N = 8  # Tablero de 8x8
solucion, ataques = simulado_recocido(N)print("Tablero solución:",
solucion)print("Número de ataques:", ataques)
```

Este código implementa una solución al problema de las N-Reinas utilizando el algoritmo de Simulated Annealing:

- **Genera un tablero aleatorio** y lo evalúa.
- **Busca una mejor solución** mediante la modificación de las posiciones de las reinas.
- **Acepta nuevas soluciones** con una probabilidad que depende de la diferencia en la evaluación de las soluciones (esto es lo que hace el recocido simulado).

Resumen:

1. **PCA** se aplica para reducir las dimensiones y entender la varianza de los datos.
2. Se realiza un **análisis no supervisado** utilizando **K-Means** para agrupar los datos en clústeres.
3. El problema de **N-Reinas** se resuelve usando el algoritmo de **Simulated Annealing**, que es un enfoque de optimización combinatoria.

Esta combinación de técnicas te permite explorar tu dataset de manera efectiva, reduciendo su dimensionalidad y también aplicando un análisis no supervisado y técnicas de optimización.