Normas

March 3, 2025

1 Normas

1.1 Explicacion Matematica

1.1.1 Antecedente geometrico

Desde la grecia antigua se tenia la distancia euclideana la cual se define como

$$d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}$$

Esto es consistente con el teorema de pitagoras de su momento

1.1.2 Espacios vectoriales

En el siglo XIX se forma la teoria de espacios vectoriales, y se empiezan a estudiar a los vectores como elementos algebraicos, así aparecio la norma euclideana, que mide la longitud de un vector respecto al origen.

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

1.1.3 Casos especificos

Se investigan problemas con restricciones en cuanto a la longitud, como el caso de solo moverse de forma horizontal y vertical, ademas de identificar el valor maximo de los componentes para temas de administración de errores

$$||x||_1 = \sum_{i=1}^{n} ||x_i||$$

$$||x||_\infty = max|x_i|$$

1.1.4 Espacio de funciones

A principios del siglo XX se dieron cuenta que mucha ideas de algebra lineal se podian extender a espacios de funciones por lo que surgio la necesidad de normas mas flexibles para medir el tamaño de funciones. Nace la familia L^p

$$||x_p|| = (\sum_{i=1}^{n} |x_i|^p)^{1/p}$$

1.1.5 Formalización de normas

Para poder tabajar con cualquier espacio vectorial se requiere una formalización.

1- Positividad y definitud positiva

- Se mide el tamaño por lo que no puede ser negativa
- El unico vector que debe tener tamaño 0 debe ser el vector 0

$$||x|| \geq 0$$
$$||x|| = 0 \iff x = 0$$

2- Homogeneidad

- Es natural que si estiras el vector su magnitud se estira igual
- Si duplicas un vector su tamaño debe duplicarse igual

$$||\alpha x = ||\alpha||x||$$

3. Desigualdad triangular

- Que es una traducción de la distancia entre dos puntos directos siempre es menor o igual que ir dando un rode.

$$||x + y|| \leq ||x|| + ||y||$$

2 Ejemplo practico: Proceso matemático de K-means

2.0.1 1. Definición inicial

Dado un conjunto de puntos ($X = x_1, x_2, ..., x_n$) en ($\mathbb{R}^d$), queremos agruparlos en k clusters.
Cada cluster tendrá un **centroide** ($c_j \in \mathbb{R}^d$), que representa el "centro" de ese cluster.

2.0.2 2. Inicialización de centroides

Se eligen k centroides iniciales al azar.
Al inicio, los centroides son simplemente puntos seleccionados de X.

2.0.3 3. Asignación de puntos a clusters

Cada punto x_i se asigna al cluster cuyo centroide está **más cerca**.
La distancia entre el punto x_i y el centroide c_j se calcula con alguna **norma**:

- Euclidiana (L2):

$$d(x_i, c_j) = \sqrt{\sum_{k=1}^{d} (x_{i,k} - c_{j,k})^2}$$

- Manhattan (L1):

$$d(x_i, c_j) = \sum_{k=1}^{d} |x_{i,k} - c_{j,k}|$$

- Infinito (L∞):

$$d(x_i, c_j) = \max_k |x_{i,k} - c_{j,k}|$$

2.0.4 4. Actualización de centroides

Una vez que todos los puntos han sido asignados, se recalcula cada centroide como el **promedio** de todos los puntos asignados a su cluster.

Si el cluster C_j tiene los puntos $(x_{i_1}, x_{i_2}, \dots, x_{i_m})$, el nuevo centroide es:

$$c_j = \frac{1}{m} \sum_{k=1}^{m} x_{i_k}$$

Esto es válido para cualquier norma usada en la asignación, porque la actualización es un promedio aritmético clásico (en espacios Euclideanos).

2.0.5 5. Repetir hasta convergencia

Se repiten los pasos 3 (asignar puntos) y 4 (actualizar centroides) hasta que los centroides no cambien significativamente entre iteraciones, es decir:

$$|c_j^{(t+1)} - c_j^{(t)}| < \epsilon$$

2.0.6 6. Función objetivo (criterio que minimiza)

K-means minimiza la **suma de distancias cuadradas** desde cada punto a su centroide:

$$\sum_{j=1}^{k} \sum_{x_i \in C_j} |x_i - c_j|^2$$

Si cambias la norma, puedes modificar esto (aunque clásicamente K-means solo minimiza en L^2).

3 Ejemplo Practico: Medida del Error en Regresión Lineal (con análisis de normas)

3.0.1 Definir el error (residuo)

Para cada dato:

$$e_i = y_i - \hat{y}_i$$

3.0.2 Medir el tamaño total del error

L^2

$$\|\mathbf{e}\|_2^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Suma los errores al cuadrado. Penaliza mucho los errores grandes.

L^1

$$\|\mathbf{e}\|_1 = \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Suma los errores absolutos. Trata igual a errores pequeños y grandes.

L^∞

$$\|\mathbf{e}\|_\infty = \max_i |y_i - \hat{y}_i|$$

Solo le importa el peor error.

3.0.3 Explicación de robustez

Norma L^2 (MSE)

- **Muy sensible a outliers.**
- Al elevar al cuadrado, un error grande pesa mucho más que varios pequeños:

$$e = (1, 1, 1, 10) \implies \|e\|_2^2 = 1^2 + 1^2 + 1^2 + 10^2 = 103$$

Norma L^1 (MAE)

- **Más robusta a outliers.**
- Al medir el valor absoluto, cada error contribuye igual, sin importar su tamaño:

$$e = (1, 1, 1, 10) \implies \|e\|_1 = 1 + 1 + 1 + 10 = 13$$

Norma L^∞

- **Extremadamente sensible al peor caso.**
- Solo le importa el error máximo:

$$e = (1, 1, 1, 10) \implies \|e\|_\infty = 10$$

4 Codigo K-NN

[16]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cityblock, euclidean, chebyshev
```

[ ]:
```
# --- Parámetros ---
np.random.seed(42)
k = 3
```

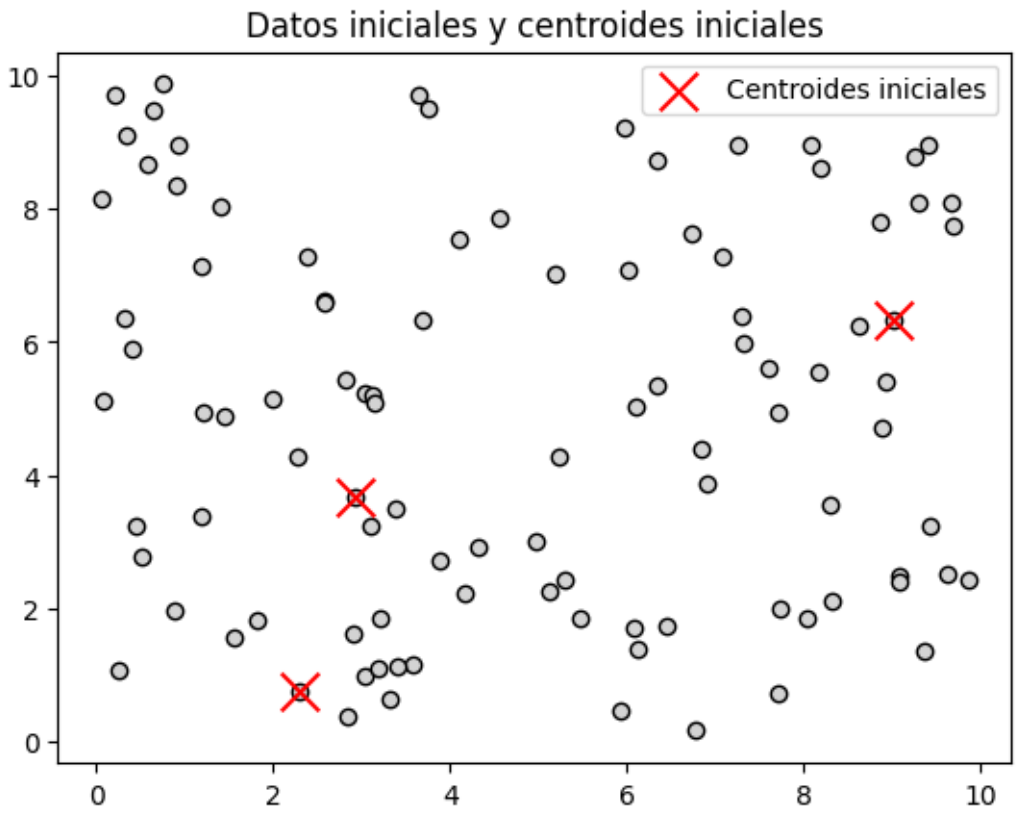
```
n_samples = 100
norma = 'L1'
```

[ ]:
```
# --- Función de distancia
def distancia(p1, p2, norma):
    if norma == 'L1':
        return cityblock(p1, p2)
    elif norma == 'L2':
        return euclidean(p1, p2)
    elif norma == 'Linf':
        return chebyshev(p1, p2)
    else:
        raise ValueError("Norma no válida")
```

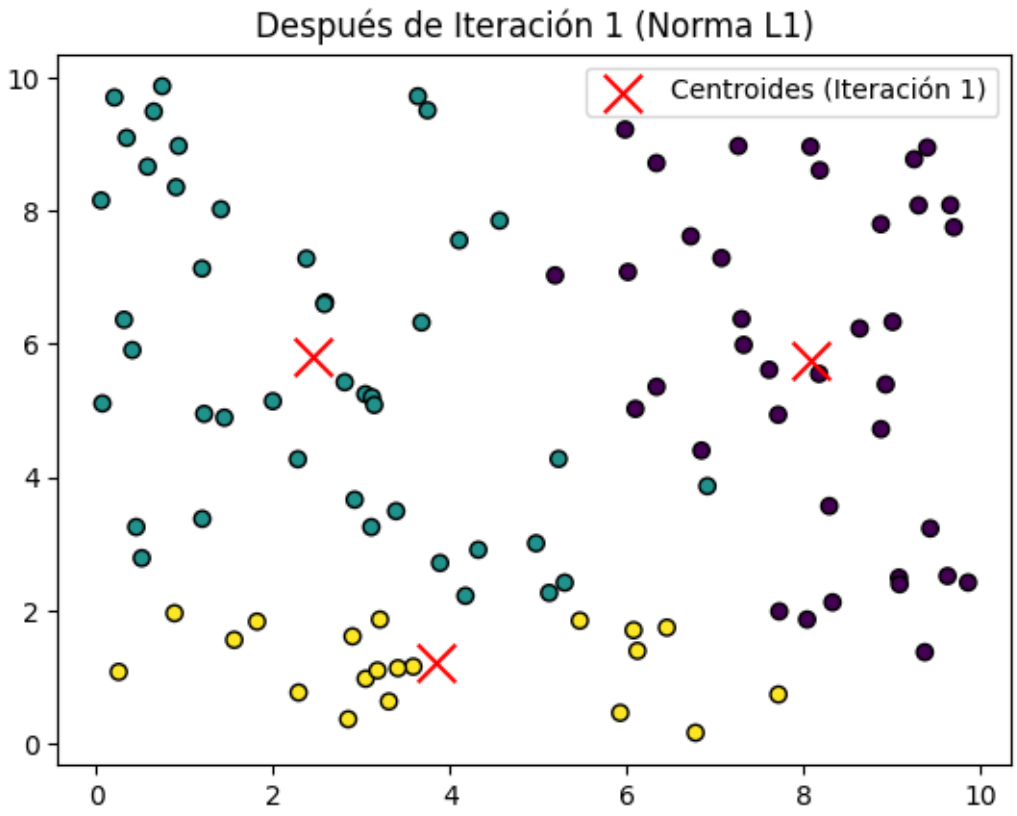
[19]:
```
# --- Datos sintéticos ---
X = np.random.rand(n_samples, 2) * 10
```

[20]:
```
# --- Inicializar centroides al azar ---
centroides = X[np.random.choice(n_samples, k, replace=False)]
```

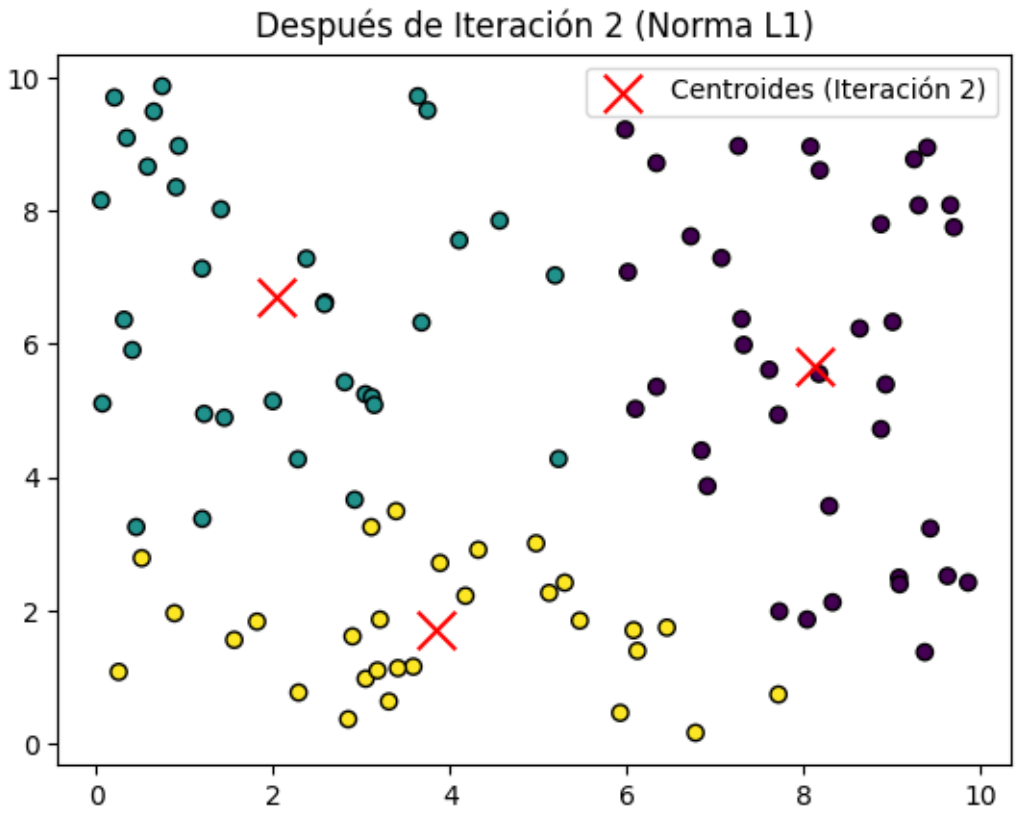
[21]:
```
# --- Visualización inicial ---
plt.scatter(X[:, 0], X[:, 1], c='lightgrey', edgecolor='k')
plt.scatter(centroides[:, 0], centroides[:, 1], c='red', marker='x', s=200,␣
 ↪label='Centroides iniciales')
plt.title("Datos iniciales y centroides iniciales")
plt.legend()
plt.show()
```



[22]:
```
# --- Primera iteración ---
labels = []
for punto in X:
    distancias = [distancia(punto, centroide, norma) for centroide in␣
 ↪centroides]
    labels.append(np.argmin(distancias))
labels = np.array(labels)

nuevos_centroides = np.array([
    X[labels == i].mean(axis=0) if len(X[labels == i]) > 0 else centroides[i]
    for i in range(k)
])

# --- Visualización tras 1ra iteración ---
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', edgecolor='k')
plt.scatter(nuevos_centroides[:, 0], nuevos_centroides[:, 1], c='red',␣
 ↪marker='x', s=200, label='Centroides (Iteración 1)')
plt.title(f"Después de Iteración 1 (Norma {norma})")
plt.legend()
plt.show()
```



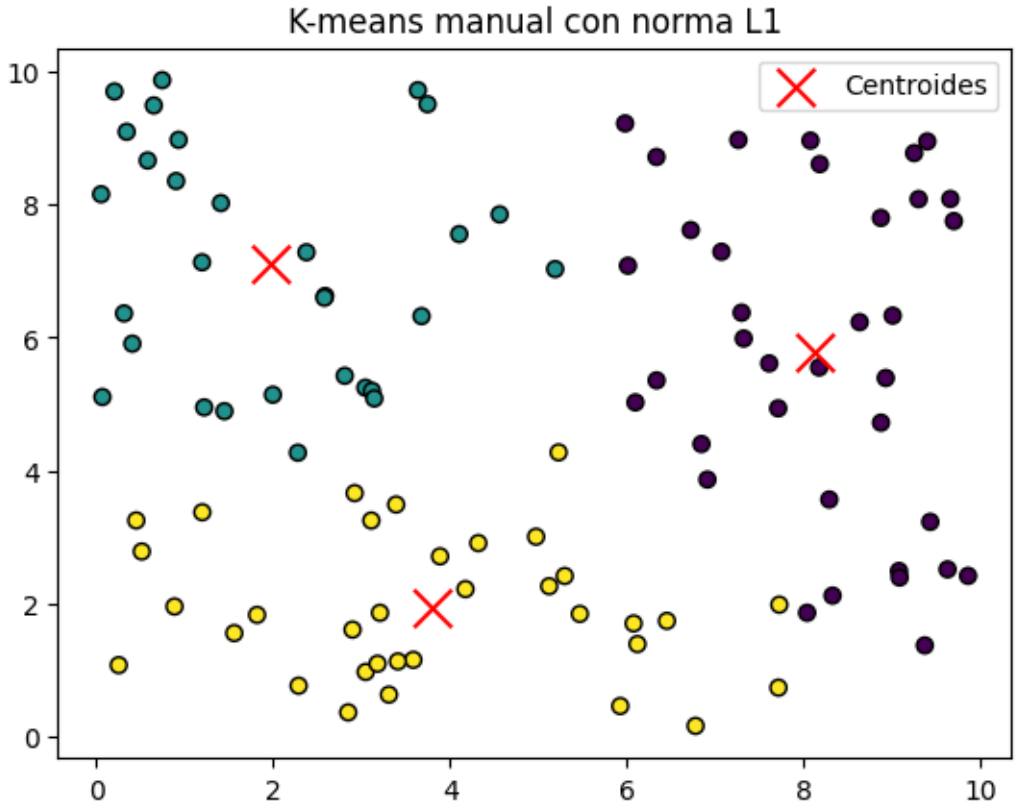
[23]:
```
# --- Segunda iteración ---
centroides = nuevos_centroides
labels = []
for punto in X:
    distancias = [distancia(punto, centroide, norma) for centroide in␣
 ↪centroides]
    labels.append(np.argmin(distancias))
labels = np.array(labels)

nuevos_centroides = np.array([
    X[labels == i].mean(axis=0) if len(X[labels == i]) > 0 else centroides[i]
    for i in range(k)
])

# --- Visualización tras 2da iteración ---
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', edgecolor='k')
plt.scatter(nuevos_centroides[:, 0], nuevos_centroides[:, 1], c='red',␣
 ↪marker='x', s=200, label='Centroides (Iteración 2)')
plt.title(f"Después de Iteración 2 (Norma {norma})")
plt.legend()
plt.show()
```



[24]:
```
# --- K-means manual ---
max_iter = 20

for _ in range(max_iter):
    # Asignar cada punto al centroide más cercano
    labels = []
    for punto in X:
        distancias = [distancia(punto, centroide, norma) for centroide in␣
 ↪centroides]
        labels.append(np.argmin(distancias))
    labels = np.array(labels)

    # Actualizar centroides (promedio de puntos asignados)
    nuevos_centroides = np.array([
        X[labels == i].mean(axis=0) if len(X[labels == i]) > 0 else␣
 ↪centroides[i]
        for i in range(k)
    ])

    # Ver si los centroides ya no cambian (convergencia)
    if np.allclose(centroides, nuevos_centroides, atol=1e-4):
```

```
        break

    centroides = nuevos_centroides
```

[25]:
```
# --- Visualizar resultado ---
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', edgecolor='k')
plt.scatter(centroides[:, 0], centroides[:, 1], c='red', marker='x', s=200,␣
 ↪label='Centroides')
plt.title(f"K-means manual con norma {norma}")
plt.legend()
plt.show()
```



5 Referencias

https://es.statisticseasily.com/glossario/what-is-normed-vector-space/

https://es.wikipedia.org/wiki/Espacio_vectorial_normado

https://es.wikipedia.org/wiki/Historia_de_las_matem%C3%A1ticas

https://es.wikipedia.org/wiki/Norma_vectorial

https://www.aprendemachinelearning.com/k-means-en-python-paso-a-paso/

Productos_internos

March 3, 2025

1 Productos internos

1.1 Explicacion Matematica

Sigo XIX Grassmann y otros formalizan el algebra vectorial y el producto interno como una herramienta par extender la geometria euclidiana a espacios de mas dimensiones. En particular de extender el calculo de angulos y normas

Un **producto interno** en un espacio vectorial V sobre R o C es una funcion

$$\langle *, * \rangle : V$$

Que cumple las siguientes propiedades para todos $u, v, w \in V$ y todo escalar $a \in R$ o C:

1- Linealidad en el primer argumento (o sesquilinealidad en complejos):

$$\langle au + bw, v \rangle = a\langle u, v \rangle, b\langle w, v \rangle$$

2- Simetria (en reales) o conjugada (en complejos):

$$\langle u, b \rangle = \overline{\langle v, u \rangle}$$

3- Positividad definida:

$$\langle u, u \rangle \geq 0$$

$$\langle u, u \rangle \iff u = 0$$

1.1.1 Uso exacto

El proposito de la definicion de los productos internos es poder realizar un analisis de espacios de dimensiones mas grandes, extrapolando las herramientas que ya se tienen en cuanto a longitud, definición de angulos y proyecciones. Como lo son

Normas

$$\|u\| = \sqrt{\langle u, u \rangle}$$

Angulos

$$cos(\theta) = \frac{\langle u, v \rangle}{\|u\|\|v\|}$$

Proyecciones

$$proj_v u = \frac{\langle u, v \rangle}{\langle v, u \rangle}$$

Ortogonalidad

$$\langle u, v \rangle = 0 \implies u \perp v$$

2 Ejemplo practico: Cálculo de Similitud (Cosine Similarity)

2.0.1 Contexto

En muchos problemas de ciencias de datos, necesitamos medir **qué tan parecidos son dos objetos** representados como vectores. Por ejemplo: - Comparar dos documentos en análisis de texto (NLP). - Comparar dos usuarios en un sistema de recomendación. - Comparar perfiles de clientes en segmentación.

La **similaridad coseno** mide el ángulo entre dos vectores en un espacio de características. Se calcula directamente con el producto interno:

$$\cos(\theta) = \frac{\langle u, v \rangle}{\|u\|\|v\|}$$

2.0.2 Interpretación

- $\cos(\theta) = 1$ vectores idénticos (máxima similitud).
- $\cos(\theta) = 0$ vectores ortogonales (sin relación).
- $\cos(\theta) = -1$ vectores opuestos (máxima disimilitud).

2.0.3 Ejemplo

Supongamos que tienes dos vectores que representan perfiles de usuario:

$$u = (1, 2, 3), \quad v = (3, 2, 1)$$

Producto interno:

$$\langle u, v \rangle = 1 \cdot 3 + 2 \cdot 2 + 3 \cdot 1 = 10$$

Norma de

$$u$$

:

$$\|u\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$$

Norma de

$$v$$

:

$$\|v\| = \sqrt{3^2 + 2^2 + 1^2} = \sqrt{14}$$

Similaridad coseno:

$$\cos(\theta) = \frac{10}{\sqrt{14} \cdot \sqrt{14}} = \frac{10}{14} \approx 0.714$$

Esto indica que los vectores son bastante similares.

3 Ejemplo practico: Kernel Trick (en SVM y Kernel PCA)

3.0.1 Contexto

En problemas de clasificación o reducción de dimensionalidad, a veces los datos no se pueden separar o analizar bien en el espacio original. Una solución es **mapearlos a un espacio de mayor dimensión** donde sí sean separables. Sin embargo, calcular explícitamente esa transformación puede ser costoso.

3.0.2 La idea clave

Con el **kernel trick**, no calculamos la transformación completa. En vez de eso, definimos un **producto interno modificado** que trabaja directamente en el espacio transformado:

$$K(x, y) = \langle \phi(x), \phi(y) \rangle$$

Aquí,

$$\phi$$

es la función que transforma los datos, pero nunca necesitamos calcular

$$\phi(x)$$

directamente. Solo definimos el kernel

$$K$$

de forma que actúe como ese producto interno.

3.0.3 Ejemplos de kernels comunes

- **Lineal (espacio original):**

$$K(x, y) = x^T y$$

- **Polinomial de grado**

$$d$$

:

$$K(x, y) = (x^T y + c)^d$$

- **RBF (Radial Basis Function):**

$$K(x, y) = \exp\left(-\gamma \|x - y\|^2\right)$$

3.0.4 Por qué funciona

En modelos como SVM o Kernel PCA, los cálculos clave dependen solo de productos internos entre puntos de datos. Si reemplazamos esos productos internos por un kernel, el algoritmo trabaja como si estuviera en un espacio transformado, sin calcular la transformación real.

3.0.5 Ejemplo

Supongamos que tienes dos puntos en

$$\mathbb{R}^2$$

:

$$x = (1, 2), \quad y = (3, 4)$$

Si usamos el kernel polinomial de grado 2:

$$K(x, y) = (x^T y + 1)^2$$

Producto interno:

$$x^T y = 1 \cdot 3 + 2 \cdot 4 = 11$$

Aplicamos el kernel:

$$K(x, y) = (11 + 1)^2 = 144$$

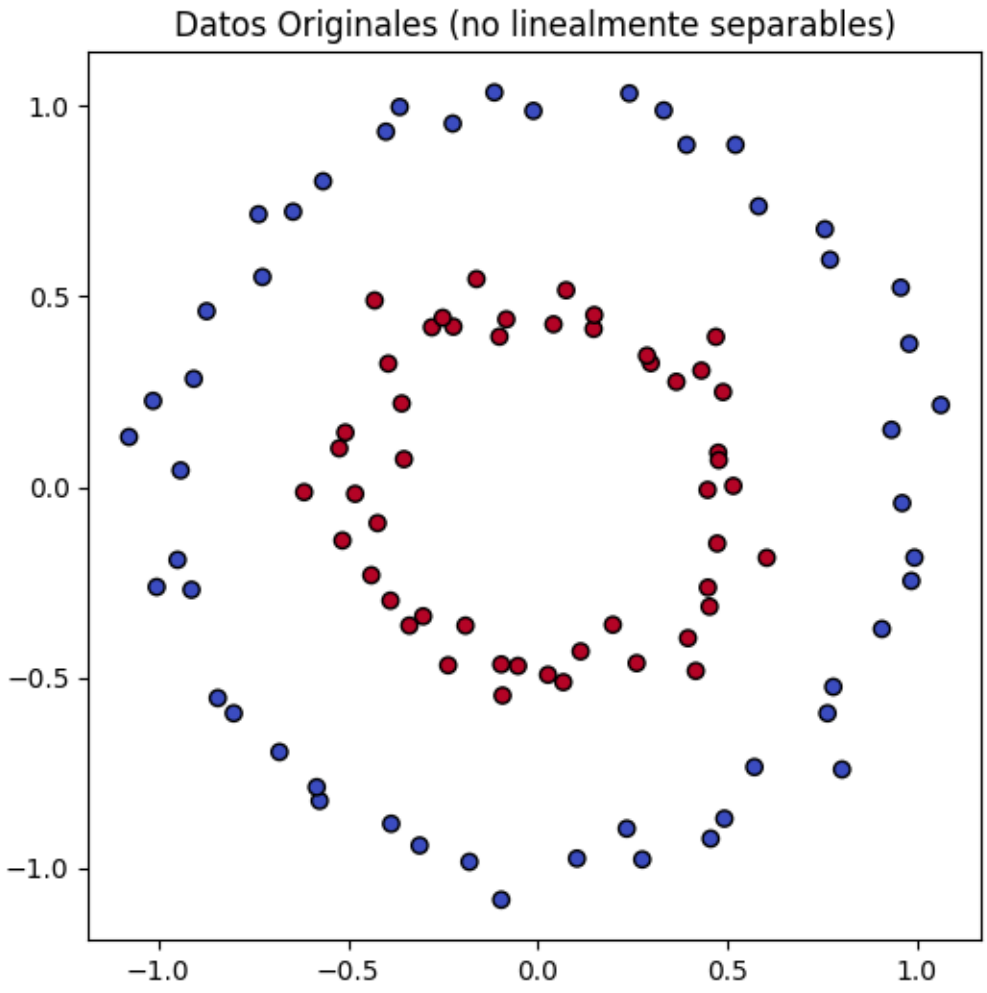
Con esto, el algoritmo está operando como si estuviera en un espacio curvado de mayor dimensión, pero sin calcular directamente esas coordenadas.

4 Codigo

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.datasets import make_circles
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler



# Función para graficar frontera de decisión
def plot_decision_boundary(model, X, y, title):
    h = 0.02
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
```

```
    plt.figure(figsize=(6, 6))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k')
    plt.title(title)
    plt.show()
```

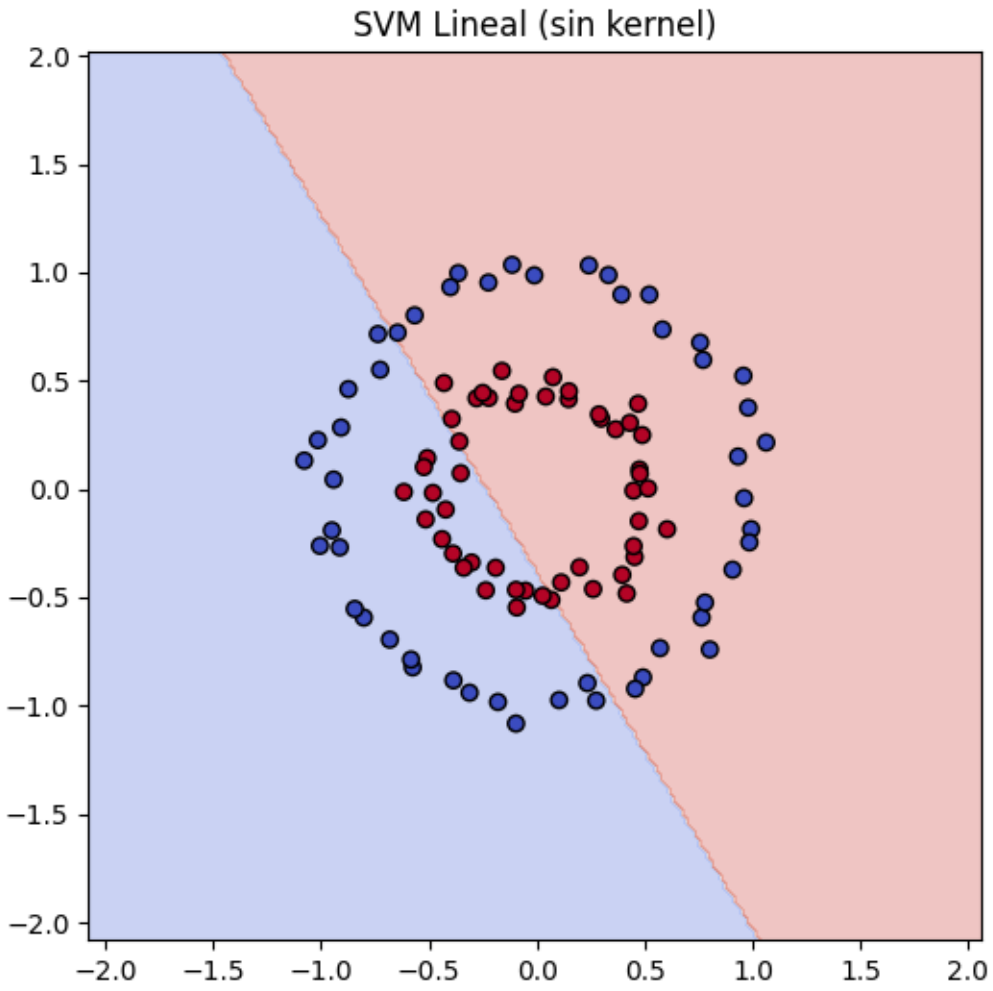
[2]:
```
# Crear datos no linealmente separables (círculos concéntricos)
X, y = make_circles(n_samples=100, noise=0.05, factor=0.5, random_state=42)

# Visualizar los datos
plt.figure(figsize=(6, 6))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k')
plt.title('Datos Originales (no linealmente separables)')
plt.show()
```



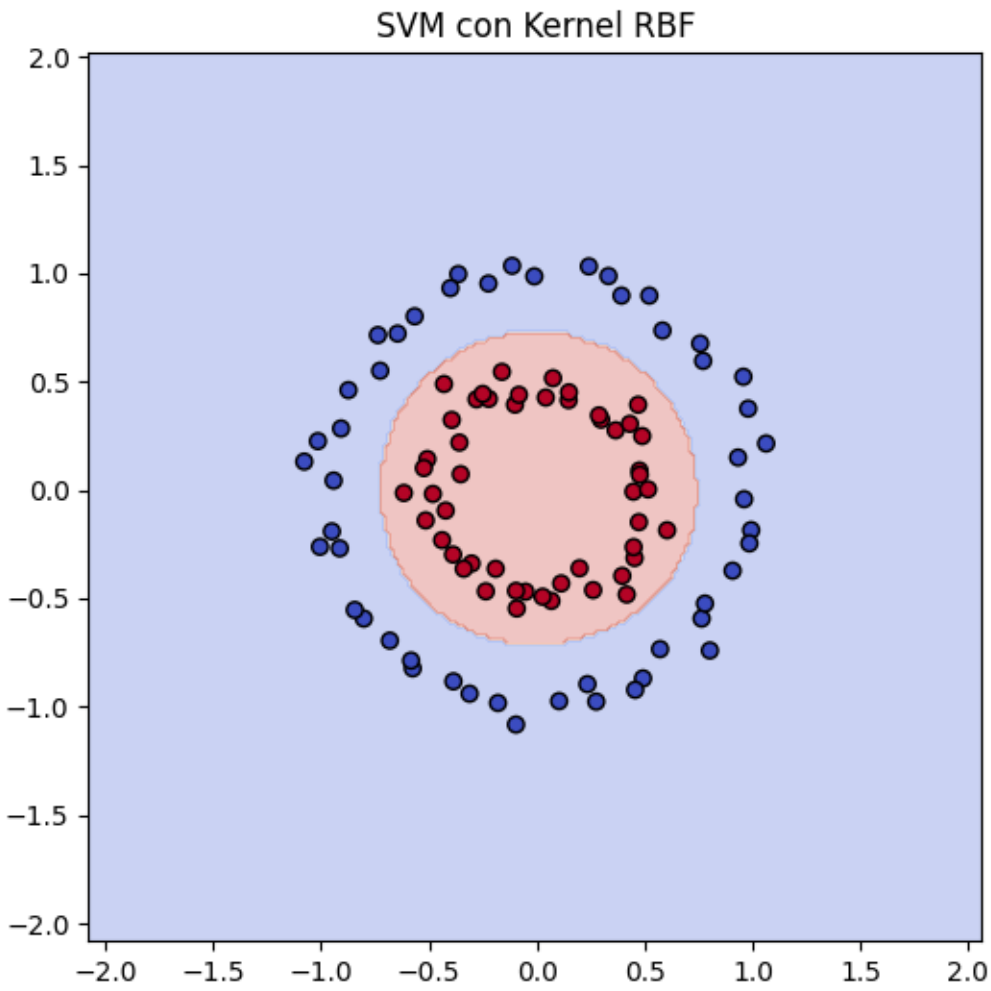
[3]:
```
# Entrenar SVM sin kernel (lineal)
svm_lineal = make_pipeline(StandardScaler(), SVC(kernel='linear'))
svm_lineal.fit(X, y)

# Entrenar SVM con kernel RBF (espacio curvado)
svm_rbf = make_pipeline(StandardScaler(), SVC(kernel='rbf', gamma='auto'))
svm_rbf.fit(X, y)
```

[3]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('svc', SVC(gamma='auto'))])
```

[6]:
```
# Mostrar frontera de decisión lineal
plot_decision_boundary(svm_lineal, X, y, 'SVM Lineal (sin kernel)')
```

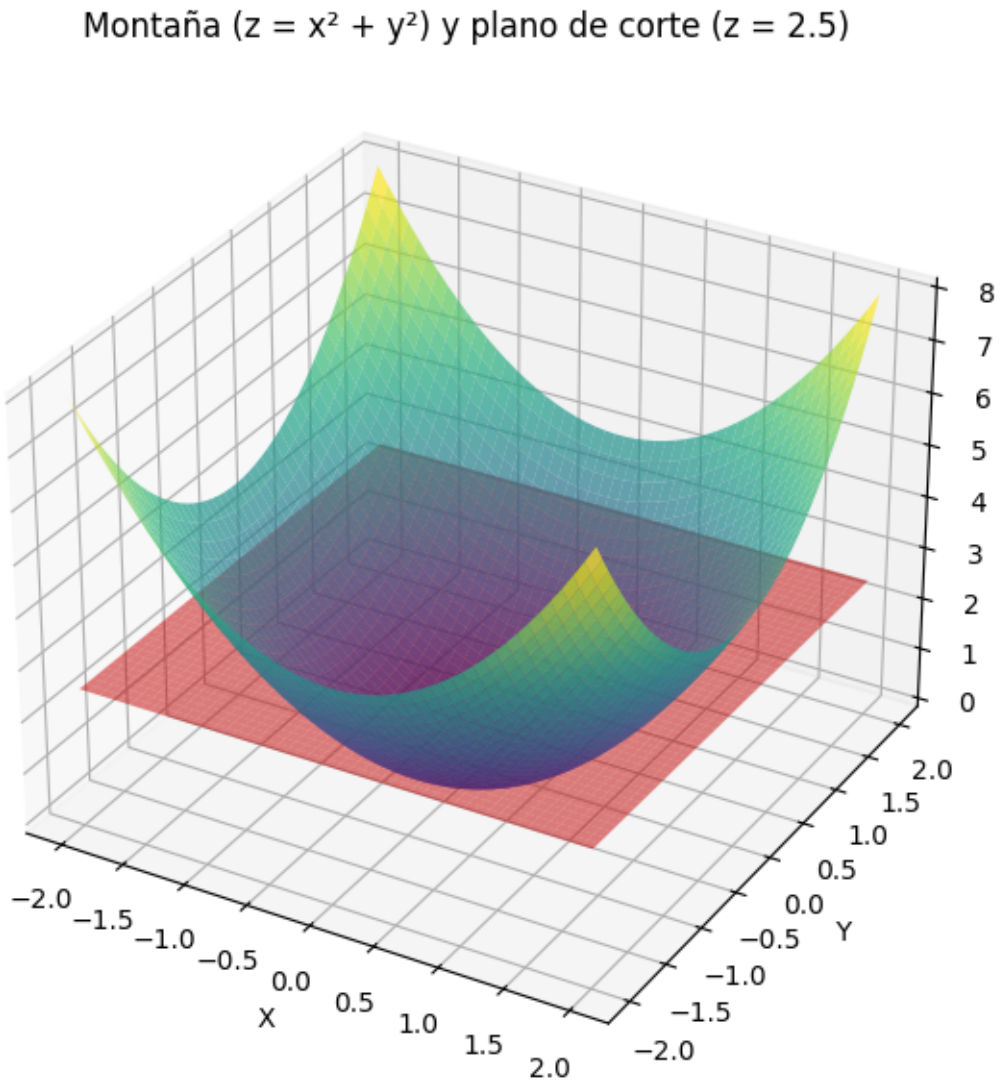


[7]:
```
# Mostrar frontera de decisión con kernel RBF
plot_decision_boundary(svm_rbf, X, y, 'SVM con Kernel RBF')
```



[8]:
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Crear una cuadrícula de puntos en 2D (x, y)
x = np.linspace(-2, 2, 50)
y = np.linspace(-2, 2, 50)
X, Y = np.meshgrid(x, y)

# Definir la montaña: z = x² + y²
Z = X**2 + Y**2

# Crear la figura 3D
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
```

```python
# Graficar la montaña
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.7)

# Definir un plano horizontal de corte (z = 2.5)
Z_plane = np.full_like(Z, 2.5)

# Graficar el plano de corte
ax.plot_surface(X, Y, Z_plane, color='red', alpha=0.5)

# Etiquetas y título
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z = x² + y²')
ax.set_title('Montaña (z = x² + y²) y plano de corte (z = 2.5)')

# Mostrar la gráfica
plt.show()
```



5 Enlaces

https://mathworld.wolfram.com/InnerProduct.html

https://www.britannica.com/science/inner-product

https://mathshistory.st-andrews.ac.uk/Biographies/Grassmann/

https://machinelearningmastery.com/support-vector-machines-for-machine-learning/

https://scikit-learn.org/stable/modules/svm.html

Longitudes y distancias

March 4, 2025

1 Longitudes y distancias

1.1 Explicación Matemática

1.1.1 Distancias

Las **métricas de distancia** permiten cuantificar la separación entre dos puntos en un espacio. A continuación, se presentan las principales:

Distancia Euclidiana (L2) La distancia euclidiana mide la longitud de la línea recta entre dos puntos. Para dos puntos $A = (x_1, x_2, \dots, x_n)$ y $B = (y_1, y_2, \dots, y_n)$ se define como:

$$d_{\text{Euclid}}(A, B) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Esta métrica corresponde a la norma L2 del vector diferencia $A - B$.

Distancia de Manhattan (L1) La distancia de Manhattan, también conocida como distancia "taxicab", se basa en la suma de las diferencias absolutas de las coordenadas:

$$d_{\text{Manhattan}}(A, B) = \sum_{i=1}^{n} |x_i - y_i|$$

Es útil en entornos en los que el movimiento se restringe a ejes ortogonales.

Distancia de Minkowski (Lp) La distancia de Minkowski es una generalización de las anteriores a través del parámetro p:

$$d_{\text{Minkowski},p}(A, B) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

Para $p = 1$ se obtiene la Manhattan y para $p = 2$ la Euclidiana. En el límite cuando $p \to \infty$, se obtiene la **distancia de Chebyshev**:

$$d_{\infty}(A, B) = \max_i |x_i - y_i|$$

Otras Métricas

- **Distancia Coseno:**
  Basada en el ángulo entre dos vectores, se define a partir de la similitud coseno:

$$\text{similitud\_cos}(A, B) = \frac{A \cdot B}{\|A\| \, \|B\|}, \quad d_{\cos}(A, B) = 1 - \text{similitud\_cos}(A, B)$$

- **Distancia de Hamming:**
  Para secuencias o vectores discretos, cuenta el número de posiciones en que difieren:

$$d_{\text{Hamming}}(s, t) = \#\{i \mid s_i \neq t_i\}$$

- **Distancia de Jaccard:**
  Mide la disimilitud entre conjuntos:

$$d_{\text{Jaccard}}(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

- **Distancia de Mahalanobis:**
  Considera la estructura de covarianza de los datos:

$$d_{\text{Mahalanobis}}(x, y) = \sqrt{(x - y)^T \Sigma^{-1} (x - y)}$$

1.1.2 Longitudes

El concepto de **longitud** en geometría se refiere a la medida de la extensión a lo largo de una dimensión.

Longitud de un segmento Para un segmento de línea recta, la longitud es la distancia euclidiana entre sus dos extremos:

$$L = d_{\text{Euclid}}(A, B)$$

Longitud de una curva Para una curva parametrizada $\gamma : [a, b] \to \mathbb{R}^n$, la longitud (o longitud de arco) se define mediante la integral:

$$L(\gamma) = \int_a^b \|\gamma'(t)\| \, dt$$

donde $\gamma'(t)$ es la derivada de la función parametrizada y $| '(t)$ su norma.

1.2 Aplicaciones en Ciencias de Datos

Las métricas de distancia y el cálculo de longitudes tienen un amplio espectro de aplicaciones en ciencia de datos, desde técnicas de clasificación y clustering hasta análisis de trayectorias, procesamiento de imágenes y detección de anomalías. A continuación se detallan diversas aplicaciones junto con ejemplos y fragmentos de código en Python.

1.2.1 1. Clasificación y Clustering

Clasificación con k-Nearest Neighbors (k-NN) En el algoritmo **k-NN** se utiliza una métrica de distancia para identificar los vecinos más cercanos a un nuevo punto y asignar la clase predominante. Dependiendo de la naturaleza de los datos, se pueden usar la distancia Euclidiana, Manhattan o incluso la de Minkowski.

[9]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Cargamos el conjunto de datos Iris
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

# Usamos k-NN con la distancia Euclidiana (Minkowski p=2)
knn_euclid = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2)
knn_euclid.fit(X_train, y_train)
print("Precisión con distancia Euclidiana:", knn_euclid.score(X_test, y_test))

# Usamos k-NN con la distancia Manhattan (Minkowski p=1)
knn_manhattan = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=1)
knn_manhattan.fit(X_train, y_train)
print("Precisión con distancia Manhattan:", knn_manhattan.score(X_test, y_test))
```

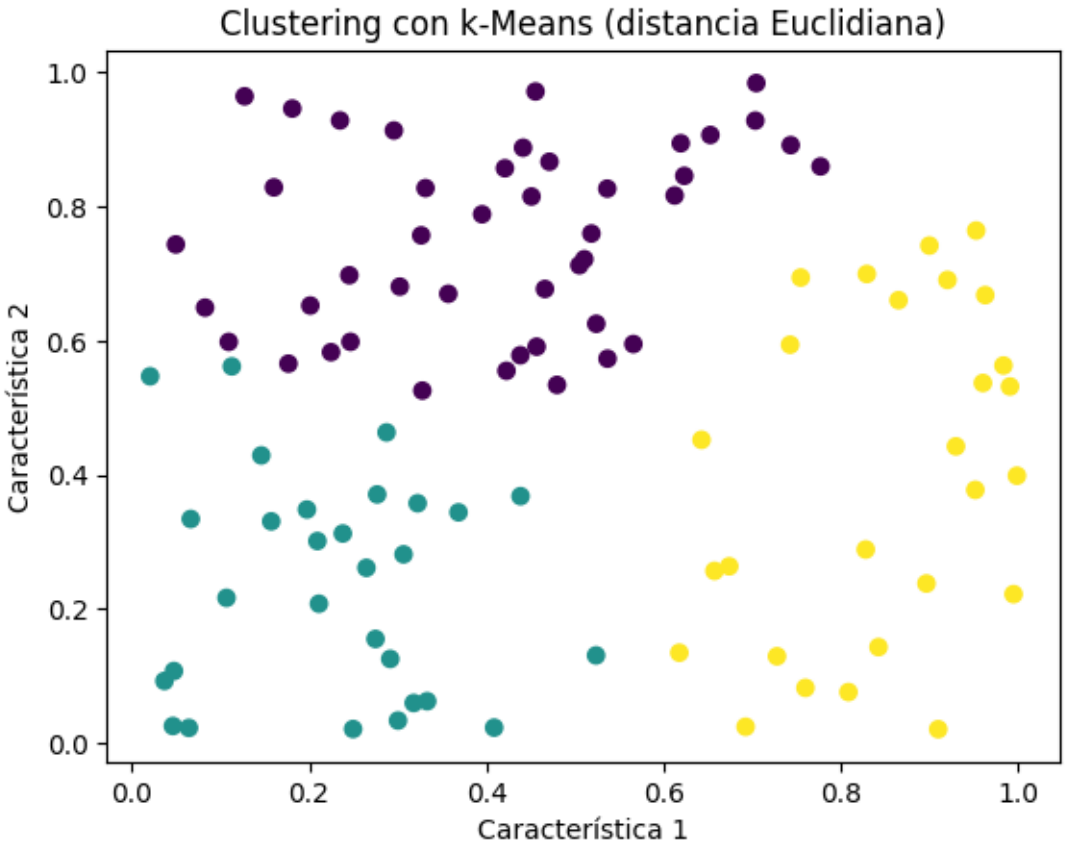
```
Precisión con distancia Euclidiana: 1.0
Precisión con distancia Manhattan: 1.0
```

Clustering con k-Means El algoritmo **k-Means** busca agrupar puntos minimizando la suma de las distancias euclidianas entre cada punto y el centroide de su clúster.

Ejemplo:

[10]:
```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np

# Supongamos un conjunto de datos bidimensional
X = np.random.rand(100, 2)

# Aplicamos k-Means para 3 clusters
```

```
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=clusters, cmap='viridis')
plt.title("Clustering con k-Means (distancia Euclidiana)")
plt.xlabel("Característica 1")
plt.ylabel("Característica 2")
plt.show()
```



1.2.2 2. Regresión

Regresión Lineal (Error Cuadrático) La **regresión lineal** se entrena minimizando la suma de errores al cuadrado, lo cual es equivalente a minimizar la distancia Euclidiana entre los valores predichos y reales.

Ejemplo:

[11]:
```
from sklearn.linear_model import LinearRegression
import numpy as np
```

```python
# Datos de ejemplo
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 5, 4, 5])

# Ajustamos un modelo de regresión lineal
model = LinearRegression()
model.fit(X, y)
y_pred = model.predict(X)

# Cálculo del error cuadrático medio (MSE)
mse = np.mean((y - y_pred) ** 2)
print("Error cuadrático medio:", mse)
```

```
Error cuadrático medio: 0.47999999999999987
```

Regresión LAD y Regularización L1 (Lasso) La **regresión LAD (Least Absolute Deviations)** minimiza la suma de errores absolutos (distancia L1) y es más robusta frente a outliers. De manera similar, la regularización L1 (Lasso) penaliza la suma de los valores absolutos de los coeficientes para obtener modelos más parsimoniosos.

Ejemplo con Lasso:

[12]:
```python
from sklearn.linear_model import Lasso

# Ajustamos un modelo Lasso
lasso = Lasso(alpha=0.1)
lasso.fit(X, y)
print("Coeficientes del modelo Lasso:", lasso.coef_)
```

```
Coeficientes del modelo Lasso: [0.55]
```

1.2.3 3. Análisis de Trayectorias y Series Temporales

El cálculo de la **longitud de una curva** es fundamental en el análisis de trayectorias, ya que permite cuantificar la "distancia recorrida" a lo largo de una trayectoria.

Ejemplo: Calcular la longitud de la curva definida por $y = \sin(x)$ en el intervalo $[0, \pi]$:

[13]:
```python
import numpy as np
from scipy.integrate import quad

def integrand(x):
    return np.sqrt(1 + np.cos(x)**2)

longitud, error = quad(integrand, 0, np.pi)
print(f"Longitud de la curva y = sin(x) en [0, ]: {longitud:.4f}")
```

```
Longitud de la curva y = sin(x) en [0, ]: 3.8202
```

1.3 Resumen

Las diversas métricas de distancia (Euclidiana, Manhattan, Minkowski, Coseno, Hamming, Jaccard y Mahalanobis) permiten modelar la "cercanía" entre puntos, secuencias o conjuntos en espacios de datos de distintas naturalezas. Asimismo, el cálculo de longitudes (ya sea de segmentos rectos o de curvas) es fundamental para analizar trayectorias, contornos y formas.

Estas herramientas matemáticas se traducen en aplicaciones prácticas en clasificación, clustering, regresión, análisis de trayectorias, procesamiento de imágenes, análisis geoespacial, sistemas de recomendación y detección de anomalías, entre otros. La implementación de estos métodos en Python, utilizando bibliotecas como **scikit-learn**, **SciPy**, **NumPy** y **OpenCV**, permite adaptar y experimentar con técnicas para mejorar la eficiencia y precisión de los modelos en proyectos de ciencia de datos.

Angulos y ortogonalidad

March 4, 2025

1 Ángulos y Ortogonalidad

En espacios vectoriales generales, los conceptos de ángulo y ortogonalidad se definen utilizando un **producto interno** que generaliza la noción clásica del producto punto en $\mathbb{R}^n$. Estos conceptos son esenciales en muchas áreas de la matemática y tienen aplicaciones directas en la ciencia de datos, tales como reducción de dimensionalidad, análisis de similitud, compresión y preprocesamiento de datos.

1.1 Explicación Matemática

1.1.1 Definición del Ángulo

Para dos vectores no nulos u y v en un espacio vectorial con producto interno $\langle \cdot, \cdot \rangle$, el **ángulo** θ entre ellos se define mediante:

$$\theta = \arccos\left(\frac{\langle u, v \rangle}{\|u\|\|v\|}\right)$$

donde la **norma** (o longitud) de un vector (u) se define como:

$$\|u\| = \sqrt{\langle u, u \rangle}$$

1.1.2 Ortogonalidad

Dos vectores u y v son **ortogonales** (o perpendiculares) si su producto interno es cero:

$$\langle u, v \rangle = 0$$

Esta propiedad implica que el ángulo entre u y v es de $\frac{\pi}{2}$ (90°). La ortogonalidad es crucial en la construcción de bases ortonormales y en procesos de descomposición, como el de Gram–Schmidt, que permite transformar un conjunto de vectores linealmente independientes en un conjunto ortogonal (o incluso ortonormal).

1.1.3 Propiedades Clave

- **Generalización del producto punto:** El producto interno generaliza la noción del producto punto, lo que permite definir ángulos y longitudes en espacios de dimensión infinita o en espacios abstractos.

- **Base Ortonormal:** Una base es ortonormal si sus vectores son unitarios (norma 1) y mutuamente ortogonales. Esto simplifica muchos cálculos, ya que se tiene que para una base $e_1, e_2, \ldots, e_n$:

$$\langle e_i, e_j \rangle = \delta_{ij}$$

  donde δ_{ij} es el delta de Kronecker (1 si $i = j$, 0 en caso contrario).

- **Descomposición:** Técnicas como la descomposición en valores singulares (SVD) y el análisis de componentes principales (PCA) se basan en la ortogonalidad de los vectores para separar la información en componentes independientes.

1.2 Aplicaciones en Ciencias de Datos

Los conceptos de ángulo y ortogonalidad tienen numerosas aplicaciones en ciencia de datos. A continuación se presentan algunas de las más relevantes junto con ejemplos prácticos en Python.

1.2.1 1. Reducción de Dimensionalidad (PCA y SVD)

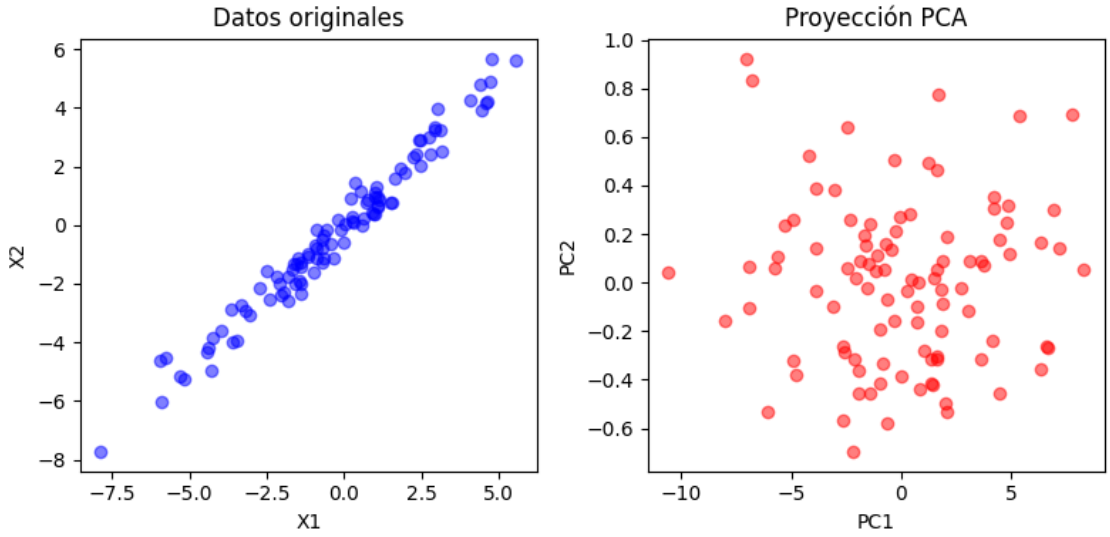
El **Análisis de Componentes Principales (PCA)** transforma los datos a un nuevo sistema de coordenadas donde los ejes (componentes principales) son ortogonales. Cada componente captura la mayor varianza posible, y la ortogonalidad garantiza que la información se distribuye de manera no redundante.

Ejemplo de PCA:

[2]:
```
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Generamos datos sintéticos 2D altamente correlacionados
np.random.seed(42)
N = 100
x = np.random.normal(0, 3, size=N)
y = x + np.random.normal(0, 0.5, size=N)
X = np.column_stack((x, y))

# Aplicamos PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

print("Varianzas explicadas:", np.round(pca.explained_variance_ratio_, 4))

# Visualizamos los datos originales y la proyección en el espacio de PCA
plt.figure(figsize=(8,4))
plt.subplot(1,2,1)
plt.scatter(X[:, 0], X[:, 1], c='blue', alpha=0.5)
plt.title("Datos originales")
plt.xlabel("X1")
```

```
plt.ylabel("X2")

plt.subplot(1,2,2)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c='red', alpha=0.5)
plt.title("Proyección PCA")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.tight_layout()
plt.show()
```

```
Varianzas explicadas: [0.9923 0.0077]
```



En este ejemplo, PCA obtiene dos componentes ortogonales. La primera componente explica la mayor parte de la varianza y la segunda, siendo ortogonal, añade información residual.

1.2.2 Compresión y Filtrado de Imágenes (SVD)

La **Descomposición en Valores Singulares (SVD)** se utiliza para aproximar matrices de datos (como imágenes) mediante componentes ortogonales. Al conservar solo los componentes principales (valores singulares más grandes), se obtiene una aproximación de rango bajo que reduce la dimensión sin perder demasiada información.

Ejemplo de SVD en Python:

[1]:
```
import numpy as np

# Creamos una matriz de ejemplo
A = np.array([[3.0, 1.0, 1.0],
              [1.0, 3.0, 1.0],
              [1.0, 1.0, 3.0]])
```

```python
U, s, Vt = np.linalg.svd(A)

print("Valores singulares:", np.round(s, 4))
print("Matriz U (columnas son vectores ortonormales):\n", np.round(U, 4))
print("Verificación de U ortogonal:\n", np.round(U.T @ U, 4))
```

```
Valores singulares: [5. 2. 2.]
Matriz U (columnas son vectores ortonormales):
 [[-0.5774 -0.      0.8165]
 [-0.5774 -0.7071 -0.4082]
 [-0.5774  0.7071 -0.4082]]
Verificación de U ortogonal:
 [[ 1. -0.  0.]
 [-0.  1. -0.]
 [ 0. -0.  1.]]
```

La SVD descompone la matriz A en $U\Sigma V^T$ y los vectores en U y V son ortogonales. Esta propiedad se utiliza en compresión de imágenes y reducción de ruido.

1.3 Conclusión

Los conceptos de ángulo y ortogonalidad en espacios vectoriales generales son herramientas fundamentales para medir la similitud, construir bases ortonormales y transformar datos. Su aplicación en ciencia de datos se refleja en técnicas de reducción de dimensionalidad (PCA, SVD), en medidas de similitud (similitud coseno) y en preprocesamientos que decorrelacionan y estabilizan el entrenamiento de modelos. Gracias a estas transformaciones, es posible extraer la información esencial de conjuntos de datos complejos y mejorar el desempeño y la interpretabilidad de los algoritmos de aprendizaje automático.

Las implementaciones en Python, mediante bibliotecas como NumPy, scikit-learn y SciPy, facilitan la aplicación de estos conceptos en escenarios reales, desde el análisis de textos hasta la compresión y filtrado de imágenes.

Referencias adicionales:

1. Strang, G. (2009). *Introduction to Linear Algebra.* Wellesley-Cambridge Press.
2. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning.* Springer.
3. Documentación de scikit-learn: https://scikit-learn.org/stable/
4. Documentación de NumPy: https://numpy.org/doc/

Bases_ortonormales

March 4, 2025

1 Bases ortonormales

1.1 Explicacion Matematica

1.1.1 Explicación

Una base ortonormal es, en escencia, un sistema de referencia hecho de vectores unitarios mutuamente perpendiculares. En el espacio $\mathbb{R}^3$, un ejemplo de base ortornoaml estándar es

$$e_1, e_2, e_3 = (1,0,0), (0,1,0), (0,0,1)$$

Aquí:

- Cada vector tiene normal $\|e_1\| = \|e_2\| = \|e_3\| = 1$
- Son ortogonales entre sí: $\langle e_i, e_j \rangle = 0$ para $i \neq j$

Geometricamente, esto significa que cada vector apunta en una dirección completamente independiente de los otros, formando un ángulo recto. El concepto se generaliza a cualquier dimensión.

1.1.2 Explicación formal

Sea V un espacio vectorial de dimensión finita n sobre $\mathbb{R}$, con producto interno $\langle *, * \rangle$. Un conjunto de vectores $v_1, v_2, ..., v_n$ es una base ortornoaml si satisface

1- Ortonormalidad:

$$\langle v_i, v_j \rangle = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

Esto significa que cada vector tiene norma 1 (normalidad) y son ortogonales entre sí (ortogonalidad)

2- Base:

El conjunto es base si cualquier vector $v \in$ se puede escribir como combinación lineal

$$v = c_1 v_1 + c_2 v_2 + \cdots + c_n v_n$$

Si el espacio tiene producto interno, los coeficientes son fáciles de calcular

$$c_i = \langle v, v_i \rangle$$

Esto simplifica muchas operaciones, lo cual es una ventaja de trabajar con bases ortonormales.

1.2 Aplicaciones en Ciencias de datos

2 Ejemplo practico: PCA (Análisis de Componentes Principales)

2.1 Objetivo

Reducir la dimensión de un conjunto de datos, manteniendo la mayor cantidad posible de información.

1. **Centralizar los datos**
   Restamos la media de cada característica para que los datos estén centrados en el origen.
2. **Calcular la matriz de covarianza**
   Esta matriz mide cómo varían las características entre sí.

   Ejemplo: Si las características son altura y peso, la covarianza mide si al aumentar la altura, también suele aumentar el peso.
3. **Descomposición espectral**
   Se calculan los **autovectores y autovalores** de la matriz de covarianza. Estos autovectores forman una **base ortonormal**.
4. **Ordenar por importancia**
   Los autovalores indican cuánta información (varianza) explica cada autovector. Los autovectores con autovalores más grandes son las direcciones principales (componentes principales).
5. **Cambio de base**
   Proyectamos los datos originales sobre esta base ortonormal de componentes principales. Esto transforma el dataset original a un nuevo espacio donde las coordenadas son independientes entre sí (descorreladas).
6. **Reducción**
   Si solo queremos las 2 o 3 direcciones principales, nos quedamos con los primeros vectores de la base ortonormal y descartamos el resto.

2.2 Ejemplo simple

Si los datos originales son puntos en 3D:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Después de PCA, los datos podrían vivir en 2D:

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

---

3 Ejemplo practico: Normalización de embeddings en NLP

3.1 Objetivo

Al trabajar con vectores de palabras (word embeddings) o embeddings de documentos, muchas veces se normalizan para tener norma 1. Esto es útil para usar métricas como el **coseno** para medir similitud.

1. **Obtener embeddings**
   El modelo convierte cada palabra en un vector:

$$w_i = \begin{bmatrix} w_{i1} \\ w_{i2} \\ \vdots \\ w_{i300} \end{bmatrix}$$

2. **Normalización**
   Cada vector se divide por su norma:

$$w_i^{(norm)} = \frac{w_i}{\|w_i\|}$$

Después de esto, cada vector tiene norma 1.

3. **Espacio esférico**
   Después de normalizar, todos los vectores viven sobre la superficie de una hiperesfera unidad. Este espacio es equivalente a trabajar en una base ortonormal estándar, ya que:

- Cada vector normalizado tiene norma 1.
- El coseno es una proyección en una base ortonormal esférica.

4. **Medir similitud**
   Ahora, la similitud entre dos palabras se mide directamente con:

$$\cos\theta = \langle w_i^{(norm)}, w_j^{(norm)} \rangle$$

Como los vectores están normalizados, este producto interno es exactamente el coseno del ángulo entre los vectores.

3.2 Ejemplo simple

Palabra “perro”:

$$w_{perro} = [0.3, 0.4, 0.5]$$

$$\|w_{perro}\| = \sqrt{0.3^2 + 0.4^2 + 0.5^2} = \sqrt{0.5} = 0.707$$

$$w_{perro}^{(norm)} = \frac{1}{0.707} \cdot [0.3, 0.4, 0.5]$$

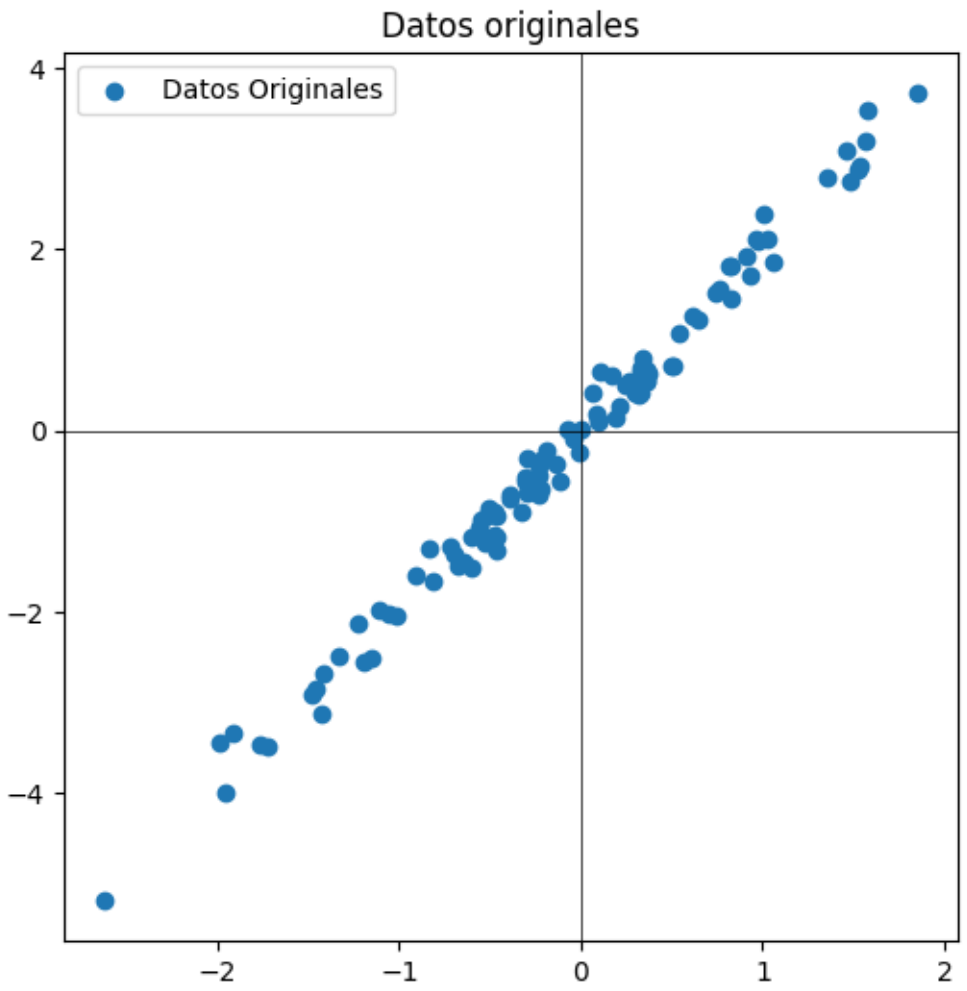
Así garantizas que cada vector vive en la misma esfera.

4 Codigo

[8]:
```
import numpy as np
import matplotlib.pyplot as plt
```

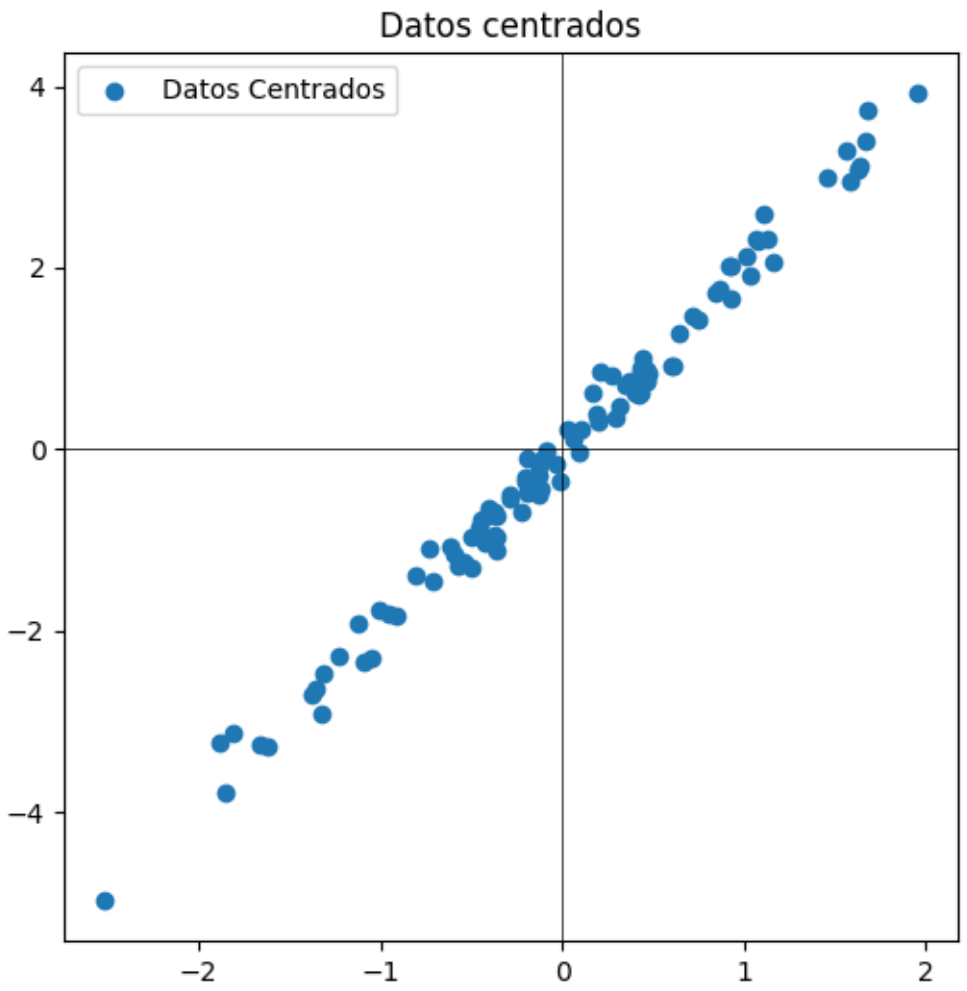
[9]:
```
# Generar datos artificiales (simulación de un dataset 2D)
np.random.seed(42)
x = np.random.normal(0, 1, 100)
y = 2 * x + np.random.normal(0, 0.2, 100)

data = np.column_stack((x, y))
```

[10]:
```
# Visualizar datos originales
plt.figure(figsize=(6, 6))
plt.scatter(data[:, 0], data[:, 1], label="Datos Originales")
plt.axhline(0, color='black', lw=0.5)
plt.axvline(0, color='black', lw=0.5)
plt.legend()
plt.title("Datos originales")
plt.show()
```



[11]:
```
# Centrar los datos (restar la media)
mean = np.mean(data, axis=0)
data_centrado = data - mean
```

[12]:
```
# Visualizar datos centrados
plt.figure(figsize=(6, 6))
plt.scatter(data_centrado[:, 0], data_centrado[:, 1], label="Datos Centrados")
plt.axhline(0, color='black', lw=0.5)
plt.axvline(0, color='black', lw=0.5)
plt.legend()
plt.title("Datos centrados")
plt.show()
```



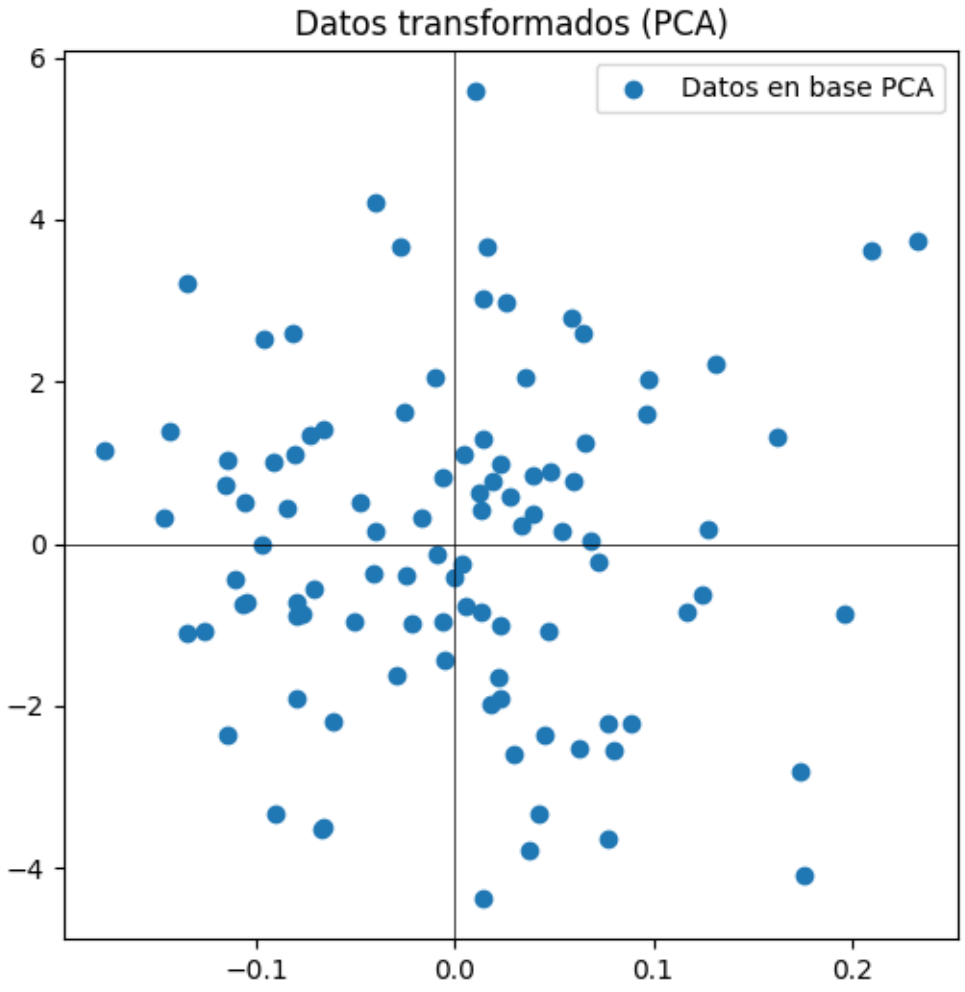
[13]:
```
# Calcular matriz de covarianza
cov_matrix = np.cov(data_centrado.T)

# alcular autovalores y autovectores
autovalores, autovectores = np.linalg.eig(cov_matrix)

# Mostrar autovectores y autovalores
print("Autovalores:")
print(autovalores)
print("\nAutovectores (Base Ortonormal):")
print(autovectores)
```

```
Autovalores:
[0.00725553 4.05844996]

Autovectores (Base Ortonormal):
```

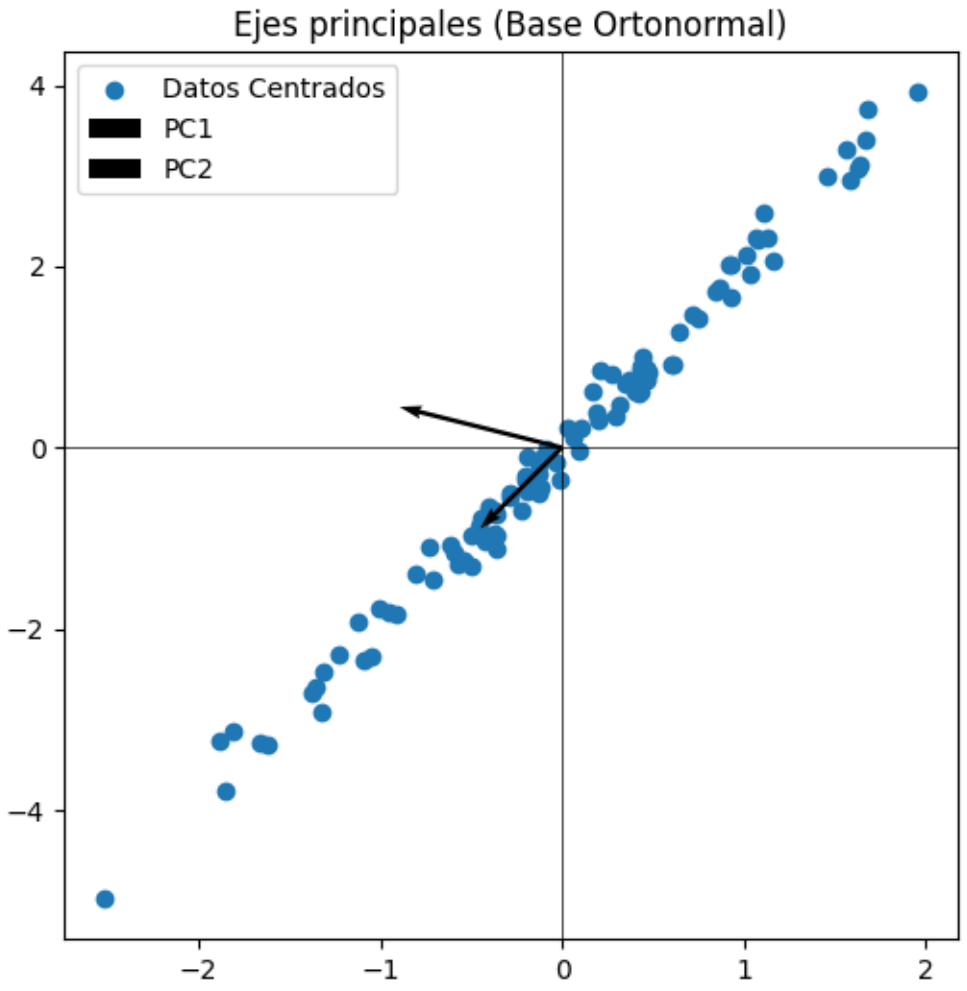
```
[[-0.8934227  -0.44921697]
 [ 0.44921697 -0.8934227 ]]
```

[14]:
```
# Proyectar datos sobre la nueva base ortonormal
data_pca = np.dot(data_centrado, autovectores)
```

[15]:
```
# Visualizar datos proyectados (en la base PCA)
plt.figure(figsize=(6, 6))
plt.scatter(data_pca[:, 0], data_pca[:, 1], label="Datos en base PCA")
plt.axhline(0, color='black', lw=0.5)
plt.axvline(0, color='black', lw=0.5)
plt.legend()
plt.title("Datos transformados (PCA)")
plt.show()
```



[16]:
```
# Visualizar ejes de PCA sobre los datos originales centrados
plt.figure(figsize=(6, 6))
plt.scatter(data_centrado[:, 0], data_centrado[:, 1], label="Datos Centrados")
for i in range(2):
    vector = autovectores[:, i]
    plt.quiver(0, 0, vector[0], vector[1], angles='xy', scale_units='xy',␣
 ↪scale=1, width=0.005, label=f"PC{i+1}")

plt.axhline(0, color='black', lw=0.5)
plt.axvline(0, color='black', lw=0.5)
plt.legend()
plt.title("Ejes principales (Base Ortonormal)")
plt.show()
```



5 Enlaces

https://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/

http://tutorial.math.lamar.edu/Classes/LinAlg/OrthonormalBases.aspx

https://www.math.utah.edu/~zwick/Classes/Fall2012_2270/Lectures/Lecture18.pdf

https://cs229.stanford.edu/notes2020spring/cs229-notes10.pdf

https://www.tensorflow.org/text/guide/word_embeddings

complemento_Ortogonal

March 4, 2025

1 Complemento Ortogonal

1.1 Explicacion Matematica

El concepto de ortogonalidad y complementos ortogonales se remonta a los trabajos de matemáticos como Euclides y Pitágoras, quienes estudiaron las propiedades de los triángulos rectángulos y las relaciones entre sus lados. Sin embargo, el desarrollo formal del complemento ortogonal como lo conocemos hoy en día se consolidó con el avance del álgebra lineal en los siglos XIX y XX.

En términos simples, el complemento ortogonal de un subespacio

W en un espacio vectorial

V es el conjunto de todos los vectores en

V que son ortogonales a cada vector en

W. Matemáticamente, si

W es un subespacio de

V, el complemento ortogonal de

W, denotado como

W , se define como:

={v V v,w =0 para todo w W}

2 Uso exacto

2.1 1. Descomposición de Vectores

En álgebra lineal, cualquier vector en un espacio vectorial puede descomponerse en una parte que pertenece a un subespacio dado y otra parte que pertenece a su complemento ortogonal. Esto es útil en la resolución de sistemas de ecuaciones lineales y en la proyección de vectores.

2.2 2. Método de Mínimos Cuadrados

En estadística y análisis de datos, el método de mínimos cuadrados se utiliza para encontrar la mejor aproximación lineal a un conjunto de datos. La idea es minimizar la suma de los cuadrados de las diferencias entre los valores observados y los valores predichos. Esto se logra proyectando los datos en el subespacio generado por las variables explicativas, y el error de proyección pertenece al complemento ortogonal.

2.3 3. Análisis de Fourier

En el análisis de Fourier, las funciones se descomponen en una serie de senos y cosenos ortogonales. Los coeficientes de Fourier se obtienen proyectando la función original en los subespacios generados por estas funciones ortogonales.

2.4 4. Procesamiento de Señales

En el procesamiento de señales, las señales se descomponen en componentes ortogonales para facilitar su análisis y manipulación. Por ejemplo, en la Transformada Discreta de Fourier (DFT), las señales se descomponen en una suma de senos y cosenos ortogonales, lo que permite analizar las frecuencias presentes en la señal.

2.5 5. Optimización

En problemas de optimización, el complemento ortogonal se utiliza para encontrar direcciones de búsqueda que sean ortogonales a las restricciones del problema. Esto es útil en métodos como la optimización cuadrática y la programación lineal.

2.5.1 Ejemplo Práctico: Regresión Lineal (lineal regretion)

Supongamos que tenemos un conjunto de datos con n observaciones (puede ser un producto vs su precio en un ambito practico o las ventas en un año o incluso la edad vs el salario) y queremos ajustar una línea recta que minimice la suma de los cuadrados de las diferencias entre los valores observados (cuando se ingrese la edad pueda de una forma sencilla acercarse a lo que gana una persona) y los valores predichos. Este es un problema clásico de regresión lineal.

Datos Imaginemos que tenemos los siguientes datos:

x	y
1	2
2	3
3	5
4	4
5	6

Queremos ajustar una línea de la forma $y = mx + b$.

Matriz de Diseño Primero, construimos la matriz de diseño X y el vector de observaciones $\mathbf{y}$:

$$X = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} 2 \\ 3 \\ 5 \\ 4 \\ 6 \end{pmatrix}$$

Solución de Mínimos Cuadrados La solución de mínimos cuadrados se obtiene resolviendo la ecuación normal:

$$\hat{} = (X^T X)^{-1} X^T \mathbf{y}$$

Donde$\hat{}$ es el vector de coeficientes (b, m).

Cálculo Calculamos $X^T X$ y $X^T \mathbf{y}$:

$$X^T X = \begin{pmatrix} 5 & 15 \\ 15 & 55 \end{pmatrix}, \quad X^T \mathbf{y} = \begin{pmatrix} 20 \\ 70 \end{pmatrix}$$

Luego, invertimos $X^T X$ y multiplicamos por $X^T \mathbf{y}$:

$$(X^T X)^{-1} = \begin{pmatrix} 11 & -3 \\ -3 & 1 \end{pmatrix}, \quad \hat{} = \begin{pmatrix} 11 & -3 \\ -3 & 1 \end{pmatrix} \begin{pmatrix} 20 \\ 70 \end{pmatrix} = \begin{pmatrix} 0.4 \\ 1.0 \end{pmatrix}$$

Por lo tanto, la línea de mejor ajuste es:

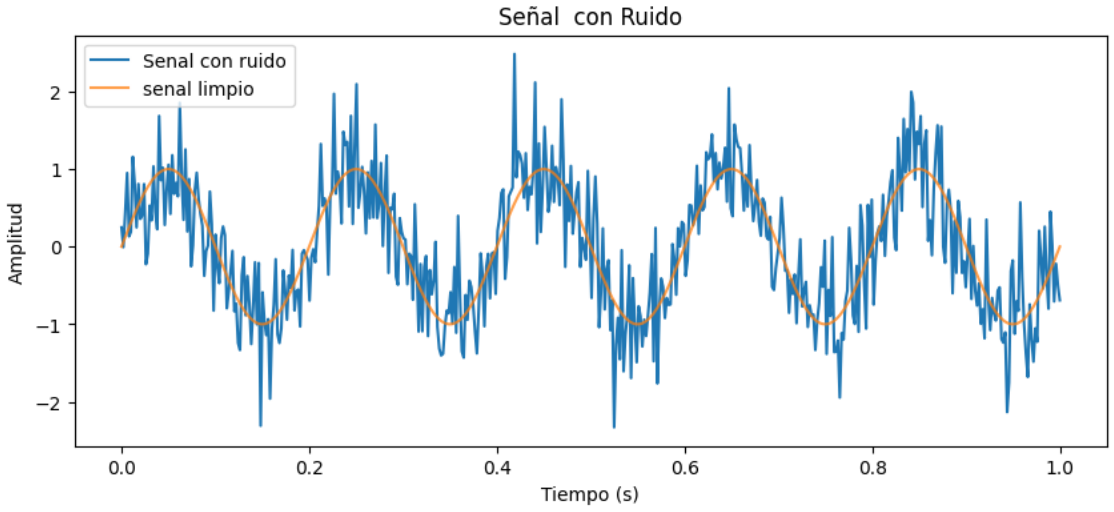
$$y = 1.0x + 0.4$$

Interpretación del Complemento Ortogonal El error de ajuste, es decir, la diferencia entre los valores observados y los valores predichos, pertenece al complemento ortogonal del subespacio generado por las columnas de X. Esto significa que los errores son ortogonales a los vectores de la matriz de diseño, lo que garantiza que hemos encontrado la mejor aproximación lineal en el sentido de mínimos cuadrados.

3 Codigo

[7]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Generar una señal  simulada
np.random.seed(42)
t = np.linspace(0, 1, 500)
ecg_clean = np.sin(2 * np.pi * 5 * t)  # Señal limpia
noise = 0.5 * np.random.randn(500)  # Ruido
ecg_noisy = ecg_clean + noise  # Señal con ruido

# Graficar la señal con ruido
plt.figure(figsize=(10, 4))
plt.plot(t, ecg_noisy, label='Senal con ruido')
plt.plot(t, ecg_clean, label='senal limpio', alpha=0.75)
plt.legend()
plt.title('Señal  con Ruido')
plt.xlabel('Tiempo (s)')
```

```
plt.ylabel('Amplitud')
plt.show()
```

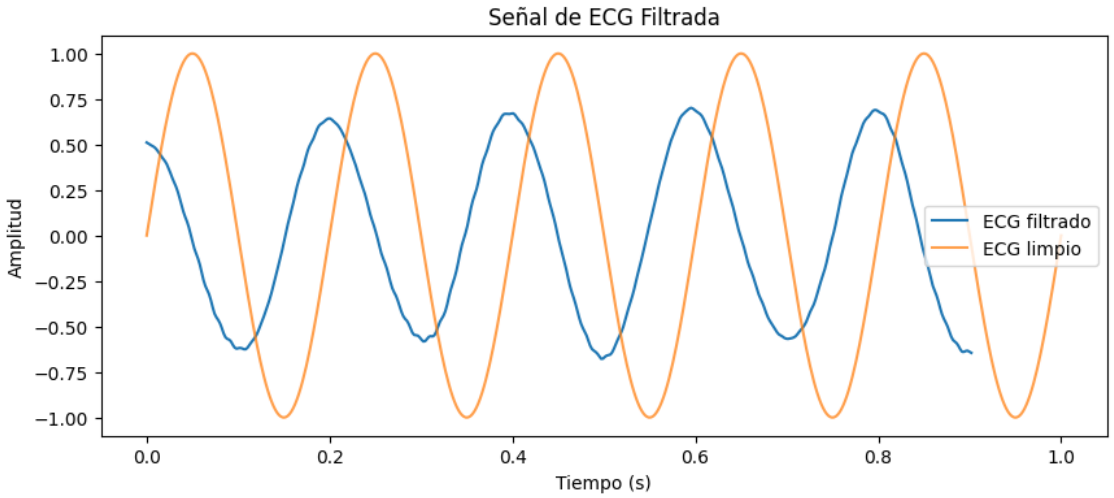


La anterior imagen es dentro lo amarillo es la senal emitida desde un sistema scada, en el cual puede ser la trasmiscion de un elemento de oil and gas donde puede pasar un elemento como el agua, la cual se puede contaminar la salida, por temas externos(fallas en el cable, contaminacion de h20 con otros elmento como lo es la tierra)

[8]:
```
from scipy.linalg import svd

# Crear una matriz de Hankel para aplicar SVD
def hankel_matrix(signal, L):
    N = len(signal)
    return np.array([signal[i:i+L] for i in range(N-L+1)])

L = 50  # Tamaño de la ventana
H = hankel_matrix(ecg_noisy, L)

# Aplicar SVD
U, s, Vt = svd(H, full_matrices=False)
```

[9]:
```
# Reconstruir la señal utilizando los primeros k componentes principales
k = 5
H_denoised = np.dot(U[:, :k], np.dot(np.diag(s[:k]), Vt[:k, :]))

# Promediar las filas de la matriz de Hankel reconstruida para obtener la señal␣
 ↪filtrada
ecg_denoised = np.mean(H_denoised, axis=1)

# Graficar la señal filtrada
```

```
plt.figure(figsize=(10, 4))
plt.plot(t[:len(ecg_denoised)], ecg_denoised, label='ECG filtrado')
plt.plot(t, ecg_clean, label='ECG limpio', alpha=0.75)
plt.legend()
plt.title('Señal de ECG Filtrada')
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.show()
```



Con los datos puedes limpiar las diferentes senales, que te permitiran de una forma mas claara que es lo que esta pasadno dentro de lo cual, el ruido se elimina con los componentes ortogonales, correspondiendo a el ruido captado o trasmitido por el SCAda(elemento de captura de datos)

4 Enlaces

http://matematicas.uam.es/~fernando.chamizo/asignaturas/2021alglin/sections/4_3.pdf

https://www.britannica.com/science/

https://mathshistory.st-andrews.ac.uk/Biographies/Grassmann/

https://machinelearningmastery.com/article/11751456084/

https://programmerclick.com/article/6115655816/

productoInternoFunciones

March 4, 2025

0.0.1 Producto Interno de Funciones

Historia y Desarrollo El concepto de producto interno tiene sus raíces en el desarrollo del álgebra lineal y el análisis funcional. Fue formalizado en el siglo XIX por matemáticos como **Hermann Grassmann** y **David Hilbert**. Grassmann introdujo la idea de un espacio vectorial, mientras que Hilbert desarrolló los espacios que llevan su nombre, donde el producto interno juega un papel crucial.

Definición Matemática El producto interno de dos funciones (f) y (g) en un intervalo ([a, b]) se define como:

$$\langle f, g \rangle = \int_a^b f(x)g(x)\,dx$$

Este producto interno satisface las siguientes propiedades: 1. **Linealidad**: $\langle af + bg, h \rangle = a\langle f, h \rangle + b\langle g, h \rangle$ 2. **Simetría**: $\langle f, g \rangle = \langle g, f \rangle$ 3. **Positividad**: $\langle f, f \rangle \geq 0$ y $\langle f, f \rangle = 0$ si y solo si $ f = 0 $

Conclusión El producto interno de funciones es una herramienta fundamental en matemáticas, permitiendo definir conceptos como la ortogonalidad y la norma de funciones. Es esencial en el estudio de espacios de Hilbert y tiene aplicaciones en diversas áreas de la ciencia y la ingeniería.

0.0.2 Aplicaciones en Ciencia de Datos

1. **Análisis de Componentes Principales (PCA)**: Utiliza el producto interno para encontrar las direcciones principales de variación en los datos.
2. **Métodos de Mínimos Cuadrados**: Utiliza el producto interno para minimizar el error en la regresión lineal.
3. **Procesamiento de Señales**: Utiliza el producto interno para filtrar y analizar señales en diferentes dominios.
4. **Análisis de Fourier**: Utiliza el producto interno para descomponer funciones en series de senos y cosenos ortogonales.
5. **Redes Neuronales**: Utiliza el producto interno en la propagación hacia adelante y el cálculo de gradientes.

0.0.3 Ejemplo Práctico: Sistema SCADA en la Industria del Petróleo y Gas

Descripción del Problema En la industria del petróleo y gas, los sistemas SCADA (Supervisory Control and Data Acquisition) son esenciales para monitorear y controlar operaciones remotas. Un desafío común es la presencia de ruido en los datos de sensores, lo que puede afectar la precisión

del monitoreo y control. Nuestro objetivo es filtrar el ruido de los datos de sensores utilizando el complemento ortogonal.

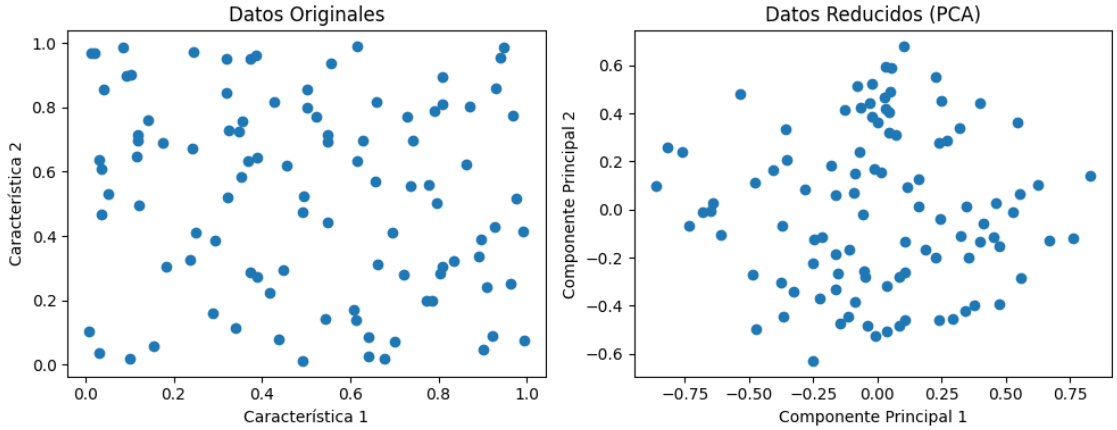
Solución Planteada Utilizaremos un método basado en la descomposición en valores singulares (SVD) para separar la señal útil del ruido en los datos de sensores. La idea es proyectar los datos en un subespacio generado por los componentes principales y utilizar el complemento ortogonal para eliminar el ruido.

Implementación en Python

1 Codigo

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
```

[2]:
```
# Generar datos simulados de sensores
np.random.seed(42)
X = np.random.rand(100, 5)  # 100 muestras, 5 características

# Aplicar PCA para reducir la dimensionalidad a 2 componentes principales
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Graficar los datos originales y los datos reducidos
plt.figure(figsize=(10, 4))

# Datos originales
plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1])
plt.title('Datos Originales')
plt.xlabel('Característica 1')
plt.ylabel('Característica 2')

# Datos reducidos
plt.subplot(1, 2, 2)
plt.scatter(X_pca[:, 0], X_pca[:, 1])
plt.title('Datos Reducidos (PCA)')
plt.xlabel('Componente Principal 1')
plt.ylabel('Componente Principal 2')

plt.tight_layout()
plt.show()
```



2 4. Uso del Producto Interno de Funciones

Dimension de la dimensionalidad de un dataset, cuando se tienen demasiadas dimensiones este analisis cobra realmente mucha impotancia, dado que se seleccionan caracteristicas fundamentales y permite trabajar de una mejor forma cocn los datos.

4.1. de esta forma se impactan temas que tiene que ver con el 4.2. overtitting del modelo, 4.3. la alta dimensionlidad y el computo disponible, 4.4 ademas de la redudancia de los datos.

El producto interno de funciones se utiliza aquí para proyectar los datos en el subespacio generado por los componentes principales. Esto permite separar la señal útil del ruido, ya que los componentes principales capturan la mayor parte de la variabilidad de la señal, mientras que el ruido se distribuye en los componentes menores.

3 Enlaces

https://medium.com/@roshmitadey/understanding-principal-component-analysis-pca-d4bb40e12d33

https://oil-gas.net/implementing-scada-systems-for-enhanced-operational-control/

http://matematicas.uam.es/~fernando.chamizo/asignaturas/2021alglin/sections/4_3.pdf

https://mathshistory.st-andrews.ac.uk/Biographies/Grassmann/

https://machinelearningmastery.com/article/11751456084/

https://programmerclick.com/article/6115655816/

ProyecciónOrtogonal

March 4, 2025

1 Proyección Ortogonal

1.1 Explicacion Matematica

Para definir una proyección ortogonal se requieren de las siguientes definiciones:

1. Espacio Vectorial y Subespacios:

Consideremos un espacio vectorial V y un subespacio W de V que también es un espacio vectorial. En el contexto de la proyección ortogonal, nos enfocamos en subespacios generados por un vector $\mathbf{u}$, que comprende todos los vectores que son múltiplos escalares de $\mathbf{u}$.

2. Producto Interno (o Producto Escalar):

El producto interno entre dos vectores $\mathbf{v}$ y $\mathbf{u}$ en V, denotado como $\langle \mathbf{v}, \mathbf{u} \rangle$, es una función que asigna un escalar a cada par de vectores. En el caso de espacios vectoriales reales, el producto interno se conoce como producto punto, definido geométricamente como:

$\langle \mathbf{v}, \mathbf{u} \rangle = ||\mathbf{v}||||\mathbf{u}|| \cos(\theta)$,

donde $||\mathbf{v}||$ y $||\mathbf{u}||$ son las magnitudes de $\mathbf{v}$ y $\mathbf{u}$, respectivamente, y θ es el ángulo entre ellos. Algebraicamente, si $\mathbf{v} = (v_1, v_2, ..., v_n)$ y $\mathbf{u} = (u_1, u_2, ..., u_n)$ entonces:

$\langle \mathbf{v}, \mathbf{u} \rangle = v_1u_1 + v_2u_2 + ... + v_nu_n$.

3. Proyección Ortogonal:

La proyección ortogonal de un vector $\mathbf{v}$ sobre un vector $\mathbf{u}$, denotada como $\text{proj}_{\mathbf{u}}\mathbf{v}$, es un vector en $\mathbf{u}$ que minimiza la distancia entre $\mathbf{v}$ y cualquier vector en $\mathbf{u}$. Esta distancia se mide perpendicularmente a $\mathbf{u}$, garantizando la ortogonalidad.

4. Deducción de la Fórmula:

Dado que $\text{proj}_{\mathbf{u}}\mathbf{v}$ pertenece a $\mathbf{u}$, se puede expresar como:

$\text{proj}_{\mathbf{u}}\mathbf{v} = \alpha\mathbf{u}$,

donde α es un escalar. Para asegurar la ortogonalidad, requerimos que $\mathbf{v} - \text{proj}_{\mathbf{u}}\mathbf{v}$ sea ortogonal a $\mathbf{u}$, es decir:

$\langle \mathbf{v} - \text{proj}_{\mathbf{u}}\mathbf{v}, \mathbf{u} \rangle = 0$.

Sustituyendo la expresión de la proyección y aplicando la linealidad del producto interno, obtenemos:

$\langle \mathbf{v} - \alpha\mathbf{u}, \mathbf{u} \rangle = \langle \mathbf{v}, \mathbf{u} \rangle - \alpha\langle \mathbf{u}, \mathbf{u} \rangle = 0$.

Despejando α, llegamos a:

$\alpha = \frac{\langle \mathbf{v}, \mathbf{u} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle}$.

Finalmente, sustituyendo α en la expresión de la proyección, obtenemos la fórmula:

$\text{proj}_{\mathbf{u}} \mathbf{v} = \frac{\langle \mathbf{v}, \mathbf{u} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle} \mathbf{u}$.

5. Propiedades:

Ortogonalidad: $\mathbf{v} - \text{proj}_{\mathbf{u}} \mathbf{v}$ es ortogonal a $\mathbf{u}$.

Unicidad: La proyección ortogonal de $\mathbf{v}$ sobre $\mathbf{u}$ es única.

Linealidad: $\text{proj}_{\mathbf{u}}(a\mathbf{v} + b\mathbf{w}) = a(\text{proj}_{\mathbf{u}} \mathbf{v}) + b(\text{proj}_{\mathbf{u}} \mathbf{w})$ para cualquier escalar a, b y vectores $\mathbf{v}, \mathbf{w}$.

Idempotencia: $\text{proj}_{\mathbf{u}}(\text{proj}_{\mathbf{u}} \mathbf{v}) = \text{proj}_{\mathbf{u}} \mathbf{v}$

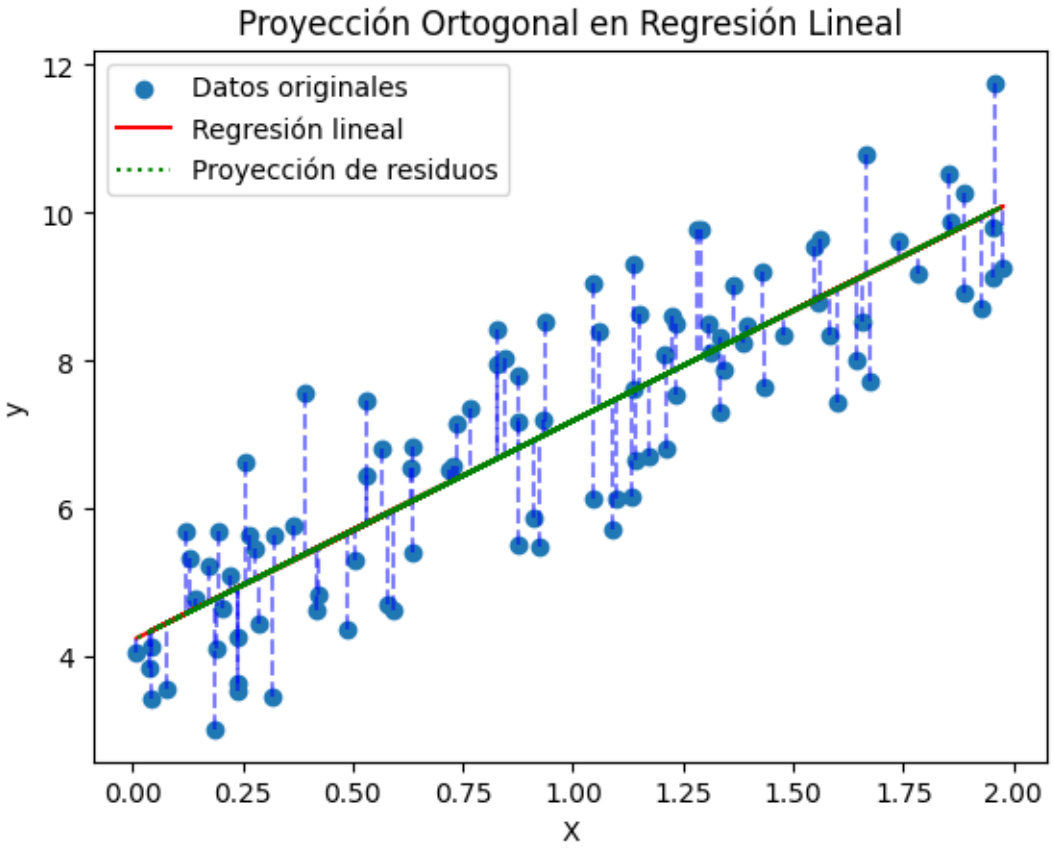
1.2 Aplicaciones en Ciencias de datos

En la regresión lineal, buscamos encontrar la mejor línea (o hiperplano en dimensiones mayores) que se ajuste a un conjunto de puntos de datos. La línea de regresión se puede encontrar minimizando la suma de los cuadrados de las distancias verticales entre los puntos de datos y la línea. Estas distancias verticales son precisamente las magnitudes de las proyecciones ortogonales de los vectores de error (la diferencia entre los valores reales y los predichos) sobre la dirección perpendicular a la línea de regresión.

1.2.1 Ejemplo en Python para Uso de proyección ortogonal en regresiones lineales

[3]:
```
import numpy as np
import matplotlib.pyplot as plt

# Generar datos de ejemplo
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Agregar una columna de unos a X para el término independiente
X_b = np.c_[np.ones((100, 1)), X]

# Calcular los coeficientes de la regresión lineal usando la ecuación normal
theta_best = np.linalg.solve(X_b.T.dot(X_b), X_b.T.dot(y))

# Predecir los valores de y usando los coeficientes
y_predict = X_b.dot(theta_best)

# Calcular los residuos (errores)
residuos = y - y_predict

# Calcular la proyección ortogonal de los residuos sobre el subespacio generado␣
↪por X_b
```

```
proyeccion_residuos = X_b.dot(np.linalg.solve(X_b.T.dot(X_b), X_b.T.
 ↪dot(residuos)))

# Visualizar los resultados
plt.scatter(X, y, label="Datos originales")
plt.plot(X, y_predict, color="red", label="Regresión lineal")
for i in range(len(X)):
    plt.plot([X[i], X[i]], [y[i], y_predict[i]], color="blue", linestyle="--",␣
 ↪alpha=0.5)  # Líneas de proyección
plt.plot(X, proyeccion_residuos + y_predict, color="green", linestyle=":",␣
 ↪label="Proyección de residuos")
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.title("Proyección Ortogonal en Regresión Lineal")
plt.show()
```



Rotaciones

March 4, 2025

1 Rotaciones

1.1 Explicacion Matematica

Las rotaciones son transformaciones geométricas que mueven uno o varios puntos en un espacio bidimensional o tridimensional alrededor de un punto central fijo, llamado centro de rotación, sin alterar la forma ni el tamaño del objeto que se está rotando. La cantidad de rotación se define por el ángulo de rotación, que generalmente se mide en grados y en ocasiones en radianes. Y se pueden describir utilizando matrices de rotación.

1.1.1 Rotaciones en 2D

En un plano cartesiano bidimensional, una rotación se define por un ángulo θ y un centro de rotación.

Matriz de Rotación en 2D:

La transformación de un punto (x, y) (x, y) a un punto rotado (x', y') (x', y') se realiza mediante la multiplicación por la siguiente matriz de rotación:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Donde:

θ es el ángulo de rotación en radianes (positivo para rotaciones en sentido antihorario). $\cos(\theta)$ es el coseno del ángulo de rotación. $\sin(\theta)$ es el seno del ángulo de rotación. Aplicando la Rotación en 2D:

Para obtener las coordenadas del punto rotado (x', y') (x', y'), se multiplica la matriz de rotación por las coordenadas del punto original (x, y) (x, y):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Esto se traduce en las siguientes ecuaciones:

$$x' = x\cos(\theta) - y\sin(\theta) y' = x\sin(\theta) + y\cos(\theta)$$

$$x' = x\cos(\theta) - y\sin(\theta) y' = x\sin(\theta) + y\cos(\theta)$$

Estas ecuaciones describen cómo las coordenadas del punto original (x, y) (x, y) se transforman en las coordenadas del punto rotado (x', y') (x', y') después de una rotación de ángulo θ alrededor del origen.

1.1.2 Rotaciones en 3D

Las rotaciones en el espacio tridimensional involucran rotaciones alrededor de tres ejes: x, y, z. Cada rotación se describe mediante una matriz de rotación de 3x3.

Matrices de Rotación en 3D:

Rotación alrededor del eje x:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Rotación alrededor del eje y:

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Rotación alrededor del eje z:

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Combinando Rotaciones en 3D:

Para realizar rotaciones compuestas en 3D, se multiplican las matrices de rotación correspondientes. Es crucial tener en cuenta que el orden de multiplicación importa, ya que las rotaciones en 3D no son conmutativas. La rotación resultante dependerá del orden en que se apliquen las rotaciones individuales.

Ejemplo:

Para rotar un punto primero alrededor del eje x por un ángulo α y luego alrededor del eje y por un ángulo β, se utiliza la siguiente matriz compuesta:

$$R = R_y(\beta)R_x(\alpha)$$
$$R = R_y(\beta)R_x(\alpha)$$

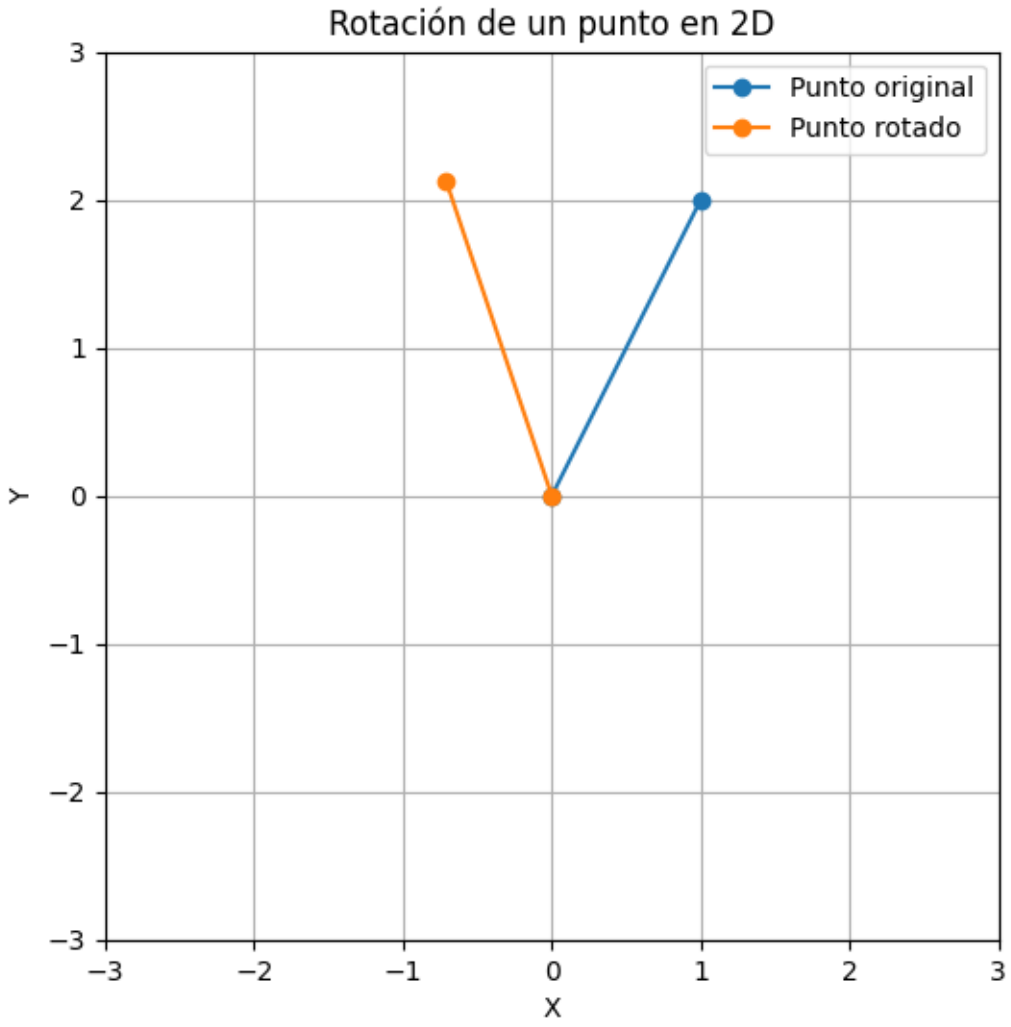
La multiplicación de matrices se realiza de derecha a izquierda, lo que significa que la rotación alrededor del eje x se aplica primero.

1.1.3 Codigo en Python para ejemplo de rotación

[1]:
```
### Ejemplo de Código en Python para rotar un punto

#importar las librerias
import numpy as np
import matplotlib.pyplot as plt

# Definir un punto
punto = np.array([1, 2])

# Definir el ángulo de rotación en radianes
angulo = np.pi / 4  # 45 grados

# Matriz de rotación 2D
matriz_rotacion = np.array([
    [np.cos(angulo), -np.sin(angulo)],
    [np.sin(angulo), np.cos(angulo)]
])

# Aplicar la rotación
punto_rotado = np.dot(matriz_rotacion, punto)

# Imprimir los resultados
print("Punto original:", punto)
print("Punto rotado:", punto_rotado)


# Visualizar la rotación
plt.figure(figsize=(6,6))
plt.plot([0, punto[0]], [0, punto[1]], label='Punto original', marker='o',␣
 ↪linestyle='-')
plt.plot([0, punto_rotado[0]], [0, punto_rotado[1]], label='Punto rotado',␣
 ↪marker='o', linestyle='-')

plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.title('Rotación de un punto en 2D')
```

```
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.grid(True)
plt.show()
```

```
Punto original: [1 2]
Punto rotado: [-0.70710678  2.12132034]
```



1.2 Aplicaciones en Ciencias de datos

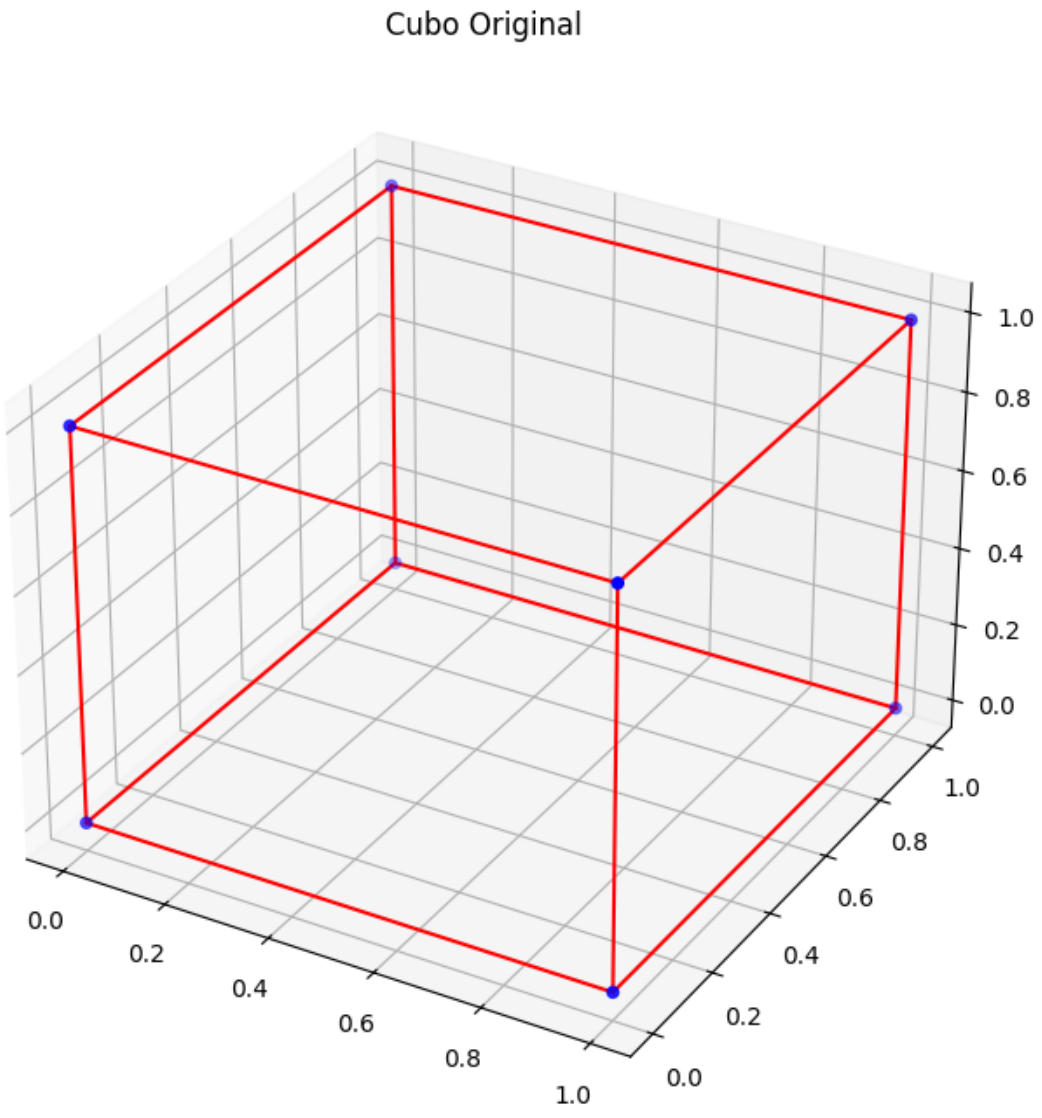
Las rotaciones tienen gran cantidad de aplicaciones en el campo de la ciencia de datos. Algunas de las que hemos usado es en la creación de objetos tridimensionales y su rotación. Por ejemplo los objetos 3D necesitan ser rotados para ser visualizados desde diferentes ángulos. Mediante el uso de las rotaciones se genera la base para la manipulación y visualización de objetos 3D. Esto funciona

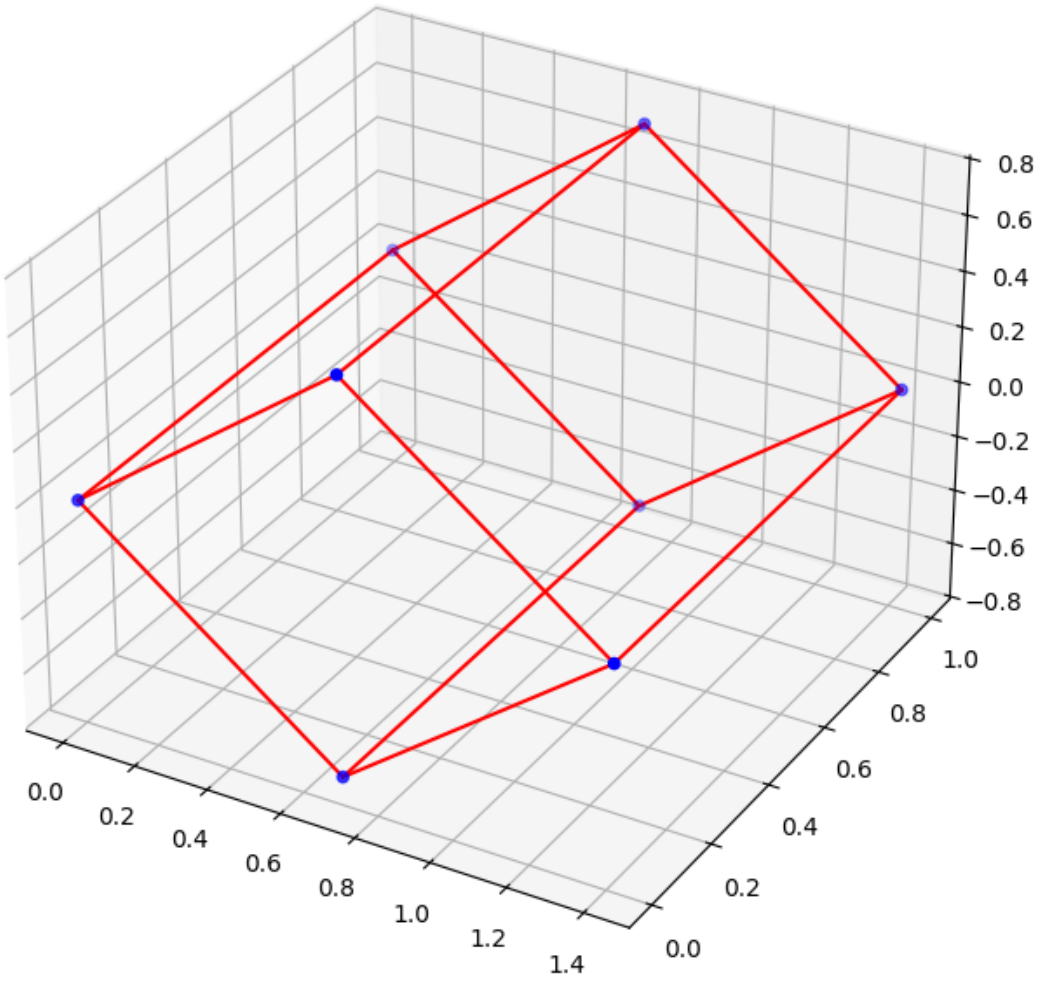
a partir de las matrices de rotación que se aplican para transformar las coordenadas de los vértices de los objetos 3D, permitiendo la rotación alrededor de diferentes ejes.

1.2.1 Ejemplo en Python para rotar un Cubo en 3D con un ángulo de rotación 45°

[2]:
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Definir un cubo simple
puntos_cubo = np.array([
    [0, 0, 0],
    [1, 0, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 1]
])

# Función para dibujar el cubo
def dibujar_cubo(puntos, ax):
    ax.scatter(puntos[:, 0], puntos[:, 1], puntos[:, 2], c='b', marker='o')
    # Conectar los puntos para formar las aristas del cubo
    edges = [
        [0, 1], [1, 2], [2, 3], [3, 0],
        [4, 5], [5, 6], [6, 7], [7, 4],
        [0, 4], [1, 5], [2, 6], [3, 7]
    ]
    for edge in edges:
        ax.plot(puntos[edge, 0], puntos[edge, 1], puntos[edge, 2], c='r')

# Crear la figura 3D
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

# Dibujar el cubo original
dibujar_cubo(puntos_cubo, ax)
ax.set_title("Cubo Original")

# Aplicar rotación alrededor del eje y (45 grados)
angulo = np.pi / 4  # 45 grados en radianes
matriz_rotacion_y = np.array([
    [np.cos(angulo), 0, np.sin(angulo)],
    [0, 1, 0],
    [-np.sin(angulo), 0, np.cos(angulo)]
```

```
])
puntos_rotados = np.dot(puntos_cubo, matriz_rotacion_y.T)

# Crear una nueva figura para el cubo rotado
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

# Dibujar el cubo rotado
dibujar_cubo(puntos_rotados, ax)
ax.set_title("Cubo Rotado (alrededor del eje y)")

plt.show()
```




Cubo Rotado (alrededor del eje y)
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4