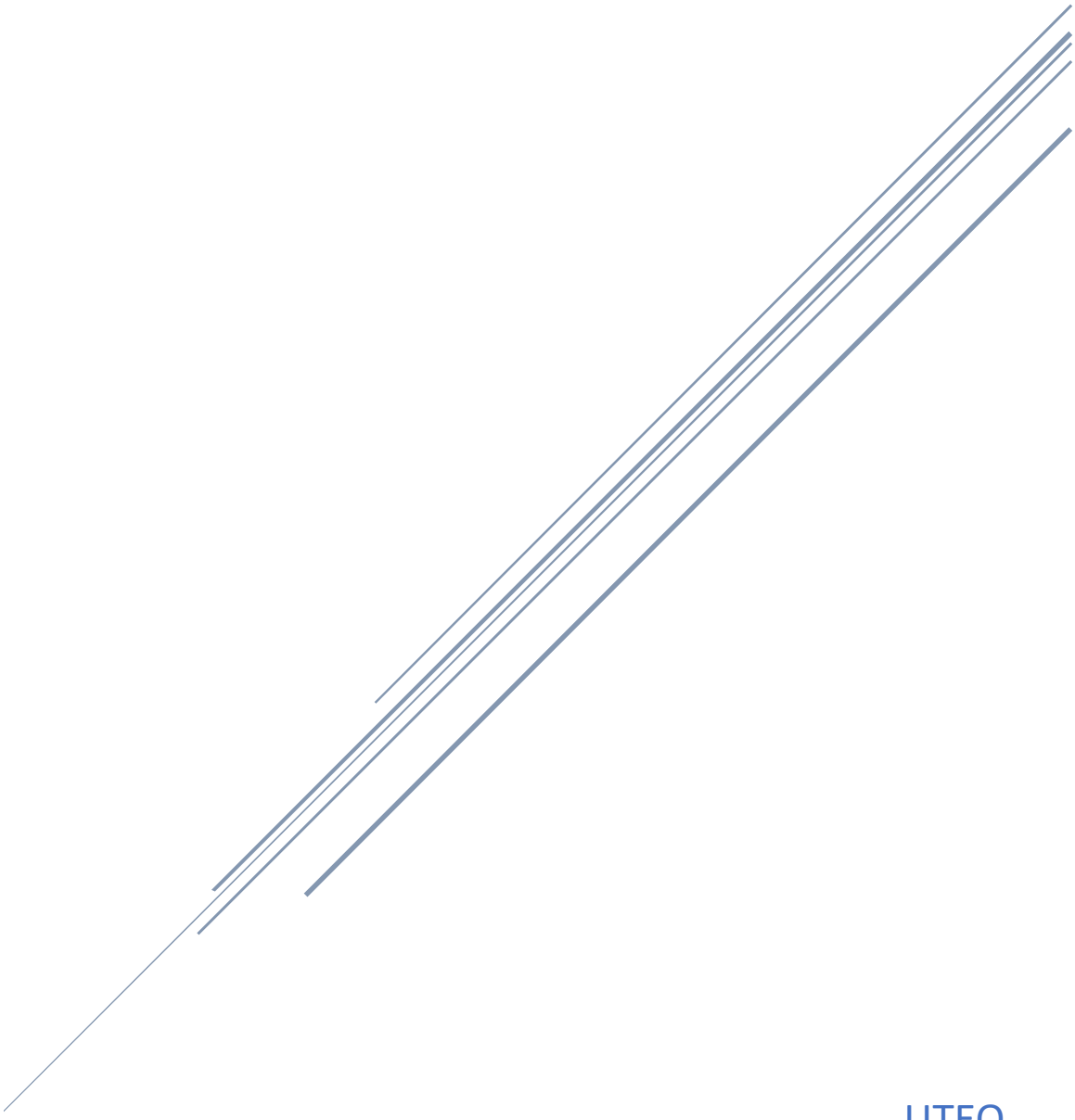
2ª. EVALUACIÓN

David Guillermo Santiago



UTEQ
13/08/2024

Índice

Análisis supervisado

Ejercicio 1

- Algoritmo seleccionado:

Regresión lineal

- Justificación del algoritmo:

La regresión lineal es apropiada cuando se presume que existe una relación lineal entre la variable independiente (en este caso, "bateos") y la variable dependiente (en este caso, "runs"). En el contexto del béisbol, es razonable suponer que la cantidad de "bateos" puede tener una relación directa con la cantidad de "runs" (carreras) que un equipo anota. La regresión lineal ayuda a modelar esta relación y a predecir la cantidad de "runs" basándose en los "bateos".

- Descripción del diseño del modelo:

Objetivo: Implementar un algoritmo de regresión para predecir la cantidad de "runs" (carreras) de un equipo de béisbol basado en la cantidad de "bateos".

Paso 1: Preparación del entorno

Primero, asegurémonos de tener las bibliotecas necesarias instaladas y listas para usar.

[1]:
```
# Importar bibliotecas necesarias
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

- **Pandas:** para manejar y manipular los datos.
- **NumPy:** para operaciones matemáticas.
- **Matplotlib:** para crear gráficos.
- **Scikit-learn:** para implementar el modelo de regresión lineal y evaluar el rendimiento.

Paso 2: Cargar y explorar los datos

```
from sklearn.metrics import mean_squared_error, r2_score
```

```
[2]: # Cargar el conjunto de datos
data = pd.read_csv(filepath_or_buffer = "/Users/david/beisbol.csv")

# Mostrar los primeros 5 registros para entender el conjunto de datos
print(data.head())
```

```
   Unnamed: 0  equipos  bateos  runs
0           0    Texas    5659   855
1           1   Boston    5710   875
2           2  Detroit    5563   787
3           3   Kansas    5672   730
4           4      St.    5532   762
```

Paso 3: Preprocesamiento de los datos

```
[3]: # Seleccionar las variables independientes y dependientes
X = data['bateos'].values.reshape(-1, 1)  # Bateos como variable independiente
y = data['runs'].values  # Runs como variable dependiente

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- **X:** Los "bateos" son nuestra característica (variable independiente).
- **y:** Las "runs" son nuestra variable a predecir (variable dependiente).
- **División de datos:** Separar los datos en conjuntos de entrenamiento (80%) y prueba (20%) para evaluar el modelo.

Paso 4: Entrenamiento del modelo

Aquí estamos creando un modelo de regresión lineal y entrenándolo con los datos de entrenamiento.

```
[4]: # Crear y entrenar el modelo de regresión lineal
model = LinearRegression()
model.fit(X_train, y_train)
```

```
[4]: LinearRegression
LinearRegression()
```

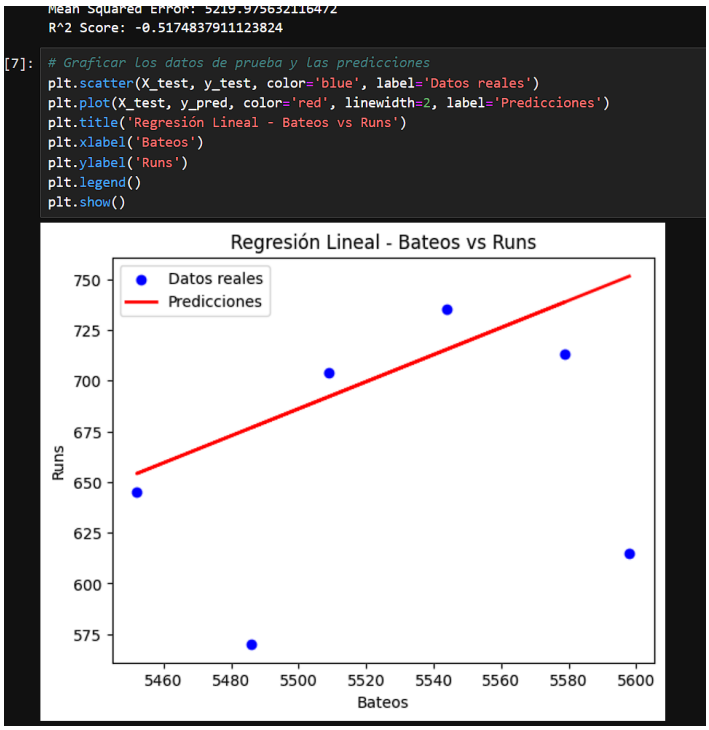
Paso 5: Evaluación del modelo

- **Error Cuadrático Medio (MSE):** Mide la media de los errores al cuadrado entre las predicciones y los valores reales. Un valor más bajo indica un mejor ajuste.
- **R^2 Score:** Indica qué tan bien los datos se ajustan al modelo (un valor más cercano a 1 indica un mejor ajuste).

```
[5]: # Predecir los valores de prueba
     y_pred = model.predict(X_test)

     # Calcular el Error Cuadrático Medio (MSE) y el Coeficiente de Determinación (R^2)
     mse = mean_squared_error(y_test, y_pred)
     r2 = r2_score(y_test, y_pred)

     print(f"Mean Squared Error: {mse}")
     print(f"R^2 Score: {r2}")

     Mean Squared Error: 5219.975632116472
     R^2 Score: -0.5174837911123824
```

Paso 6: Visualización de los resultados

Esta gráfica muestra los valores reales y las predicciones del modelo. La línea roja representa las predicciones de nuestro modelo de regresión lineal.



Ejercicio 2:

- Algoritmo Seleccionado:

Regresión Logística

- Justificación del Algoritmo:

La Regresión Logística es un algoritmo de clasificación simple y eficiente que se utiliza para predecir el resultado de una variable categórica binaria, como en este caso, donde queremos predecir si una persona tiene diabetes (1) o no (0). Es adecuado cuando la relación entre las características (variables independientes) y la variable objetivo (variable dependiente) no es lineal, pero las probabilidades de las clases se pueden modelar mediante una función logística. Además, es fácil de interpretar y se puede optimizar para mejorar el rendimiento.

- Descripción del diseño del modelo:

Paso 1: Importar las librerías necesarias

```
[1]: import numpy as np
     import pandas as pd
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
     import matplotlib.pyplot as plt
     import seaborn as sns
```

- **numpy y pandas:** Son librerías fundamentales para trabajar con datos en Python. numpy maneja arreglos numéricos y pandas facilita la manipulación de datos en forma de tablas (dataframes).
- **train_test_split:** Esta función de sklearn divide los datos en dos partes: una para entrenar el modelo y otra para probar su rendimiento.
- **StandardScaler:** Se usa para escalar o normalizar los datos, lo que significa ajustar las características para que todas estén en la misma escala. Esto es importante en muchos algoritmos de machine learning, incluida la regresión logística.
- **LogisticRegression:** Es el modelo de regresión logística que utilizaremos para clasificar los datos.
- **confusion_matrix, accuracy_score, classification_report:** Son métricas y herramientas que nos permiten evaluar el rendimiento del modelo.
- **matplotlib.pyplot y seaborn:** Estas librerías se utilizan para crear gráficos. seaborn es particularmente útil para visualizaciones estadísticas como la matriz de confusión.

Paso 2: Cargar y preparar los datos

```
[3]: # Cargar los datos
     data = pd.read_csv(filepath_or_buffer = "/Users/david/diabetes_indiana.csv")

     # Separar las variables dependientes e independientes
     X = data.iloc[:, :-1].values  # Todas las columnas excepto la última (características)
     y = data.iloc[:, -1].values   # La última columna (variable objetivo)

     # Dividir los datos en conjuntos de entrenamiento y prueba
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

     # Estandarizar los datos
     scaler = StandardScaler()
     X_train = scaler.fit_transform(X_train)
     X_test = scaler.transform(X_test)
```

- **data = pd.read_csv('diabetes_indiana.csv'):** Cargamos el archivo CSV con los datos de diabetes en un dataframe de pandas.
- **X = data.iloc[:, :-1].values**: Aquí estamos seleccionando todas las columnas excepto la última. Estas columnas representan las características que utilizaremos para hacer la predicción (por ejemplo, niveles de glucosa, presión sanguínea, etc.).
- **y = data.iloc[:, -1].values:** Seleccionamos la última columna que es la variable objetivo, es decir, si la persona tiene diabetes (1) o no (0).
- **train_test_split(X, y, test_size=0.2, random_state=42):** Dividimos los datos en dos partes: X_train y y_train para entrenar el modelo, y X_test y y_test para probarlo. El test_size=0.2 significa que el 20% de los datos se usarán para probar el modelo y el 80% para entrenarlo. random_state=42 se usa para asegurarnos de que obtengamos los mismos resultados cada vez que ejecutemos el código.
- **scaler.fit_transform(X_train):** Normalizamos (escalamos) los datos de entrenamiento para que todas las características tengan la misma importancia al entrenar el modelo.
- **scaler.transform(X_test):** También normalizamos los datos de prueba usando los mismos parámetros calculados con los datos de entrenamiento.

Paso 3: Entrenar el modelo de Regresión Logística

```
[4]: # Entrenar el modelo de regresión logística
     model = LogisticRegression(random_state=42)
     model.fit(X_train, y_train)
```

```
[4]: LogisticRegression
     LogisticRegression(random_state=42)
```

- **model = LogisticRegression(random_state=42):** Aquí creamos una instancia del modelo de regresión logística. La regresión logística es ideal para este tipo de problemas de clasificación binaria (diabetes: sí o no).
- **model.fit(X_train, y_train):** Entrenamos el modelo con los datos de entrenamiento (X_train y y_train). El modelo aprenderá de estos datos para poder hacer predicciones en nuevos datos.

Paso 4: Realizar predicciones

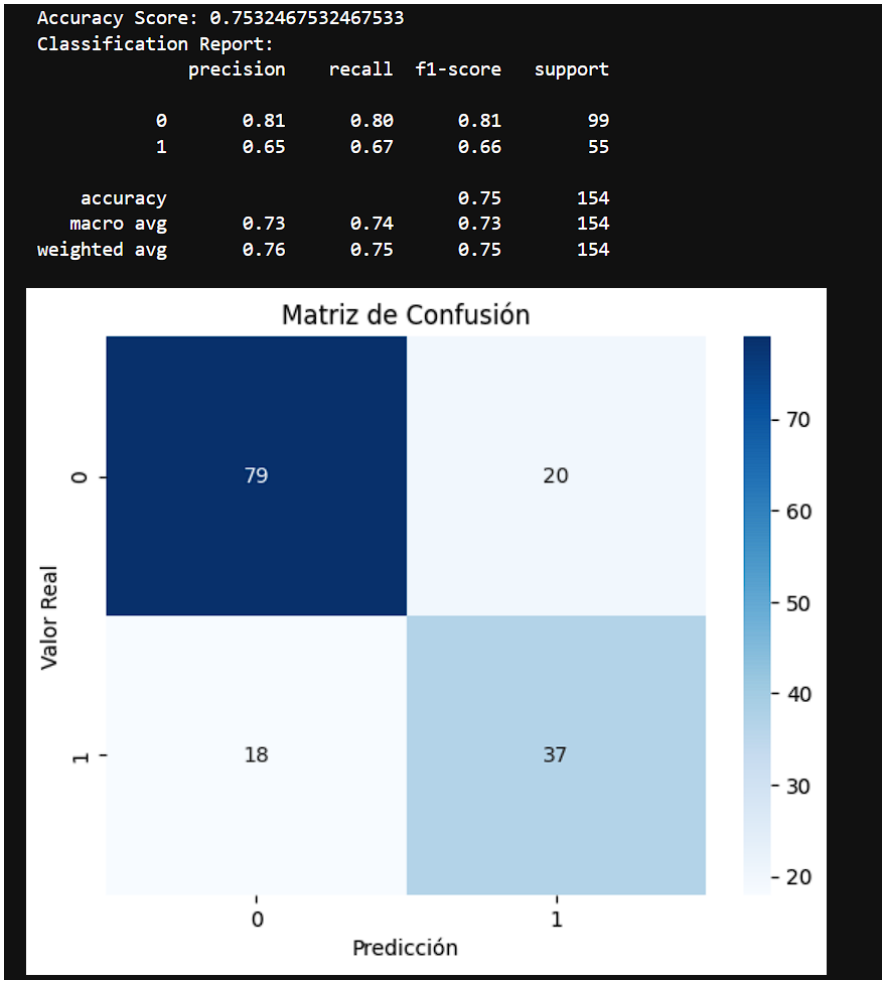
```
[5]: # Realizar predicciones
     y_pred = model.predict(X_test)
```

- **y_pred = model.predict(X_test):** Usamos el modelo entrenado para predecir si las personas en el conjunto de prueba (X_test) tienen diabetes o no. El resultado es un arreglo de predicciones (y_pred) que podemos comparar con los valores reales (y_test).

Paso 5: Evaluar el modelo

```
# Calcular la matriz de confusión
conf_matrix = confusion_matrix(y_test, y_pred)

# Calcular el accuracy score
accuracy = accuracy_score(y_test, y_pred)

# Generar un reporte de clasificación
class_report = classification_report(y_test, y_pred)

# Mostrar resultados
print(f"Accuracy Score: {accuracy}")
print("Classification Report:")
print(class_report)

# Visualizar la matriz de confusión
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title('Matriz de Confusión')
plt.xlabel('Predicción')
plt.ylabel('Valor Real')
plt.show()
```

- **confusion_matrix(y_test, y_pred):** Una matriz de confusión es una tabla que muestra el rendimiento del modelo al comparar las predicciones (y_pred) con los valores reales (y_test). Nos dice cuántos casos fueron predichos correctamente y cuántos fueron predichos incorrectamente.
- **accuracy_score(y_test, y_pred):** Calcula la precisión, que es el porcentaje de predicciones correctas del modelo.
- **classification_report(y_test, y_pred):** Genera un informe detallado que incluye métricas como la precisión, el recall y el F1-score para cada clase (0 y 1 en este caso).
- **sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues'):** Usa seaborn para crear una visualización gráfica de la matriz de confusión, lo que facilita la interpretación de los resultados.
- **plt.show():** Muestra el gráfico de la matriz de confusión en pantalla.



Precisión (Precision): La precisión para la clase ‘0’ es 0.81 y para la clase ‘1’ es 0.65. Esto indica que el modelo es más preciso al predecir la clase ‘0’.

Recuperación (Recall): La recuperación para la clase ‘0’ es 0.80 y para la clase ‘1’ es 0.67. Esto sugiere que el modelo es mejor identificando correctamente los casos de la clase ‘0’.

F1-Score: Es una medida combinada de precisión y recuperación. Para la clase ‘0’ es 0.81 y para la clase ‘1’ es 0.66, lo que indica un mejor rendimiento general para la clase ‘0’.

Exactitud (Accuracy): La exactitud global del modelo es 0.75, lo que significa que el 75% de las predicciones fueron correctas.

Matriz de Confusión:

- **Verdaderos Negativos (True Negatives):** 79, lo que indica que el modelo identificó correctamente 79 casos de la clase ‘0’.
- **Falsos Positivos (False Positives):** 20, lo que significa que el modelo incorrectamente predijo 20 casos como clase ‘1’ cuando eran clase ‘0’.
- **Falsos Negativos (False Negatives):** 37, lo que muestra que el modelo falló en identificar 37 casos de la clase ‘1’.
- **Verdaderos Positivos (True Positives):** 18, lo que indica que el modelo identificó correctamente 18 casos de la clase ‘1’.

Evaluación y Optimización del Modelo

Accuracy Score: La precisión nos dice qué proporción de las predicciones fueron correctas. Es una métrica clave en clasificación, especialmente cuando las clases están balanceadas.

Matriz de Confusión: Nos permite ver cuántos casos positivos fueron clasificados correctamente (verdaderos positivos) y cuántos fueron clasificados incorrectamente (falsos positivos y falsos negativos). Esto nos ayuda a entender mejor el rendimiento del modelo.

Reporte de Clasificación: Incluye métricas como la precisión, recall y F1-score para cada clase, lo que nos da una visión más completa del rendimiento del modelo en ambos casos (positivos y negativos).

Posibles Mejoras:

Balanceo de Clases: Si las clases están desbalanceadas (es decir, una clase es mucho más frecuente que la otra), podríamos considerar técnicas de balanceo, como oversampling de la clase minoritaria o undersampling de la clase mayoritaria.

Ajuste de Hiperparámetros: Podemos usar técnicas como la búsqueda en cuadrícula (GridSearchCV) para encontrar los mejores hiperparámetros del modelo.

Análisis no Supervisado

Ejercicio 3:

- Algoritmo seleccionado:

K-means clustering

- Justificación del algoritmo:

K-means clustering es un algoritmo de agrupación no supervisado que busca dividir un conjunto de datos en k grupos (o clusters) basados en características similares. Se elige K-means por las siguientes razones:

- **Simplicidad:** Es fácil de entender y de implementar.
- **Eficiencia:** Funciona bien en la práctica con grandes conjuntos de datos.
- **Flexibilidad:** Puede adaptarse a diferentes formas de datos y números de clusters.

Descripción del Diseño del Modelo:

1.- Preprocesamiento de Datos:

- **Normalización:** Escalar los datos para que todas las características tengan la misma importancia.

2.- Aplicación del Algoritmo K-means:

- **Número de Clusters:** Determinar el número óptimo de clusters usando el método del codo (elbow method).
- **Entrenamiento:** Aplicar el algoritmo K-means para agrupar los datos en el número seleccionado de clusters.

3.- Evaluación:

- Aunque K-means no produce una matriz de confusión o métricas similares directamente, se puede evaluar visualmente la calidad de los clusters y la variación interna.

4.- Visualización:

- **Gráficas:** Visualizar los clusters para interpretar cómo se agrupan los datos.

Código con Explicaciones Detalladas

Paso 1: importar librerias

```
]: # Importar las librerías necesarias
   import numpy as np
   import pandas as pd
   from sklearn.preprocessing import StandardScaler
   from sklearn.cluster import KMeans
   import matplotlib.pyplot as plt
   import seaborn as sns
```

- numpy: Biblioteca para cálculos numéricos y manejo de arrays.
- pandas: Biblioteca para manipulación y análisis de datos en estructuras de DataFrame.
- StandardScaler: Clase de sklearn para estandarizar características (normalización).
- KMeans: Clase de sklearn para realizar el clustering con el algoritmo K-means.
- matplotlib.pyplot: Biblioteca para crear gráficos.
- seaborn: Biblioteca para crear gráficos estadísticos con una interfaz de alto nivel.

Paso 2: cargar datos

```
]: # Cargar los datos
   data = pd.read_csv(filepath_or_buffer = "/Users/david/samsung.csv")
```

Paso 3: Preprocesar los datos

- data[['Close', 'Volume']]: Selecciona las columnas 'Close' y 'Volume' del DataFrame para usarlas como características para el clustering.
- .values: Convierte estas columnas en un array NumPy, que es el formato requerido para la mayoría de los algoritmos de sklearn.

```
: # Seleccionar las variables independientes y dependientes
  X = data['bateos'].values.reshape(-1, 1)  # Bateos como variable independiente
  y = data['runs'].values  # Runs como variable dependiente

  # Dividir los datos en conjuntos de entrenamiento y prueba
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Paso 4: Estandarizacion de datos

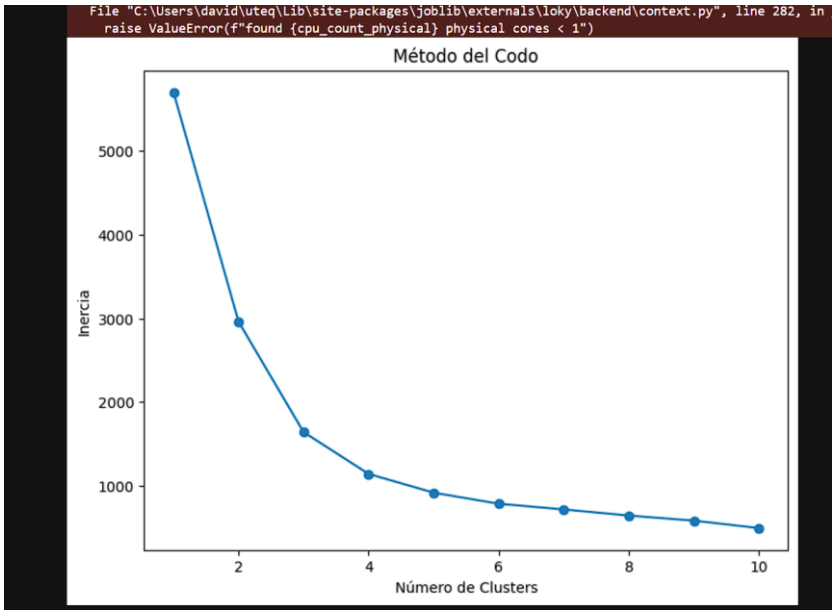
- StandardScaler(): Crea un objeto scaler para estandarizar las características.
- scaler.fit_transform(X): Ajusta el scaler a los datos y luego transforma los datos. Esto asegura que cada característica tenga media 0 y desviación estándar 1, lo que ayuda a mejorar el rendimiento del algoritmo K-means.

```
]: # Estandarizar los datos
   scaler = StandardScaler()
   X_scaled = scaler.fit_transform(X)
```

Paso 5: Determinación del Número Óptimo de Clusters

- inertia = []: Lista para almacenar la inercia para cada valor de k.
- for k in range(1, 11): Itera sobre un rango de números de clusters de 1 a 10.
- KMeans(n_clusters=k, random_state=42): Crea una instancia del modelo K-means con k clusters.
- kmeans.fit(X_scaled): Ajusta el modelo a los datos estandarizados.
- inertia.append(kmeans.inertia_): Añade la inercia (suma de distancias al cuadrado de los puntos de datos a su centro de cluster) a la lista inertia.
- plt.plot(...): Grafica la inercia frente al número de clusters para encontrar el punto donde la inercia deja de disminuir significativamente (el codo).

```
]: # Determinar el número óptimo de clusters usando el método del codo
   inertia = []
   for k in range(1, 11):  # Probamos con k de 1 a 10
       kmeans = KMeans(n_clusters=k, random_state=42)
       kmeans.fit(X_scaled)
       inertia.append(kmeans.inertia_)

   # Graficar el método del codo
   plt.figure(figsize=(8, 6))
   plt.plot(range(1, 11), inertia, marker='o')
   plt.title('Método del Codo')
   plt.xlabel('Número de Clusters')
   plt.ylabel('Inercia')
   plt.show()
```



6. Aplicación de K-means con el Número Óptimo de Clusters

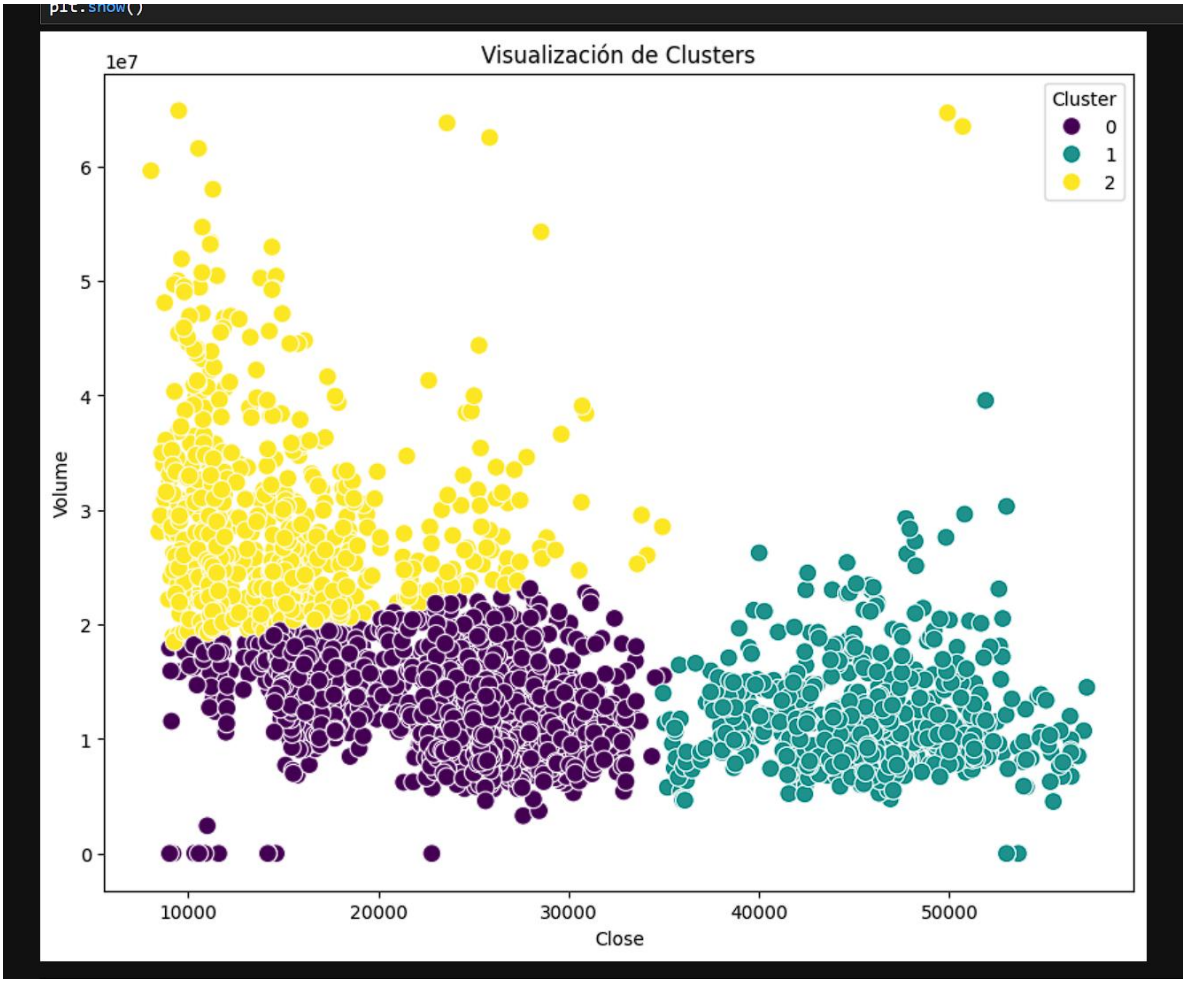
```
7]: # Elegir el número óptimo de clusters (suponiendo que se elige 3 basándonos en el codo)
    optimal_k = 3

    # Aplicar K-means con el número óptimo de clusters
    kmeans = KMeans(n_clusters=optimal_k, random_state=42)
    clusters = kmeans.fit_predict(X_scaled)
```

- optimal_k = 3: Se elige el número óptimo de clusters basado en la gráfica del codo.
- KMeans(n_clusters=optimal_k, random_state=42): Crea una instancia del modelo K-means con el número óptimo de clusters.
- kmeans.fit_predict(X_scaled): Ajusta el modelo a los datos estandarizados y predice el cluster para cada punto de datos. Los resultados se almacenan en clusters.

7. Añadir Clusters al DataFrame y Visualización

```python
# Añadir los clusters al dataframe original
data['Cluster'] = clusters

# Visualizar los clusters
plt.figure(figsize=(10, 8))
sns.scatterplot(x=data['Close'], y=data['Volume'], hue=data['Cluster'], palette='viridis', s=100)
plt.title('Visualización de Clusters')
plt.xlabel('Close')
plt.ylabel('Volume')
plt.legend(title='Cluster')
plt.show()
```

```
plt.show()
```



Ejercicio 4

Algoritmo seleccionado

Análisis de Componentes Principales (PCA)

Justificación del algoritmo

Análisis de Componentes Principales (PCA) es una técnica común para la reducción de dimensionalidad. Su objetivo es transformar los datos a un nuevo sistema de coordenadas, donde las dimensiones (componentes principales) son ordenadas según la varianza de los datos que explican. Esto es útil para simplificar el análisis, reducir el ruido y mejorar la eficiencia computacional sin perder demasiada información importante.

Descripción del diseño del modelo:

1.- Importación de Librerías

```
:  import numpy as np
   import pandas as pd
   from sklearn.decomposition import PCA
   from sklearn.preprocessing import StandardScaler
   import matplotlib.pyplot as plt
```

2.- Cargar los datos

```
]:  # Cargar los datos
    data = pd.read_csv(filepath_or_buffer = "/Users/david/comprar_alquilar.csv")
```

3.- Seleccionar las Características

- data[['ingresos', ...]]: Selecciona las columnas de características para el análisis de PCA.
- .values: Convierte el DataFrame a un array NumPy.

```
4]:  X = data[['ingresos', 'gastos_comunes', 'pago_coche', 'gastos_otros', 'ahorros', 'vivienda', 'estado_civil', 'hijos', 'trabajo']].values
```

4.- Estandarizar los Datos

- StandardScaler(): Crea un objeto scaler para estandarizar los datos.
- scaler.fit_transform(X): Ajusta el scaler a los datos y los transforma para tener media 0 y desviación estándar 1.

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

5.- Crear el Modelo PCA

```
pca = PCA(n_components=2)  # Reducir a 2 dimensiones para visualización
X_pca = pca.fit_transform(X_scaled)
```

- PCA(n_components=2): Crea un objeto PCA para reducir los datos a 2 componentes principales (dimensiones).
- pca.fit_transform(X_scaled): Ajusta el modelo PCA a los datos estandarizados y transforma los datos.

6.- Mostrar la Varianza Explicada

```
print(f"Varianza explicada por el primer componente: {pca.explained_variance_ratio_[0]:.2f}")
print(f"Varianza explicada por el segundo componente: {pca.explained_variance_ratio_[1]:.2f}")
```

```
Varianza explicada por el primer componente: 0.30
Varianza explicada por el segundo componente: 0.23
```

- pca.explained_variance_ratio_: Muestra la cantidad de varianza explicada por cada componente principal. Estos valores indican cuánta información de los datos originales se conserva en cada componente.

7.- Visualizar los datos reducidos

```
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=data['comprar'], cmap='viridis', edgecolor='k', s=50)
plt.colorbar(label='Comprar')
plt.title('PCA: Datos Reducidos a 2 Dimensiones')
plt.xlabel('Componente Principal 1')
plt.ylabel('Componente Principal 2')
plt.show()
```

- plt.scatter(...): Crea un gráfico de dispersión de los datos reducidos a dos dimensiones.
- c=data['comprar']: Colorea los puntos según la variable 'comprar', lo que ayuda a ver cómo se agrupan los datos en el nuevo espacio reducido.
- plt.colorbar(label='Comprar'): Añade una barra de colores para indicar los valores de 'comprar'.