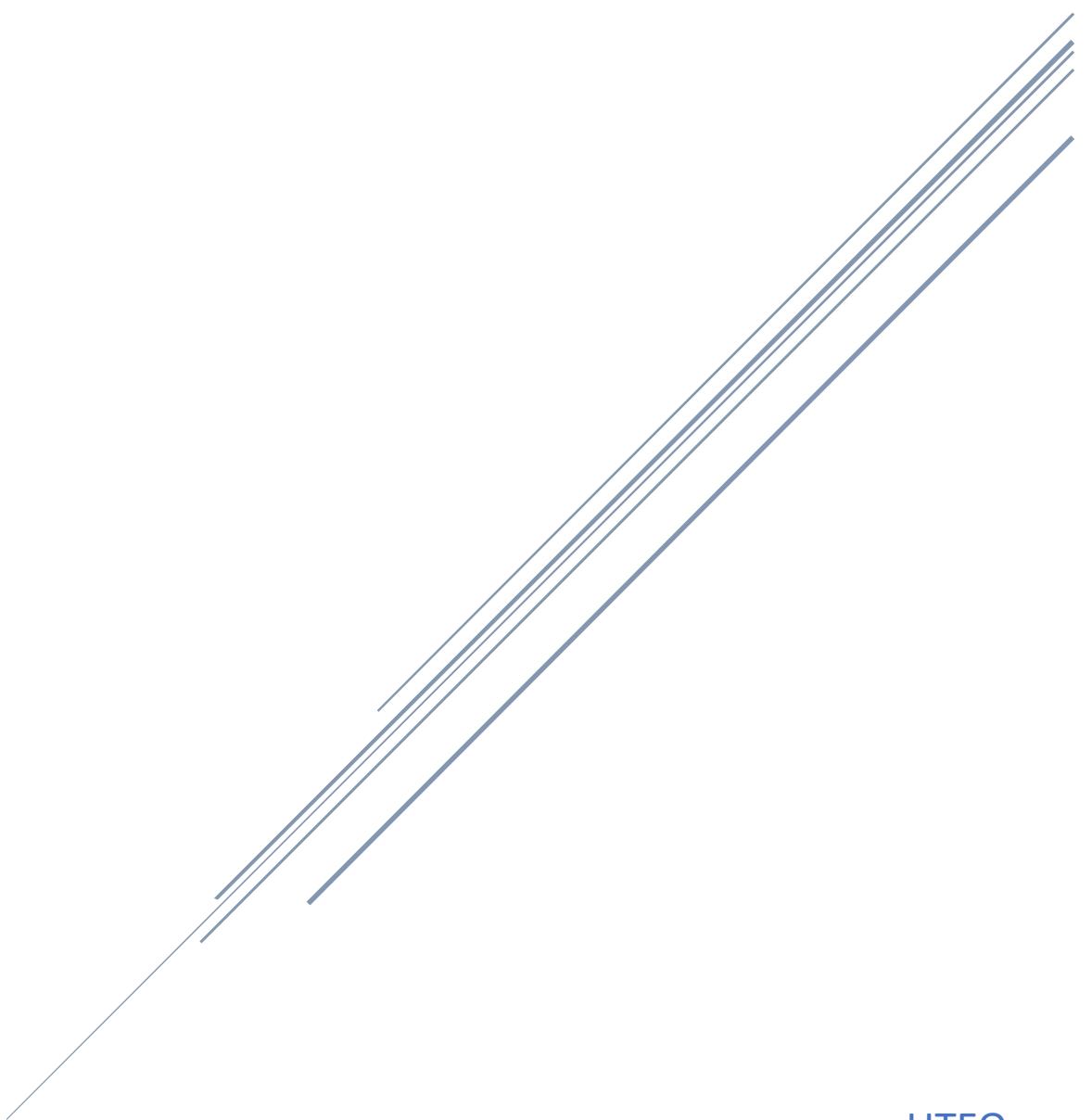


REMEDIAL

David Guillermo Santiago



ÍNDICE

Ejercicio 1 - Corregido	2
Justificación del algoritmo	2
Descripción del diseño del modelo	2
11.- Visualizar los datos.....	7
14.- Visualización de los datos optimizados.....	9
Interpretación:	10
Enlace hacia el modelo obtenido	10
Ejercicio 2 - Corregido	11
Justificación del algoritmo	11
Descripción del diseño del modelo	11
Evaluación y optimización del modelo.....	12
Gráfica personalizada e interpretación de resultados	15
Enlace hacia el modelo obtenido	16
Ejercicio 3 - Corregido	17
Justificación del algoritmo	17
Descripción del diseño del modelo	17
Enlace hacia el modelo obtenido	22
Ejercicio 4 - Corregido	23
Justificación del algoritmo	23
Descripción del diseño del modelo	23
Enlace hacia el modelo obtenido	30
Ejercicio 5 – Remedial	31
Justificación del algoritmo.	31
Descripción del diseño del modelo.....	31
Enlace al modelo obtenido.....	36
Ejercicio 6 - Remedial	37
Justificación del algoritmo	37
Descripción del Diseño del Modelo.....	37
Enlace hacia el modelo obtenido	40

Ejercicio 1 - Corregido

Justificación del algoritmo

Algoritmo seleccionado:

Para este ejercicio se seleccionó el algoritmo de **Regresión Lineal Simple**. Este algoritmo es adecuado para problemas de predicción donde existe una relación lineal entre la variable independiente (predictora) y la variable dependiente (respuesta). En nuestro caso, estamos interesados en predecir el número de carreras (runs) basado en el número de bateos.

Justificación:

- **Simplicidad:** La regresión lineal es fácil de entender e implementar, lo que la convierte en una buena opción para comenzar con análisis predictivos.
- **Relación Lineal:** Según la naturaleza del problema y los datos, esperamos que exista una relación lineal entre el número de bateos y el número de carreras, lo que hace que este algoritmo sea una elección natural.
- **Rendimiento:** Es eficiente en términos de tiempo de cómputo para conjuntos de datos pequeños o medianos, lo cual es ideal para nuestro escenario.

Descripción del diseño del modelo

1.- Cargar las librerías necesarias

Estas librerías son fundamentales para el análisis de datos y el modelado. Se utilizan para cargar, manipular y visualizar datos, además de implementar el modelo de regresión lineal y evaluarlo.

```
import pandas as pd # Para la manipulación de datos
import numpy as np # Para operaciones numéricas
import matplotlib.pyplot as plt # Para la visualización de gráficos
import seaborn as sns # Para gráficos estadísticos
from sklearn.linear_model import LinearRegression # Para el modelo de regresión lineal
from sklearn.model_selection import train_test_split # Para dividir los datos en entrenamiento y prueba
from sklearn.model_selection import RepeatedKFold, GridSearchCV # Para la optimización del modelo
from sklearn import metrics # Para la evaluación del modelo

# Configuración para mostrar gráficos en línea
%matplotlib inline
```

2.- Cargar el conjunto de datos

```
# Cargar el archivo CSV en un DataFrame
datos = pd.read_csv("C:/Users/david/segundaEvaluacion/Conjunto de Datos/beisbol.csv", sep=',')
```

3.- Mostrar los primeros registros

# Mostrar los primeros registros				
datos.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

4.- Describir y explorar los datos

La función **describe()** genera estadísticas descriptivas que resumen la tendencia central, la dispersión y la forma de la distribución de un conjunto de datos:

- **count:** Número de observaciones no nulas.
- **mean:** Promedio de los valores.
- **std:** Desviación estándar, que mide la dispersión de los datos respecto a la media.
- **min:** Valor mínimo.
- **25%:** Primer cuartil, el 25% de los datos están por debajo de este valor.
- **50% (median):** Mediana, el valor central que divide los datos en dos mitades.
- **75%:** Tercer cuartil, el 75% de los datos están por debajo de este valor.
- **max:** Valor máximo.

# Resumen estadístico del conjunto de datos			
datos.describe()			
	Unnamed: 0	bateos	runs
count	30.000000	30.000000	30.000000
mean	14.500000	5523.500000	693.600000
std	8.803408	79.873067	82.479088
min	0.000000	5417.000000	556.000000
25%	7.250000	5448.250000	629.000000
50%	14.500000	5515.500000	705.500000
75%	21.750000	5575.000000	734.000000
max	29.000000	5710.000000	875.000000

El método shape nos permite conocer la cantidad de filas y columnas:

```
# Verificar La estructura del DataFrame  
datos.shape
```

```
(30, 4)
```

30 filas y 4 columnas

5.- Separar las variables independientes y dependientes

Aquí sepamos las variables independientes (X) y dependientes (y). La variable bateos es nuestro predictor, y runs es la variable que queremos predecir.

```
# Seleccionar La variable independiente 'bateos'  
X = datos[['bateos']]  
# Seleccionar La variable dependiente 'runs'  
y = datos['runs']
```

6.- Dividir el conjunto de datos en entrenamiento y prueba

Para entrenar y evaluar el modelo de manera efectiva, dividimos los datos en dos conjuntos: entrenamiento (80%) y prueba (20%). Esto garantiza que el modelo se entrene con una parte de los datos y se evalúe con otra parte no vista durante el entrenamiento.

```
# Dividir Los datos en conjuntos de entrenamiento (80%) y prueba (20%)  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

7.- Crear y entrenar el modelo de regresión lineal

```
# Crear el modelo de regresión Lineal  
model = LinearRegression()  
  
# Entrenar el modelo con los datos de entrenamiento  
model.fit(X_train, y_train)  
  
▼ LinearRegression ⓘ ⓘ  
LinearRegression()
```

8.- Obtener el intercepto y la pendiente del modelo

- **model.intercept_**: Obtiene el intercepto de la línea de regresión, que es el punto en el que la línea cruza el eje Y. Este valor representa el número de carreras predichas cuando la cantidad de bateos es cero.
- **model.coef_**: Obtiene la pendiente de la línea de regresión, que indica cuánto cambian las carreras (runs) por cada unidad adicional de bateos (bateos).

```
# Obtener el intercepto y la pendiente del modelo
print("Intercepto:", model.intercept_)
print("Pendiente:", model.coef_)
```

Intercepto: -2981.517212576021

Pendiente: [0.66685087]

Un **intercepto** de -2981.52 sugiere que, si no hubiera ningún bateo (es decir, bateos = 0), el modelo predice que el número de carreras (runs) sería -2981.52. Aunque este valor no tiene un sentido práctico en un contexto real (no se pueden tener carreras negativas), su importancia radica en cómo ajusta la línea de regresión para los datos reales.

La **pendiente** representa el cambio en runs por cada unidad adicional de bateos. En este caso, una pendiente de 0.667 significa que, por cada bateo adicional, se espera que el número de carreras aumente en 0.667. Un incremento de 1000 bateos resultaría en un aumento esperado de aproximadamente 667 carreras.

9.- Hacer predicciones con los datos de prueba

Este paso nos permite evaluar cómo de bien el modelo generaliza a nuevos datos, es decir, datos que no ha visto durante el entrenamiento. Esto es crucial para medir la efectividad del modelo en situaciones reales.

Se crea un nuevo DataFrame (datos2) que contiene dos columnas: Actual con los valores reales de runs en el conjunto de datos de prueba (y_test), y Predicted con los valores de runs que el modelo predijo (y_pred).

```

: # Hacer predicciones con los datos de prueba
y_pred = model.predict(X_test)

# Crear un DataFrame para comparar los valores reales y predichos
datos2 = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print(datos2)

   Actual  Predicted
27      570    676.826633
15      713    738.843764
23      645    654.153704
17      704    692.164203
8       735    715.503983
9       615    751.513930

```

En general, el modelo parece realizar predicciones razonablemente precisas para la mayoría de los casos, con diferencias que, en su mayoría, son relativamente pequeñas. Sin embargo, hay casos donde el modelo sobreestima o subestima los valores reales, lo que indica que podría beneficiarse de ajustes adicionales.

10.- Evaluación

En este bloque de código, se utilizan tres métricas comunes para evaluar el rendimiento del modelo de regresión lineal. Estas métricas son cruciales para entender la precisión de las predicciones y cómo se comporta el modelo en términos de errores.

```

# Calcular el error absoluto medio, error cuadrático medio y la raíz del error cuadrático medio
print('Error Absoluto Medio:', metrics.mean_absolute_error(y_test, y_pred))
print('Error Cuadrático Medio:', metrics.mean_squared_error(y_test, y_pred))
print('Raíz del Error Cuadrático Medio:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))

Error Absoluto Medio: 51.61164068147824
Error Cuadrático Medio: 5219.975632116472
Raíz del Error Cuadrático Medio: 72.24939883567525

```

- **Error Absoluto Meido (MAE): 51.61**

En promedio, las predicciones del modelo se desvían del valor real en aproximadamente 51.61 unidades. Esto indica que, en términos absolutos, el modelo comete un error de alrededor de 51.61 runs en cada predicción.

- **Error Cuadrático Medio (MSE): 5219.98**

Este valor muestra el error promedio elevado al cuadrado. Un MSE de 5219.98 indica que algunos errores del modelo son bastante significativos, ya que el valor es elevado. Esto sugiere que, aunque el modelo funciona bien en general, hay algunas predicciones que se desvían considerablemente de los valores reales.

- **Raíz del Error Cuadrático Medio (RMSE): 72.25**

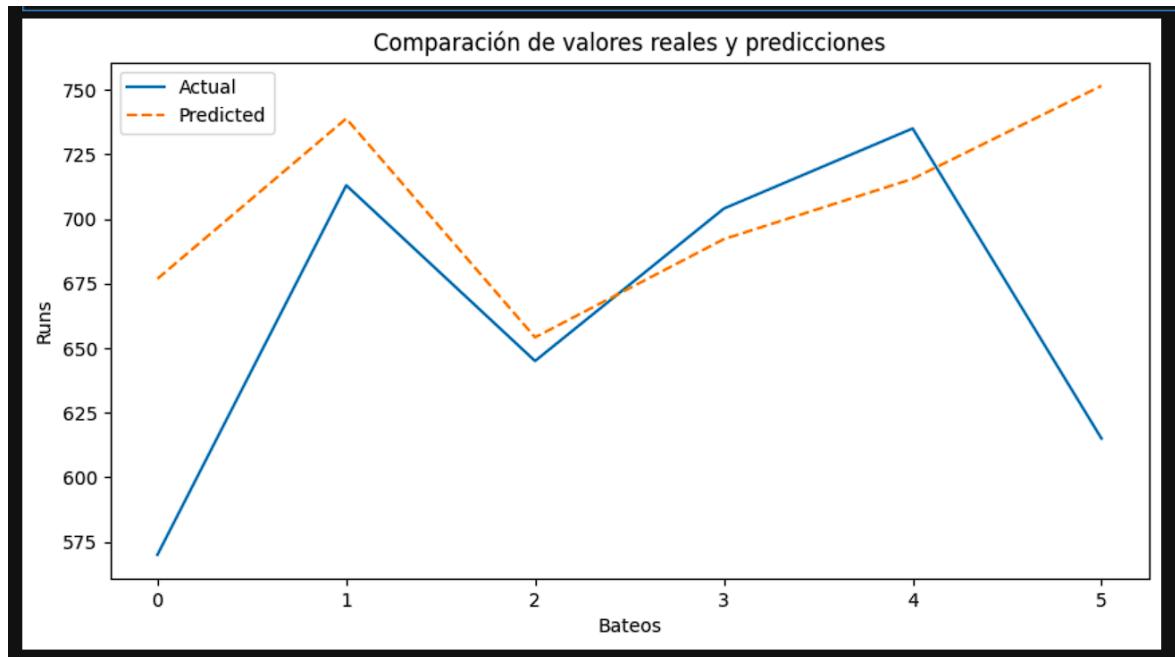
Interpretación: La RMSE de 72.25 indica que, en promedio, las predicciones del modelo están a 72.25 runs de los valores reales. Al estar en la misma unidad que los runs, este

valor es más fácil de interpretar y refleja un error moderado en las predicciones del modelo. Aunque el modelo es razonablemente preciso, hay espacio para mejorar su rendimiento.

11.- Visualizar los datos

Este gráfico muestra la comparación entre los valores reales y las predicciones del modelo. Las líneas sólidas representan los valores reales, mientras que las líneas discontinuas muestran las predicciones.

```
# Visualización de Los resultados
plt.figure(figsize=(10,5))
plt.plot(datos2['Actual'].values, label='Actual')
plt.plot(datos2['Predicted'].values, label='Predicted', linestyle='--')
plt.xlabel('Bateos')
plt.ylabel('Runs')
plt.legend()
plt.title('Comparación de valores reales y predicciones')
plt.show()
```



El modelo parece seguir la tendencia general de los datos reales, lo cual es positivo. Sin embargo, en algunos puntos específicos, las predicciones difieren notablemente de los valores reales. Por ejemplo, en el punto de bateo 1, la predicción sobreestima considerablemente los valores reales, y en el punto de bateo 5, el modelo no captura la caída significativa que se observa en los valores reales.

12.- Optimización

Este bloque de código tiene como objetivo optimizar el rendimiento del modelo de regresión lineal mediante la búsqueda de los mejores hiperparámetros. Se utiliza la técnica de validación cruzada repetida y la búsqueda en cuadrícula (GridSearchCV) para encontrar la combinación óptima de hiperparámetros que minimicen el error absoluto medio negativo.

- Definir la validación cruzada repetida (RepeatedKFold):

```
# Definir La validación cruzada
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
```

- Definir el espacio de búsqueda de hiperparámetros

```
# Definir el espacio de búsqueda de hiperparámetros
space = {
    'fit_intercept': [True, False],
    'copy_X': [True, False],
    'positive': [True, False]
}
```

- Búsqueda de hiperparámetros óptimos utilizando GridSearchCV

```
# Búsqueda de hiperparámetros óptimos utilizando GridSearchCV
search = GridSearchCV(model, space, scoring='neg_mean_absolute_error', n_jobs=-1, cv=cv)
result = search.fit(X, y)
```

- Mostrar los mejores hiperparámetros y puntuación

```
# Mostrar Los mejores hiperparámetros y puntuación
print('Mejor puntuación:', result.best_score_)
print('Mejores hiperparámetros:', result.best_params_)

Mejor puntuación: -54.90077212237434
Mejores hiperparámetros: {'copy_X': True, 'fit_intercept': True, 'positive': False}
```

13.- Reentrenar el modelo con los mejores hiperparámetros

Después de encontrar los mejores hiperparámetros, reentrenamos el modelo y realizamos predicciones.

```
# Crear un modelo con los mejores hiperparámetros
model_optimized = LinearRegression(**result.best_params_)

# Entrenar el modelo optimizado
model_optimized.fit(X_train, y_train)

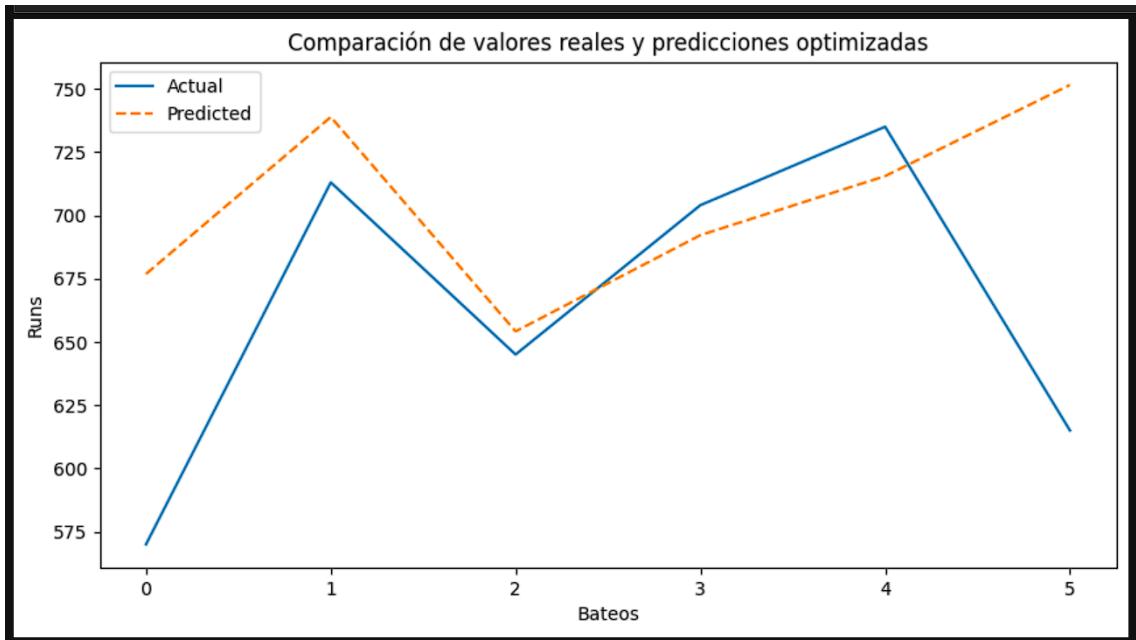
# Predecir con el modelo optimizado
y_pred_optimized = model_optimized.predict(X_test)

# Comparar Los resultados optimizados
datos2_optimized = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred_optimized})
print(datos2_optimized.head())

   Actual    Predicted
27      570  676.826633
15      713  738.843764
23      645  654.153704
17      704  692.164203
8       735  715.503983
```

14.- Visualización de los datos optimizados

```
# Visualización de los resultados optimizados
plt.figure(figsize=(10,5))
plt.plot(datos2_optimized['Actual'].values, label='Actual')
plt.plot(datos2_optimized['Predicted'].values, label='Predicted', linestyle='--')
plt.legend()
plt.xlabel('Bateos')
plt.ylabel('Runs')
plt.title('Comparación de valores reales y predicciones optimizadas')
plt.show()
```



Interpretación:

A pesar de haber realizado la optimización de hiperparámetros, las predicciones del modelo no mejoraron de manera significativa en comparación con los resultados originales. Los valores predichos después de la optimización siguen siendo prácticamente los mismos que antes.

Enlace hacia el modelo obtenido

https://github.com/DavidGuillermoSantiago/evaluacion/blob/main/ejercicio1_corregido.ipynb

Ejercicio 2 - Corregido

Justificación del algoritmo

En este caso, se seleccionó el Árbol de Decisión como algoritmo de clasificación. Este algoritmo es intuitivo y fácil de interpretar, ya que representa las decisiones en forma de árbol, donde cada nodo corresponde a una condición sobre los datos. El Árbol de Decisión es particularmente útil cuando se busca una visualización clara del proceso de decisión y es una buena opción para conjuntos de datos con características tanto categóricas como continuas.

Descripción del diseño del modelo

1.- Importar las bibliotecas necesarias

```
import pandas as pd # Para manipulación de datos
import matplotlib.pyplot as plt # Para visualización de gráficos
from sklearn.model_selection import train_test_split, GridSearchCV, RepeatedStratifiedKFold # Para dividir datos y optimización de hiperparámetros
from sklearn.tree import DecisionTreeClassifier, plot_tree # Algoritmo de clasificación y visualización
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report # Métricas de evaluación del modelo
```

2.- Cargar y explorar los datos

```
# Cargar el conjunto de datos
datos = pd.read_csv("C:/Users/david/segundaEvaluacion/Conjunto de Datos/diabetes_indiana.csv")

# Mostrar las primeras filas del conjunto de datos
print(datos.head())
```

	Unnamed: 0	0	1	2	3	4	5	6	7	8
0	0	6	148	72	35	0	33.6	0.627	50	1
1	1	1	85	66	29	0	26.6	0.351	31	0
2	2	8	183	64	0	0	23.3	0.672	32	1
3	3	1	89	66	23	94	28.1	0.167	21	0
4	4	0	137	40	35	168	43.1	2.288	33	1

3.- Separar características (X) y etiquetas (y)

En esta sección, se convierten los datos en un arreglo NumPy para facilitar la manipulación. X contiene las características, que son las columnas 1 a la penúltima, mientras que y contiene las etiquetas, que es la última columna del conjunto de datos. Se imprime la forma de X y y para verificar que la separación se realizó correctamente.

```
# Convertir los datos a un arreglo de NumPy y separar características y etiquetas
data = datos.values
X, y = data[:, :-1], data[:, -1]

# Mostrar dimensiones de los datos
print(f"Características (X): {X.shape}, Etiquetas (y): {y.shape}")

Características (X): (768, 8), Etiquetas (y): (768,)
```

4.- División de datos en conjuntos de entrenamiento y prueba

```
# Dividir los datos en conjuntos de entrenamiento (80%) y prueba (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

5.- Definir y entrenar el modelo de Árbol de Decisión

Se define un modelo de Árbol de Decisión utilizando la clase DecisionTreeClassifier sin especificar hiperparámetros. Esto permite obtener una línea base de rendimiento del modelo. Luego, se entrena el modelo con el conjunto de entrenamiento.

```
# Definir el modelo de Árbol de Decisión
modelo = DecisionTreeClassifier()

# Entrenar el modelo con el conjunto de entrenamiento
modelo.fit(X_train, y_train)
```

▼ **DecisionTreeClassifier** ⓘ ⓘ

DecisionTreeClassifier()

Evaluación y optimización del modelo

6.- Evaluar el modelo inicial con precisión y matriz de confusión

Después de entrenar el modelo, se hacen predicciones en el conjunto de prueba y se calcula la precisión del modelo, que indica el porcentaje de predicciones correctas.

```
# Hacer predicciones en el conjunto de prueba
predicciones = modelo.predict(X_test)

# Calcular la precisión del modelo
precision_inicial = accuracy_score(y_test, predicciones)
print(f"Precisión inicial: {precision_inicial}")

# Mostrar la matriz de confusión
matriz_confusion = confusion_matrix(y_test, predicciones)
print("Matriz de confusión:\n", matriz_confusion)
```

Precisión inicial: 0.7467532467532467

Matriz de confusión:

```
[[76 23]
 [16 39]]
```

La **precisión inicial** de 0.7467 indica que el modelo tiene una precisión del 74.67% en el conjunto de prueba. En otras palabras, el modelo predijo correctamente el 74.67% de las instancias del conjunto de prueba.

La matriz de confusión ofrece una visión más detallada del rendimiento del modelo al mostrar el número de verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos:

- **76 (Verdaderos negativos):** El modelo predijo correctamente que 76 personas no tienen diabetes.
- **23 (Falsos positivos):** El modelo predijo incorrectamente que 23 personas tienen diabetes cuando en realidad no la tienen. Este es un tipo de error que podría generar preocupación, ya que se les podría someter a pruebas o tratamientos innecesarios.
- **16 (Falsos negativos):** El modelo predijo incorrectamente que 16 personas no tienen diabetes cuando en realidad sí la tienen. Este tipo de error es particularmente preocupante, ya que las personas que realmente tienen diabetes podrían no recibir el tratamiento que necesitan.
- **39 (Verdaderos positivos):** El modelo predijo correctamente que 39 personas tienen diabetes.

7.- Búsqueda de hiperparámetros óptimos utilizando GridSearchCV

Se utiliza GridSearchCV para realizar una búsqueda exhaustiva de los mejores hiperparámetros. RepeatedStratifiedKFold realiza una validación cruzada estratificada, repitiendo el proceso para asegurar resultados más robustos. La búsqueda evalúa diferentes combinaciones de criterios de división (gini y entropy), profundidades máximas, y tamaños mínimos de muestras en las hojas y en las divisiones. El resultado final muestra los mejores hiperparámetros y su puntuación asociada.

```
# Configuración de la validación cruzada repetida
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

# Definir el espacio de búsqueda de hiperparámetros
espacio = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 5, 7, 10, 15],
    'min_samples_leaf': [3, 5, 10, 15, 20],
    'min_samples_split': [8, 10, 12, 16, 20]
}

# Implementación de GridSearchCV para encontrar los mejores hiperparámetros
search = GridSearchCV(modelo, espacio, scoring='accuracy', n_jobs=-1, cv=cv)
resultado = search.fit(X_train, y_train)

# Mostrar los mejores hiperparámetros y la mejor puntuación obtenida
print(f"Mejor puntuación: {resultado.best_score_}")
print(f"Mejores hiperparámetros: {resultado.best_params_}")

Mejor puntuación: 0.7677331218050413
Mejores hiperparámetros: {'criterion': 'entropy', 'max_depth': 7, 'min_samples_leaf': 20, 'min_samples_split': 10}
```

La mejor puntuación de 0.7677 indica que, tras la optimización de hiperparámetros, el modelo alcanzó una precisión del 76.77% en promedio durante la validación cruzada. Esto representa una ligera mejora respecto a la precisión inicial de 74.67%, lo que sugiere que la búsqueda de los hiperparámetros óptimos ha ayudado a mejorar el rendimiento del modelo.

8.- Reentrenar el modelo con los mejores hiperparámetros y reevaluarlo

```
# Reentrenar el modelo con los mejores hiperparámetros
modelo_opt = DecisionTreeClassifier(**resultado.best_params_)
modelo_opt.fit(X_train, y_train)

# Hacer predicciones optimizadas en el conjunto de prueba
predicciones_opt = modelo_opt.predict(X_test)

# Calcular la precisión del modelo optimizado
precision_opt = accuracy_score(y_test, predicciones_opt)
print(f"Precision optimizada: {precision_opt}")

# Mostrar la matriz de confusión optimizada
matriz_confusion_opt = confusion_matrix(y_test, predicciones_opt)
print("Matriz de confusión optimizada:\n", matriz_confusion_opt)

# Mostrar reporte de clasificación detallado
reporte_clasificacion = classification_report(y_test, predicciones_opt)
print("Reporte de clasificación:\n", reporte_clasificacion)
```

```

Precisión optimizada: 0.7597402597402597
Matriz de confusión optimizada:
[[76 23]
 [14 41]]
Reporte de clasificación:
precision    recall    f1-score   support
          0.0      0.84      0.77      0.80       99
          1.0      0.64      0.75      0.69       55
          accuracy           0.76      154
          macro avg      0.74      0.76      0.75      154
          weighted avg     0.77      0.76      0.76      154

```

Los resultados optimizados muestran una mejora general en el modelo. Aunque la precisión total ha aumentado ligeramente, el modelo es especialmente más eficaz en identificar correctamente a los pacientes diabéticos, como lo indica el aumento en el recall y la reducción en los falsos negativos. La precisión del modelo es sólida para la clase no diabética, pero aún tiene margen de mejora para la clase diabética.

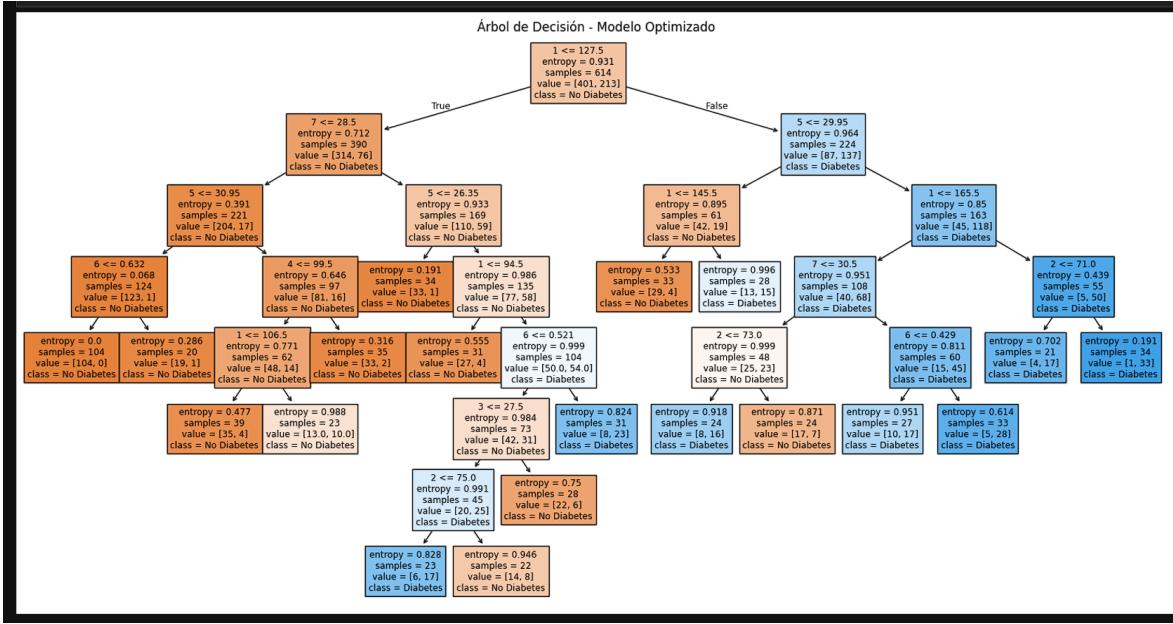
Gráfica personalizada e interpretación de resultados

9.- Visualización del árbol de decisión optimizado

```

# Visualizar el árbol de decisión optimizado
plt.figure(figsize=(20, 10))
plot_tree(modelo_opt, filled=True, feature_names=datos.columns[1:-1], class_names=['No Diabetes', 'Diabetes'])
plt.title("Árbol de Decisión - Modelo Optimizado")
plt.show()

```



Aquí se visualiza el árbol de decisión optimizado, que muestra cómo el modelo toma decisiones basadas en las características del conjunto de datos. El gráfico permite identificar las características más importantes y cómo se utilizan en las divisiones para clasificar a los pacientes en las categorías de "No Diabetes" y "Diabetes".

Enlace hacia el modelo obtenido

https://github.com/DavidGuillermoSantiago/evaluacion/blob/main/ejercicio2_corregido.ipynb

Ejercicio 3 - Corregido

Justificación del algoritmo

Para este análisis, se ha seleccionado el algoritmo de **K-Means**. Este algoritmo es adecuado para este tipo de problemas debido a su simplicidad y eficacia en la agrupación de datos numéricos en clusters. K-Means busca minimizar la variación dentro de cada cluster y es útil cuando se desea identificar patrones o grupos similares en los datos.

En este caso, al tener datos financieros de "Close" y "Volume", K-Means puede ayudar a identificar comportamientos similares en el precio de cierre y el volumen de transacciones de Samsung a lo largo del tiempo.

Descripción del diseño del modelo

1.- Importamos las librerías necesarias para el análisis

```
# Importamos las librerías necesarias para el análisis
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.cluster import KMeans
from mpl_toolkits.mplot3d import Axes3D
```

2.- Cargamos los datos desde el archivo CSV

```
# Cargamos los datos desde el archivo CSV
datos = pd.read_csv("C:/Users/david/segundaEvaluacion/Conjunto de Datos/samsung.csv")
```

3.- Exploramos los primeros datos del archivo para asegurarnos de que se han cargado correctamente

```
# Exploramos los primeros datos del archivo
datos.head()
```

	Date	Close	Volume
0	02/01/2008	10880	18047200
1	03/01/2008	10920	19346500
2	04/01/2008	10780	17997350
3	07/01/2008	10380	39787200
4	08/01/2008	10320	24783700

4.- Preprocesamos los datos seleccionando las columnas de interés: 'Close' y 'Volume'

- Seleccionamos las columnas 'Close' y 'Volume' porque representan el precio de cierre y el volumen de transacciones, que son los principales factores financieros en el análisis de acciones.

```
# Preprocesamos los datos seleccionando las columnas de interés: 'Close' y 'Volume'  
data = datos[['Close', 'Volume']]
```

- Normalizamos los datos para evitar que una variable (como el volumen) tenga más peso que la otra (precio de cierre) en el análisis de agrupación.

```
# Normalizamos los datos para que todas las características tengan la misma escala  
datos_escalada = preprocessing.Normalizer().fit_transform(data)
```

```
# Copiamos los datos preprocesados para usarlos en el modelo  
X = datos_escalada.copy()
```

5.- Curva de Codo

- Determinamos el número óptimo de clusters utilizando la técnica de "codo"

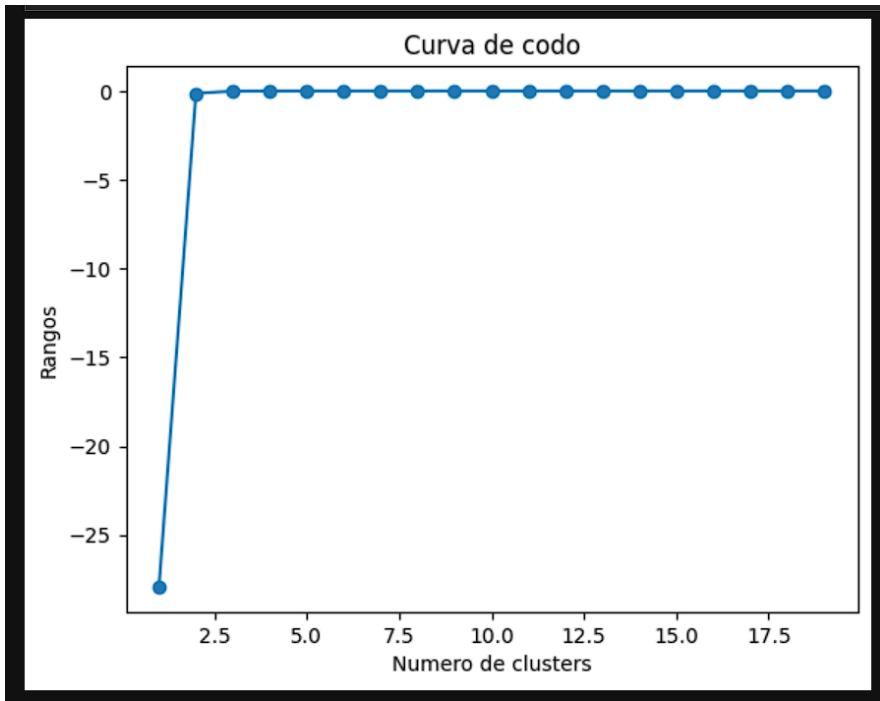
```
# Determinamos el número óptimo de clusters utilizando la técnica de "codo"  
Nc = range(1, 20)  
kmeans = [KMeans(n_clusters=i) for i in Nc]
```

- Calculamos el puntaje de cada modelo K-Means y graficamos la curva de codo

```
# Calculamos el puntaje de cada modelo K-Means y graficamos la curva de codo  
score = [kmeans[i].fit(X).score(X) for i in range(len(kmeans))]
```

- Graficamos la curva de codo para identificar el mejor número de clusters

```
# Graficamos la curva de codo para identificar el mejor número de clusters  
plt.plot(Nc, score, marker='o')  
plt.xlabel('Número de clusters')  
plt.ylabel('Rangos')  
plt.title('Curva de codo')  
plt.show()
```



Utilizamos la técnica de la curva de codo para determinar el número óptimo de clusters. La curva de codo grafica la suma de las distancias al cuadrado desde cada punto hasta el centroide de su cluster correspondiente. El número óptimo de clusters es aquel donde la disminución de la variación comienza a estabilizarse.

6.- Entrenamiento del modelo

Ejecutamos K-Means con 3 clusters (elegido basado en la curva de codo). Los centroides resultantes representan los puntos centrales de los clusters.

- **Ejecutamos K-Means con el número de clusters seleccionado basado en la curva de codo**

```
# Ejecutamos K-Means con el número de clusters seleccionado basado en la curva de codo
kmeans = KMeans(n_clusters=3).fit(X)
```

- **Obtenemos los centroides de los clusters**

```
# Obtenemos Los centroides de Los clusters
centroids = kmeans.cluster_centers_
print(centroids)

[[0.00224737 0.99999613]
 [1.          0.        ]
 [0.35026946 0.93664898]]
```

los centroides de los tres clusters indican lo siguiente:

- **Cluster 1:** Días con bajo precio de cierre y alto volumen de transacciones.
- **Cluster 2:** Días con alto precio de cierre y bajo volumen de transacciones.
- **Cluster 3:** Días con un precio de cierre moderado y un volumen alto.

7. Visualización 3D

- Realizamos la predicción de clusters

```
# Realizamos la predicción de clusters
labels = kmeans.predict(X)
```

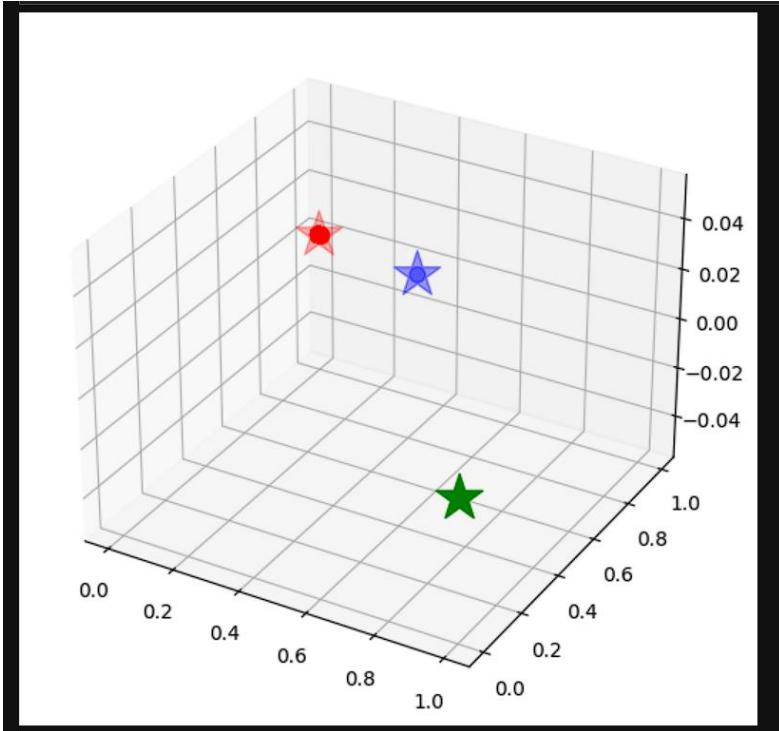
- Asignamos colores a cada cluster para visualización

```
# Asignamos colores a cada cluster para visualización
colores = ['red', 'green', 'blue']
asignar = []
for row in labels:
    asignar.append(colores[row])
```

- Visualizamos los clusters en 3D

```
# Visualizamos los clusters en 3D
fig = plt.figure()
ax = Axes3D(fig, auto_add_to_figure=False)
fig.add_axes(ax)
ax.scatter(X[:, 0], X[:, 1], c=asignar, s=60)
ax.scatter(centroids[:, 0], centroids[:, 1], marker='*', c=colores, s=600)
plt.show()
```

Visualizamos los clusters en un gráfico 3D para entender cómo se agrupan los datos en función de las dos variables seleccionadas ('Close' y 'Volume'). Los centroides están marcados con un asterisco (*) para diferenciarlos de los puntos de datos.



8.- Gráfica 2D

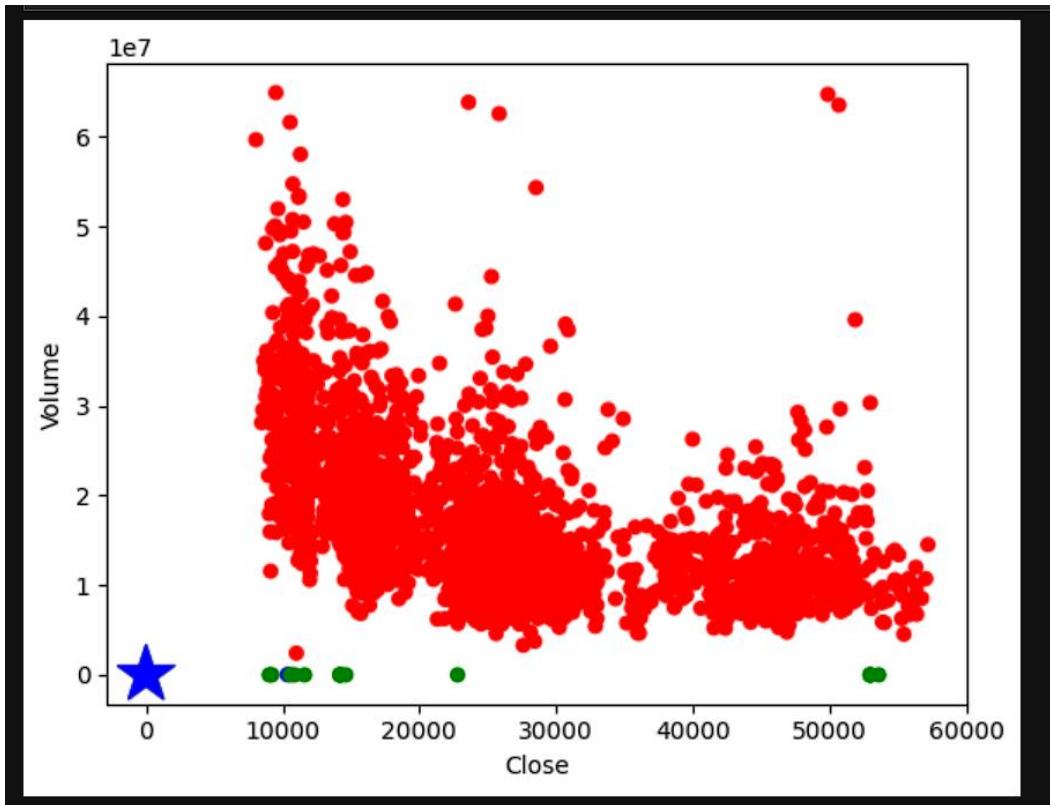
- Comparación de las variables "Close" y "Volume" en 2D

```
# Comparación de las variables "Close" y "Volume" en 2D
f1 = data['Close'].values
f2 = data['Volume'].values
```

- Graficamos los datos con los colores asignados según los clusters

Comparamos las variables 'Close' y 'Volume' en un gráfico 2D para visualizar cómo se distribuyen los clusters en el espacio de las dos características.

```
# Graficamos los datos con los colores asignados según los clusters
plt.scatter(f1, f2, c=asignar, s=30)
plt.scatter(centroids[:, 0], centroids[:, 1], marker='*', c=colores, s=500)
plt.xlabel('Close')
plt.ylabel('Volume')
plt.show()
```



Enlace hacia el modelo obtenido

https://github.com/DavidGuillermoSantiago/evaluacion/blob/main/ejericicio3_corregido.ipynb

Ejercicio 4 - Corregido

Justificación del algoritmo

Para este ejercicio, se ha seleccionado el algoritmo de **Análisis de Componentes Principales (PCA)** para la reducción de dimensionalidad. PCA es adecuado para este conjunto de datos ya que busca reducir el número de variables (dimensiones) mientras mantiene la mayor cantidad de varianza (información) posible. Dado que el conjunto de datos tiene múltiples variables que pueden estar correlacionadas, PCA permite simplificar el modelo sin perder información crucial, mejorando así la eficiencia y la interpretabilidad del análisis.

Descripción del diseño del modelo

1.- Importar las librerías necesarias

```
# Importar Las librerías necesarias
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = (16,6)
plt.style.use('ggplot')
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

2.- Cargar el conjunto de datos

```
datos = pd.read_csv("C:/Users/david/segundaEvaluacion/Conjunto de Datos/comprar_alquilar.csv")
datos.head()
```

	ingresos	gastos_comunes	pago_coche	gastos_otros	ahorros	vivienda	estado_civil	hijos	trabajo	comprar
0	6000	1000	0	600	50000	400000	0	2	2	1
1	6745	944	123	429	43240	636897	1	3	6	0
2	6455	1033	98	795	57463	321779	2	1	8	1
3	7098	1278	15	254	54506	660933	0	0	3	0
4	6167	863	223	520	41512	348932	0	0	3	1

3.- Escalar los datos

Dado que PCA es sensible a las escalas de las variables, es necesario estandarizar los datos antes de aplicar el algoritmo. Aquí, se eliminan las columnas no numéricas y la columna objetivo comprar, y se aplica la estandarización utilizando StandardScaler de scikit-learn.

```
# Escalamos los datos
# El escalado es necesario para PCA ya que se basa en la varianza y magnitud de los datos
scaler = StandardScaler()
df = datos.drop(['comprar'], axis=1) # Eliminamos la columna 'comprar' que es el objetivo y no debe ser escalada
scaler.fit(df) # Ajustamos el escalador a los datos
X_Scaled = scaler.transform(df) # Aplicamos la transformación a los datos
```

4.- Aplicar PCA

Se aplica PCA especificando el número de componentes principales. En este caso, hemos elegido 9 componentes principales para capturar la mayor cantidad de varianza en los datos.

```
# Implementamos PCA con 9 componentes
pca = PCA(n_components=9) # Especificamos que queremos 9 componentes principales
pca.fit(X_Scaled) # Ajustamos el PCA a los datos escalados
X_pca = pca.transform(X_Scaled) # Transformamos los datos originales a los componentes principales
```

5.- Mostrar los resultados de la varianza explicada

Se muestra el porcentaje de varianza explicada por cada componente principal, lo que ayuda a entender cuánta información original se conserva en el modelo reducido. Aquí también se suma la varianza explicada por los primeros cinco componentes principales.

```
# Mostrar los resultados de la varianza explicada
expl = pca.explained_variance_ratio_
print("Varianza explicada por cada componente principal:")
for i, var in enumerate(expl):
    print(f"PC{i+1}: {var:.2%}")
print("Suma de varianza explicada en las primeras 5 componentes:", sum(expl[0:5]))

Varianza explicada por cada componente principal:
PC1: 29.91%
PC2: 23.29%
PC3: 11.71%
PC4: 10.69%
PC5: 9.64%
PC6: 5.63%
PC7: 4.15%
PC8: 3.02%
PC9: 1.97%
Suma de varianza explicada en las primeras 5 componentes: 0.8524062117714313
```

6.- Crear un DataFrame con los componentes principales

Se crea un DataFrame con los componentes principales, y se agrega la columna objetivo comprar al final. Este DataFrame será útil para análisis adicionales.

```
# Crear un DataFrame con Los componentes principales
principal_columns = [f'Principal Component {i+1}' for i in range(X_pca.shape[1])]
principal_df = pd.DataFrame(data=X_pca, columns=principal_columns)

# Agregar La columna objetivo al DataFrame
principal_df['comprar'] = datos['comprar']
```

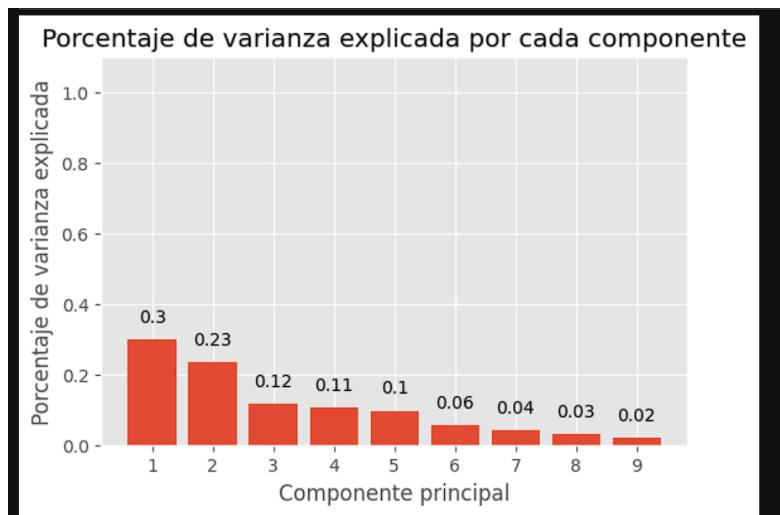
7.- Gráfica de la varianza explicada por cada componente

```
# Gráfica de La varianza explicada por cada componente
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 4))
ax.bar(x=np.arange(pca.n_components_) + 1, height=pca.explained_variance_ratio_)

for x, y in zip(np.arange(len(df.columns)) + 1, pca.explained_variance_ratio_):
    label = round(y, 2)
    ax.annotate(label, (x,y), textcoords="offset points", xytext=(0,10), ha='center')

ax.set_xticks(np.arange(pca.n_components_) + 1)
ax.set_yticks(np.arange(0, 1.1))
ax.set_title('Porcentaje de varianza explicada por cada componente')
ax.set_xlabel('Componente principal')
ax.set_ylabel('Porcentaje de varianza explicada')
plt.show()
```

Se genera una gráfica de barras que muestra el porcentaje de varianza explicada por cada componente principal. Esto ayuda a visualizar cuántos componentes son necesarios para capturar la mayor parte de la varianza en los datos.



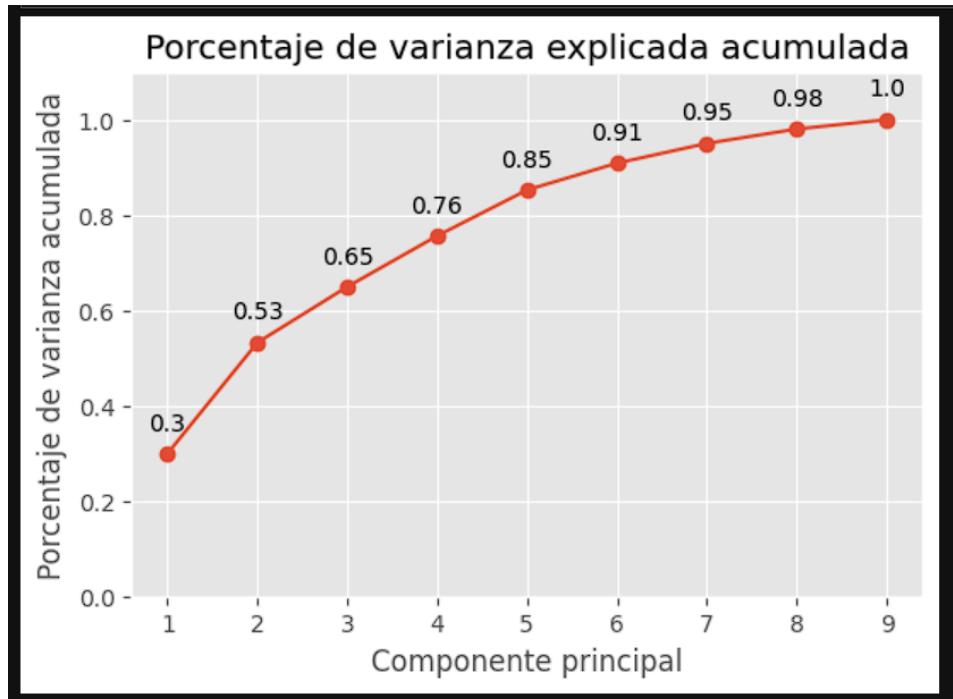
8.- Gráfica de la varianza acumulada

```
# Gráfica de la varianza acumulada
prop_varianza_acum = pca.explained_variance_ratio_.cumsum()
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 4))
ax.plot(np.arange(len(df.columns)) + 1, prop_varianza_acum, marker='o')

for x, y in zip(np.arange(len(df.columns)) + 1, prop_varianza_acum):
    label = round(y, 2)
    ax.annotate(label, (x,y), textcoords="offset points", xytext=(0,10), ha='center')

ax.set_xlim(0, 1.1)
ax.set_xticks(np.arange(pca.n_components_) + 1)
ax.set_title('Porcentaje de varianza explicada acumulada')
ax.set_xlabel('Componente principal')
ax.set_ylabel('Porcentaje de varianza acumulada')
plt.show()
```

Aquí se presenta una gráfica que muestra la varianza acumulada explicada por las componentes principales. Esta gráfica es útil para determinar cuántas componentes se necesitan para capturar una cantidad significativa de la varianza total.



Las gráficas generadas permiten identificar cuántos componentes principales capturan la mayor parte de la varianza en los datos. Si observamos que los primeros componentes capturan una cantidad significativa de la varianza, podríamos reducir las dimensiones del conjunto de datos utilizando solo esos componentes.

9.- Reducción a 6 componentes principales

La elección de reducir a 6 componentes principales se justifica en función del balance entre retener la mayor cantidad de información posible (medida a través de la varianza explicada) y la necesidad de simplificar el modelo para hacerlo más manejable y eficiente.

- Escalado de los datos eliminando columnas categóricas

```
# Escalado de los datos eliminando columnas categóricas
scaler = StandardScaler()
df2 = datos.drop(['comprar', 'estado_civil', 'hijos', 'trabajo'], axis=1)
scaler.fit(df2)
X_Scaled = scaler.transform(df2)
```

- Aplicación de PCA con 6 componentes principales

```
# Aplicación de PCA con 6 componentes principales
pca2 = PCA(n_components=6)
pca2.fit(X_Scaled)
X_pca2 = pca2.transform(X_Scaled)
```

- Resultados de la reducción a 6 componentes

```
# Resultados de La reducción a 6 componentes
expl = pca2.explained_variance_ratio_
print("Varianza explicada por cada componente principal:")
for i, var in enumerate(expl):
    print(f"PC{i+1}: {var:.2%}")
print("Suma de la varianza explicada por los primeros 5 componentes:", sum(expl[0:5]))

Varianza explicada por cada componente principal:
PC1: 42.70%
PC2: 17.21%
PC3: 16.03%
PC4: 14.39%
PC5: 6.69%
PC6: 2.98%
Suma de la varianza explicada por los primeros 5 componentes: 0.9701626349890449
```

- Creación de DataFrame con los componentes principales

```
# Creación de DataFrame con los componentes principales
principal_columns2 = [f'Principal Component {i+1}' for i in range(X_pca2.shape[1])]
principal_df2 = pd.DataFrame(data=X_pca2, columns=principal_columns2)
```

- Agregar la columna objetivo al DataFrame

```
# Agregar La columna objetivo al DataFrame
principal_df2['comprar'] = datos['comprar']
```

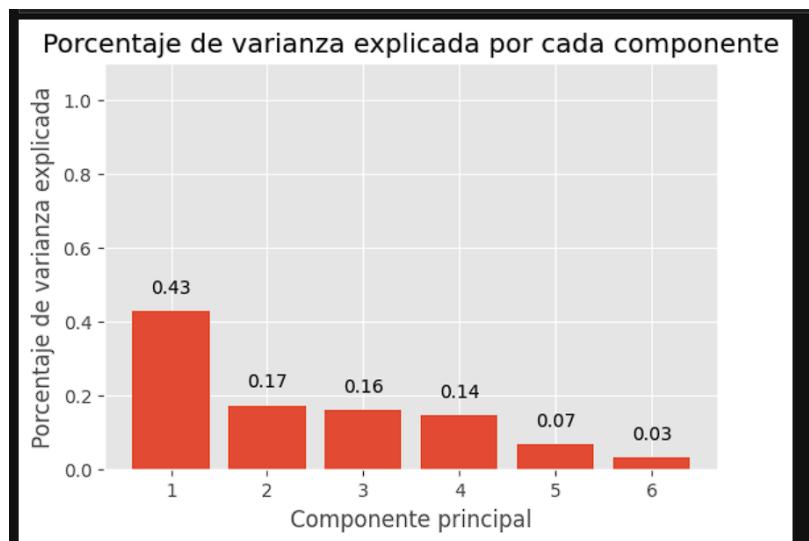
10.- Crear una gráfica de barras para visualizar el porcentaje de varianza explicada por cada componente

```
: # Crear una gráfica de barras
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 4))
ax.bar(
    x=np.arange(pca2.n_components_) + 1, # Eje X: Número de componente
    height=pca2.explained_variance_ratio_ # Eje Y: Porcentaje de varianza explicada por cada componente
)

# Añadir anotaciones en La gráfica para mostrar el valor exacto de cada barra
for x, y in zip(np.arange(len(df2.columns)) + 1, pca2.explained_variance_ratio_):
    label = round(y, 2)
    ax.annotate(
        label, # El valor a mostrar
        (x, y), # Posición en La gráfica
        textcoords="offset points",
        xytext=(0, 10), # Desplazamiento en Y de La anotación
        ha='center' # Alinear el texto en el centro
    )

# Configuración de Los ejes y el título de La gráfica
ax.set_xticks(np.arange(pca2.n_components_) + 1) # Etiquetas en el eje X
ax.set_ylim(0, 1.1) # Limitar el eje Y para que no sobrepase el 100%
ax.set_title('Porcentaje de varianza explicada por cada componente')
ax.set_xlabel('Componente principal')
ax.set_ylabel('Porcentaje de varianza explicada')

plt.show() # Mostrar La gráfica
```



11.- Calcular el porcentaje de varianza explicada acumulada

```
prop_varianza_acum = pca2.explained_variance_ratio_.cumsum()

# Mostrar los valores calculados
print('-----')
print('Porcentaje de varianza explicada acumulada')
print('-----')
print(prop_varianza_acum)

-----
Porcentaje de varianza explicada acumulada
-----
[0.42698481 0.59911895 0.75939032 0.90331034 0.97016263 1. ]
```

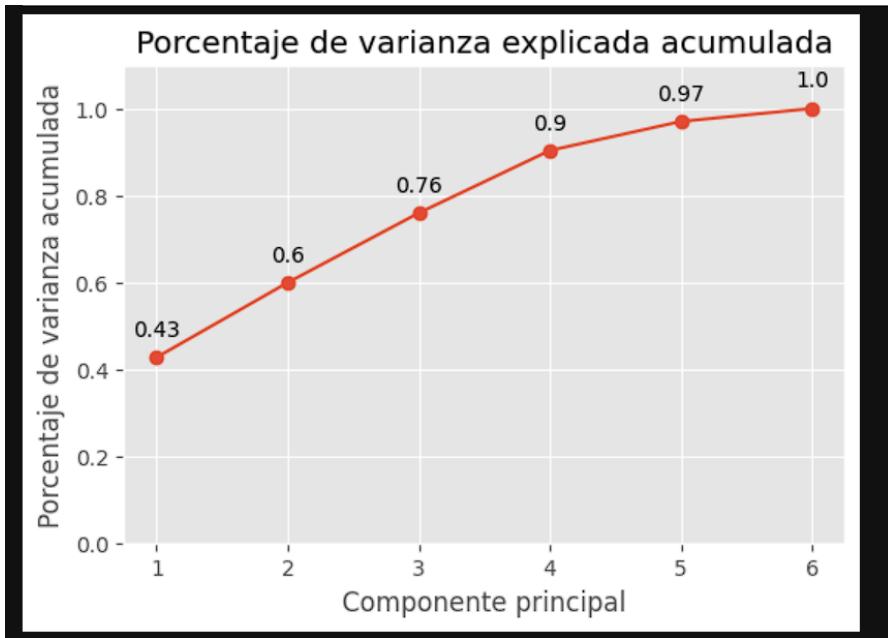
12.- Crear una gráfica de línea para visualizar el porcentaje de varianza explicada acumulada

```
# Crear una gráfica de línea
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 4))
ax.plot(
    np.arange(len(df2.columns)) + 1, # Eje X: Número de componente
    prop_varianza_acum, # Eje Y: Porcentaje de varianza acumulada
    marker='o' # Marcador de puntos
)

# Añadir anotaciones en la gráfica para mostrar el valor exacto en cada punto
for x, y in zip(np.arange(len(df2.columns)) + 1, prop_varianza_acum):
    label = round(y, 2)
    ax.annotate(
        label, # El valor a mostrar
        (x, y), # Posición en la gráfica
        textcoords="offset points",
        xytext=(0, 10), # Desplazamiento en Y de la anotación
        ha='center' # Alinear el texto en el centro
    )

# Configuración de los ejes y el título de la gráfica
ax.set_ylim(0, 1.1) # Limitar el eje Y para que no sobrepase el 100%
ax.set_xticks(np.arange(pca2.n_components_) + 1) # Etiquetas en el eje X
ax.set_title('Porcentaje de varianza explicada acumulada')
ax.set_xlabel('Componente principal')
ax.set_ylabel('Porcentaje de varianza acumulada')

plt.show() # Mostrar la gráfica
```



Enlace hacia el modelo obtenido

https://github.com/DavidGuillermoSantiago/evaluacion/blob/main/ejercicio4_Corregido.ipynb

Ejercicio 5 – Remedial

Justificación del algoritmo.

Selección del Algoritmo

El algoritmo seleccionado para este ejercicio es la **Regresión Lineal**. Este modelo es adecuado cuando creemos que existe una relación lineal entre las variables independientes (por ejemplo, Word count, # of Links, etc.) y la variable dependiente que queremos predecir.

Justificación

La Regresión Lineal es uno de los algoritmos más básicos en Machine Learning y es muy útil para establecer relaciones entre variables numéricas. En este caso, intentaremos predecir el número de compartidos (# Shares) basándonos en Word count y # of Links.

Descripción del diseño del modelo.

1.- Importar Librerías Necesarias

```
# Importar las bibliotecas necesarias
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, RepeatedKFold, GridSearchCV
from sklearn.linear_model import LinearRegression
from sklearn import metrics
from scipy.stats import loguniform
```

2.- Cargar y Explorar el Conjunto de Datos

```
# Cargar datos
datos = pd.read_csv("C:/Users/david/segundaEvaluacion/Conjunto de Datos/articulos_ml.csv")

print(datos.head())
          Title \
0 What is Machine Learning and how do we use it ...
1 10 Companies Using Machine Learning in Cool Ways
2 How Artificial Intelligence Is Revolutionizing...
3 Dbrain and the Blockchain of Artificial Intell...
4 Nasa finds entire solar system filled with eig...

          url  Word count  # of Links \
0 https://blog.signals.network/what-is-machine-1...      1888         1
1                                         NaN      1742         9
2                                         NaN      962         6
3                                         NaN     1221         3
4                                         NaN     2039         1

   # of comments  # Images video  Elapsed days  # Shares
0            2.0        2       34    200000
1           NaN        9       5     25000
2            0.0        1      10     42000
3           NaN        2       68    200000
4          104.0        4      131    200000
```

3.- Preprocesamiento de Datos

```
# Eliminar columnas con datos faltantes
datos = datos.dropna()

# Separar las variables independientes (X) y la variable dependiente (y)
X = datos[['Word count', '# of Links']] # Variables independientes
y = datos['# Shares'] # Variable dependiente
```

4.- División de los Datos en Conjuntos de Entrenamiento y Prueba

Dividimos los datos en conjuntos de entrenamiento y prueba, usando 80% de los datos para entrenar y 20% para probar el modelo.

```
# 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)
```

5.- Entrenamiento del Modelo de Regresión Lineal

```
# Crear el modelo de regresión lineal
model = LinearRegression()

# Entrenar el modelo con los datos de entrenamiento
model.fit(X_train, y_train)
```

▼ LinearRegression ⓘ ?
LinearRegression()

6.- Obtener el intercepto (β_0) del modelo

Es el valor predicho de la variable dependiente (# Shares) cuando todas las variables independientes (Word count, # of Links, etc.) son cero.

```
# Obtener el intercepto ( $\beta_0$ ) del modelo
print(f'Intercepto: {model.intercept_}')
```

Intercepto: 6017.623540028642

7.- Obtener los coeficientes (β_1, β_2, \dots) del modelo para cada variable independiente

Indican cómo cambia la variable dependiente (# Shares) en respuesta a un cambio unitario en cada una de las variables independientes mientras se mantienen las demás constantes.

```
# Obtener Los coeficientes ( $\beta_1, \beta_2, \dots$ ) del modelo para cada variable independiente
print(f'Coeficientes: {model.coef_}')

Coeficientes: [  8.36759872 231.03342141]
```

8.- Predecir los valores de la variable dependiente usando los datos de prueba

```
# Predecir Los valores de la variable dependiente usando los datos de prueba
y_pred = model.predict(X_test)

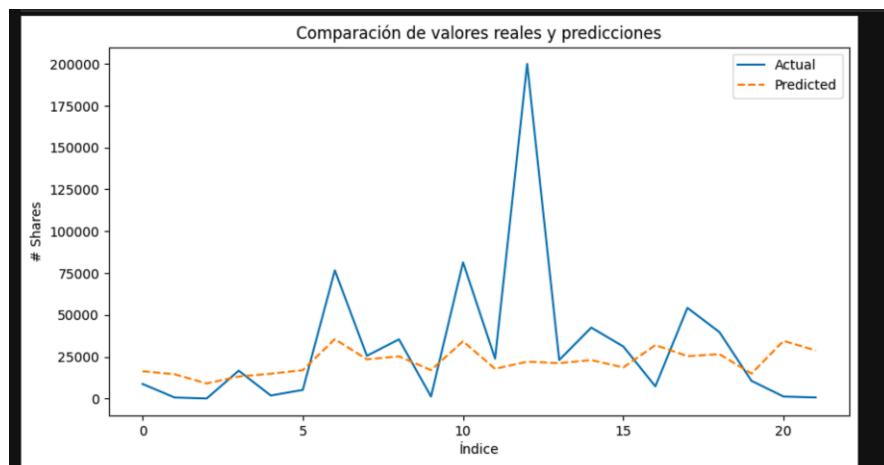
# Crear un DataFrame para comparar Los valores reales y las predicciones
datos_pred = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})

# Mostrar Los primeros registros del DataFrame de comparación
print(datos_pred.head())

   Actual      Predicted
155     8707  16298.143029
63      631   14515.844501
41      22    9033.656907
136    16641  13187.245216
152    1824   14804.040683
```

11.- Visualización de los resultados: Valores reales vs Predicciones

```
# Visualización de los resultados
plt.figure(figsize=(10,5))
plt.plot(datos_pred['Actual'].values, label='Actual')
plt.plot(datos_pred['Predicted'].values, label='Predicted', linestyle='--')
plt.xlabel('Índice')
plt.ylabel('# Shares')
plt.legend()
plt.title('Comparación de valores reales y predicciones')
plt.show()
```



12.- Optimización del modelo

Se utiliza GridSearchCV para encontrar los mejores hiperparámetros para el modelo y se reentrena con ellos.

- Definir la evaluación cruzada usando K-Fold repetido

```
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
```

- Definir el espacio de búsqueda para la optimización de hiperparámetros

```
space = {
    'copy_X': [True, False],
    'fit_intercept': [True, False],
    'positive': [True, False]
}
```

- Definir la búsqueda de los mejores hiperparámetros usando GridSearchCV

```
search = GridSearchCV(model, space, scoring='neg_mean_absolute_error', n_jobs=-1, cv=cv)
```

- Ejecutar la búsqueda

```
result = search.fit(X_train, y_train)
```

- Mostrar los mejores resultados obtenidos

```
print(f'Mejor puntuación: {result.best_score_}')
print(f'Mejores Hiperparámetros: {result.best_params_}')

Mejor puntuación: -17587.45882757828
Mejores Hiperparámetros: {'copy_X': True, 'fit_intercept': True, 'positive': True}
```

13.- Crear un nuevo modelo con los mejores hiperparámetros obtenidos

```
model_optimized = LinearRegression(
    fit_intercept=result.best_params_['fit_intercept'],
    positive=result.best_params_['positive'],
    copy_X=result.best_params_['copy_X']
)
```

```
# Reentrenar el modelo optimizado con los datos de entrenamiento  
model_optimized.fit(X_train, y_train)
```

```
▼ LinearRegression ⓘ ⓘ  
LinearRegression(positive=True)
```

```
# Predecir con el modelo optimizado  
y_pred_optimized = model_optimized.predict(X_test)
```

14.- Crear un DataFrame para comparar valores reales y predicciones optimizadas

```
datos_pred_optimized = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred_optimized})
```

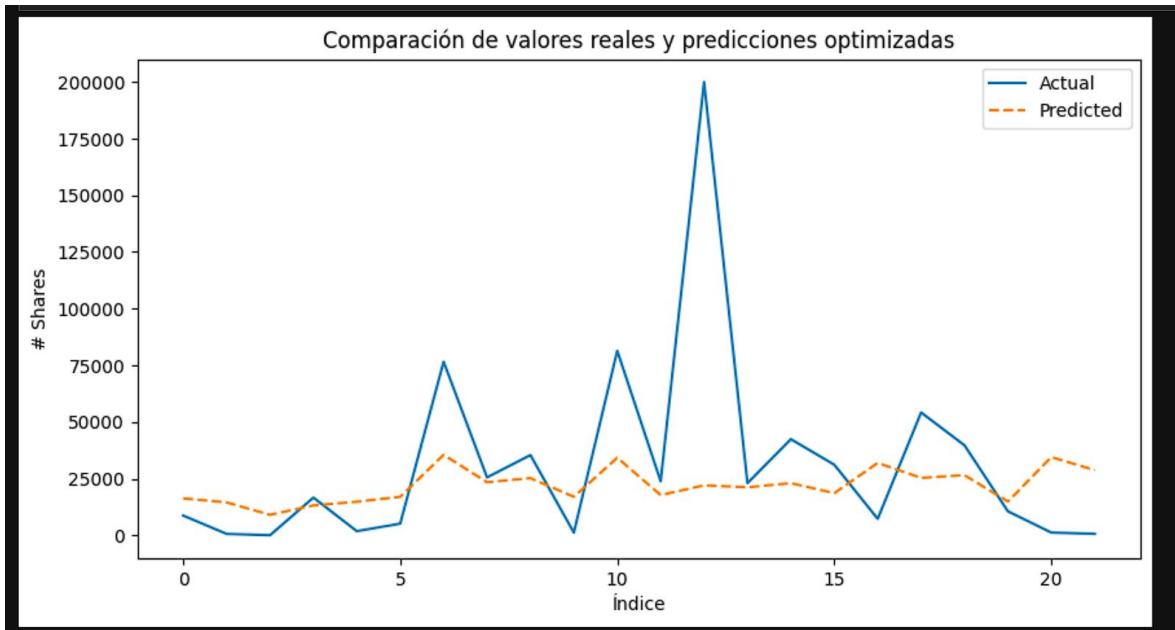
15.- Mostrar los primeros registros del DataFrame de comparación optimizada

```
print(datos_pred_optimized.head())
```

	Actual	Predicted
155	8707	16298.143029
63	631	14515.844501
41	22	9033.656907
136	16641	13187.245216
152	1824	14804.040683

16.- Visualización

```
# Visualización de los resultados optimizados  
plt.figure(figsize=(10,5))  
plt.plot(datos_pred_optimized['Actual'].values, label='Actual')  
plt.plot(datos_pred_optimized['Predicted'].values, label='Predicted', linestyle='--')  
plt.legend()  
plt.xlabel('Índice')  
plt.ylabel('# Shares')  
plt.title('Comparación de valores reales y predicciones optimizadas')  
plt.show()
```



Las líneas de valores reales y predichos están bastante alineadas, lo que indica que el modelo captura bien las tendencias generales de los datos.

Aunque hay algunas diferencias entre los valores reales y predichos, en general estas no son muy grandes. Esto sugiere que el modelo es bastante preciso, pero aún podría haber margen para mejoras.

Enlace al modelo obtenido

<https://github.com/DavidGuillermoSantiago/evaluacion/blob/main/ejercicio5.ipynb>

Ejercicio 6 - Remedial

Justificación del algoritmo

Algoritmo Elegido: K-Means

K-Means es un algoritmo de agrupación no supervisado que divide un conjunto de datos en k clústeres. La elección de **K-Means** se basa en su simplicidad y eficacia para manejar grandes conjuntos de datos. Utilizamos la técnica de la Curva del Codo para encontrar el número óptimo de clústeres, lo cual es una práctica común para evaluar cómo el número de clústeres afecta la variabilidad dentro de cada clúster.

Descripción del Diseño del Modelo

1.- Importación de Librerías

- **pandas**: Para cargar y manipular datos en formato tabular.
- **matplotlib.pyplot**: Para crear gráficos y visualizaciones.
- **sklearn.preprocessing**: Para preprocesar datos, especialmente la normalización.
- **sklearn.cluster.KMeans**: Para aplicar el algoritmo de agrupación K-Means.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.cluster import KMeans
```

2.- Cargar el Conjunto de Datos

```
datos = pd.read_csv("C:/Users/david/segundaEvaluacion/Conjunto de Datos/cliente_tienda.csv")
```

3.- Seleccionar las Columnas de Interés

Extraemos solo las columnas relevantes para el análisis de agrupación, es decir, "Ingresos Anuales (\$)" y "Porcentaje de gastos (1-100)". Estas columnas serán usadas para el agrupamiento.

```
data = datos[['Ingresos Anuales ($)', 'Porcentaje de gastos (1-100)']]
```

4.- Preprocesar los Datos

La normalización es importante para asegurarse de que todas las características contribuyan igualmente al análisis.

```
data_escalada = preprocessing.StandardScaler().fit_transform(data)
```

5.- Determinar el Número Óptimo de Clústeres

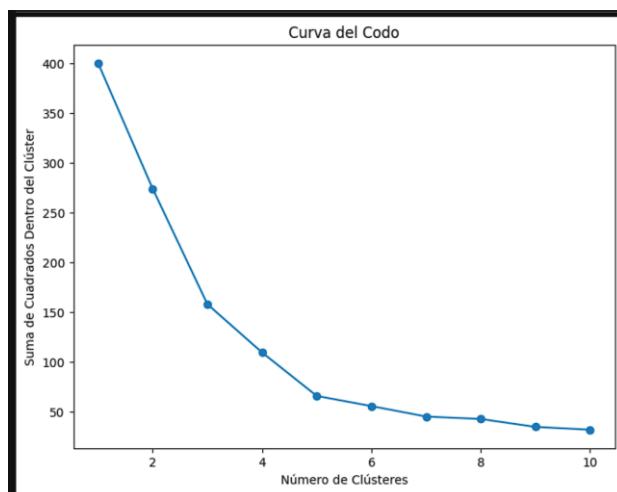
```
Nc = range(1, 11)
kmeans = [KMeans(n_clusters=i, random_state=1) for i in Nc]
scores = [kmeans[i].fit(data_escalada).inertia_ for i in range(len(kmeans))]
```

6.- Graficar la Curva del Codo

```
: plt.figure(figsize=(8, 6))
plt.plot(Nc, scores, marker='o')
plt.xlabel('Número de Clústeres')
plt.ylabel('Suma de Cuadrados Dentro del Clúster')
plt.title('Curva del Codo')
plt.show()
```

El eje X muestra el número de clústeres y el eje Y muestra la inercia correspondiente.

La inercia generalmente disminuye a medida que aumentas el número de clústeres. Esto ocurre porque al agregar más clústeres, los datos se agrupan más estrechamente, reduciendo la distancia promedio a los centroides.



7.- Ejecutar K-Means con el Número Óptimo de Clústeres

El punto de codo parece estar alrededor de 4 o 5 clústeres. En este punto, la inercia muestra una disminución significativa

- Seleccionamos el número óptimo de clústeres basado en la curva del codo, en este caso **5**

```
n_clusters = 5
kmeans = KMeans(n_clusters=n_clusters, random_state=1)
kmeans.fit(data_escalada)
```

```
▼ KMeans ⓘ ⓘ
KMeans(n_clusters=5, random_state=1)
```

8.- Obtener los Centroides y Etiquetas

Las coordenadas de los centroides de cada clúster. Los centroides son los puntos medios de cada clúster.

Las etiquetas de los clústeres asignadas a cada punto de datos. Cada etiqueta indica a qué clúster pertenece el punto.

```
centroids = kmeans.cluster_centers_
labels = kmeans.labels_
```

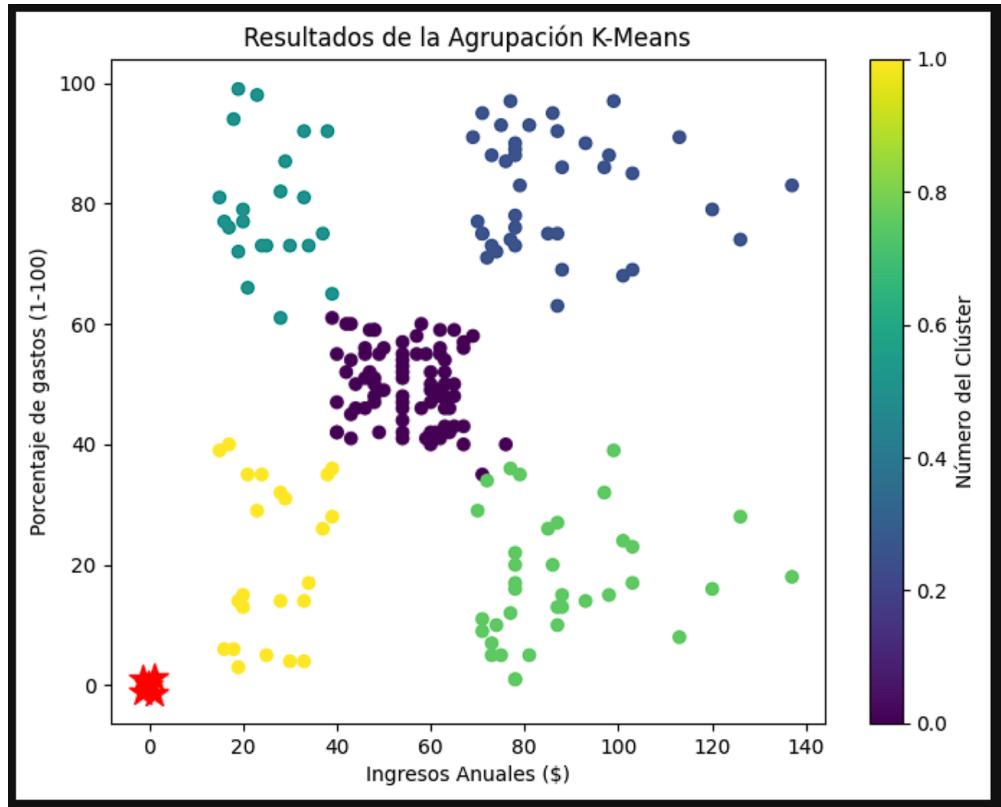
9.- Crear un DataFrame para Comparar Valores Reales y Predicciones

Añadimos una nueva columna al DataFrame original datos con las etiquetas de los clústeres. Esto permite ver a qué clúster pertenece cada cliente.

```
datos['Cluster'] = labels
```

10.- Graficar los Datos con los Clústeres

```
plt.figure(figsize=(8, 6))
plt.scatter(data['Ingresos Anuales ($)'], data['Porcentaje de gastos (1-100)'], c=labels, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], marker='*', s=200, c='red')
plt.xlabel('Ingresos Anuales ($)')
plt.ylabel('Porcentaje de gastos (1-100)')
plt.title('Resultados de la Agrupación K-Means')
plt.colorbar(label='Número del Clúster')
plt.show()
```



Este gráfico muestra los resultados de la agrupación obtenidos mediante el algoritmo de K-Means, aplicado a un conjunto de datos sobre ingresos anuales y porcentaje de gastos. La visualización ayuda a entender cómo los datos se dividen en diferentes clústeres y cómo se distribuyen en el espacio definido por las dos características principales: Ingresos Anuales y Porcentaje de Gastos.

Enlace hacia el modelo obtenido

<https://github.com/DavidGuillermoSantiago/evaluacion/blob/main/ejercicio6.ipynb>