Trabajo de Clasificación Binaria

Módulo Machine Learning

David Noya Couselo

15 de septiembre de 2024

Índice

Introducción

En la sociedad moderna, en la que el volumen de información inicial crece exponencialmente con el tiempo, el aprendizaje automático se ha convertido en una necesidad para descifrar, entender y predecir patrones complejos en grandes conjuntos de datos. Capaz de transformar información en conclusiones útiles y pronósticos precisos, la disciplina es inestimable para múltiples industrias, entre ellas, pero no limitadas a la medicina, el control de recursos, la seguridad, etc. El presente trabajo aborda el tema del aprendizaje automático con énfasis en la clasificación, una de las varias tareas fundamentales del campo.

El objetivo de este trabajo es examinar y mejorar diversos algoritmos de clasificación y técnicas de ensamble para determinar el mejor modelo de predicción de la obesidad basándose en datos biométricos y de estilo de vida. La obesidad es un problema de salud mundial con efectos devastadores en la vida y la atención médica de las personas. Por lo tanto, tener un modelo fiable para prever a aquellos en mayor riesgo es crucial. El trabajo comienza con la limpieza y el análisis exploratorio de datos para asegurarse de que los datos estén listos para el modelado. Los algoritmos de clasificación, tanto lineal como no lineal, se prueban primero. Los modelos lineales, como la regresión logística o las máquinas de soporte de vectores con kernel lineal, son simples y fáciles de entender. A su vez, los modelos no lineales como árboles de decisión, Random Forest o Gradient Boosting pueden captar relaciones complejas.

Además de abordar las limitaciones de los modelos simples, este documento también investiga los métodos de ensamblaje más avanzados de Bagging, Boosting y Stacking, ya que combinan múltiples modelos para aumentar la precisión y la confiabilidad de las predicciones. Al integrar las fortalezas de varios clasificadores base, los métodos de ensamblaje brindan mayor flexibilidad y potencial predictivo.

Durante el estudio, se realiza una búsqueda paramétrica exhaustiva para todos los modelos y métodos de ensamblaje, con el objetivo de encontrar la mejor configuración de hiperparámetros para maximizar la precisión en nuestra tarea de clasificación. Al seguir este enfoque riguroso, permitimos que los modelos se ajusten cuidadosamente a las particularidades de nuestros datos. Finalmente, nuestra evaluación concluye con una comparación de los rendimientos de todos los modelos, resaltando sus especializaciones clave y los desafíos que enfrentan. La elección del mejor modelo se guía tanto por la precisión como por la complejidad, la capacidad de interpretación y la aplicabilidad práctica. Mediante esta metodología completa, esperamos contribuir a la literatura de aprendizaje automático y a la lucha contra la obesidad proporcionando información sobre la efectividad de los métodos presentados en la predicción de la enfermedad. Aquí yace el potencial innovador de estas valiosas herramientas de análisis de datos para informar y mejorar las decisiones en el ámbito de la salud.

Preprocesamiento de datos

El preprocesamiento de datos es una etapa crítica en cualquier proyecto de Machine Learning, ya que asegura la calidad y consistencia de los datos antes de su uso en los modelos. En este apartado, se explica el proceso realizado para la limpieza, imputación

de valores faltantes y la división del conjunto de datos en muestras de entrenamiento y prueba.

Selección de la Muestra Aleatoria

Dado que el conjunto de datos original contenía más de 1000 instancias, fue necesario realizar una selección aleatoria para trabajar con una muestra de tamaño reducido. Para asegurar la reproducibilidad, esta selección fue realizada utilizando una semilla basada en los últimos cuatro dígitos del DNI. Esto garantiza que cualquier persona que ejecute el código con la misma semilla obtendrá la misma muestra.

Identificación de Valores Perdidos

El primer paso en nuestro proceso de limpieza fue identificar los valores perdidos en nuestro conjunto de datos. Este paso es crucial, ya que valores que faltan pueden indicar problemas en la recopilación o ingreso de datos y manejarlos incorrectamente puede conducir a conclusiones equivocadas. A través de la exploración inicial, identificamos las siguientes características que presentan valores perdidos:

Característica	Valores Faltantes	Porcentaje (%)
Consumo de alimentos altos en calorías (FAVC)	24	2.4
Número de comidas principales (NCP)	23	2.3
Consumo de alimentos entre comidas (CAEC)	22	2.2
Frecuencia de actividad física (FAF)	22	2.2
Consumo de verduras (FCVC)	21	2.1
Medio de transporte (MTRANS)	20	2.0
Consumo de agua diario (CH2O)	18	1.8
Consumo de alcohol (CALC)	19	1.9
Talla (Height)	17	1.7
Peso (Weight)	18	1.8
Edad (Age)	19	1.9
Historia familiar de sobrepeso (SCC)	20	2.0
Obesidad (NObeyesdad)	18	1.8
Historia familiar de obesidad (family_history_with_overweight)	17	1.7
Tiempo en dispositivos electrónicos (TUE)	15	1.5

Tabla 1: valores faltantes

Esta exploración detallada nos permitió visualizar la distribución de valores perdidos y su posible impacto en nuestro estudio.

Imputación de Valores Faltantes

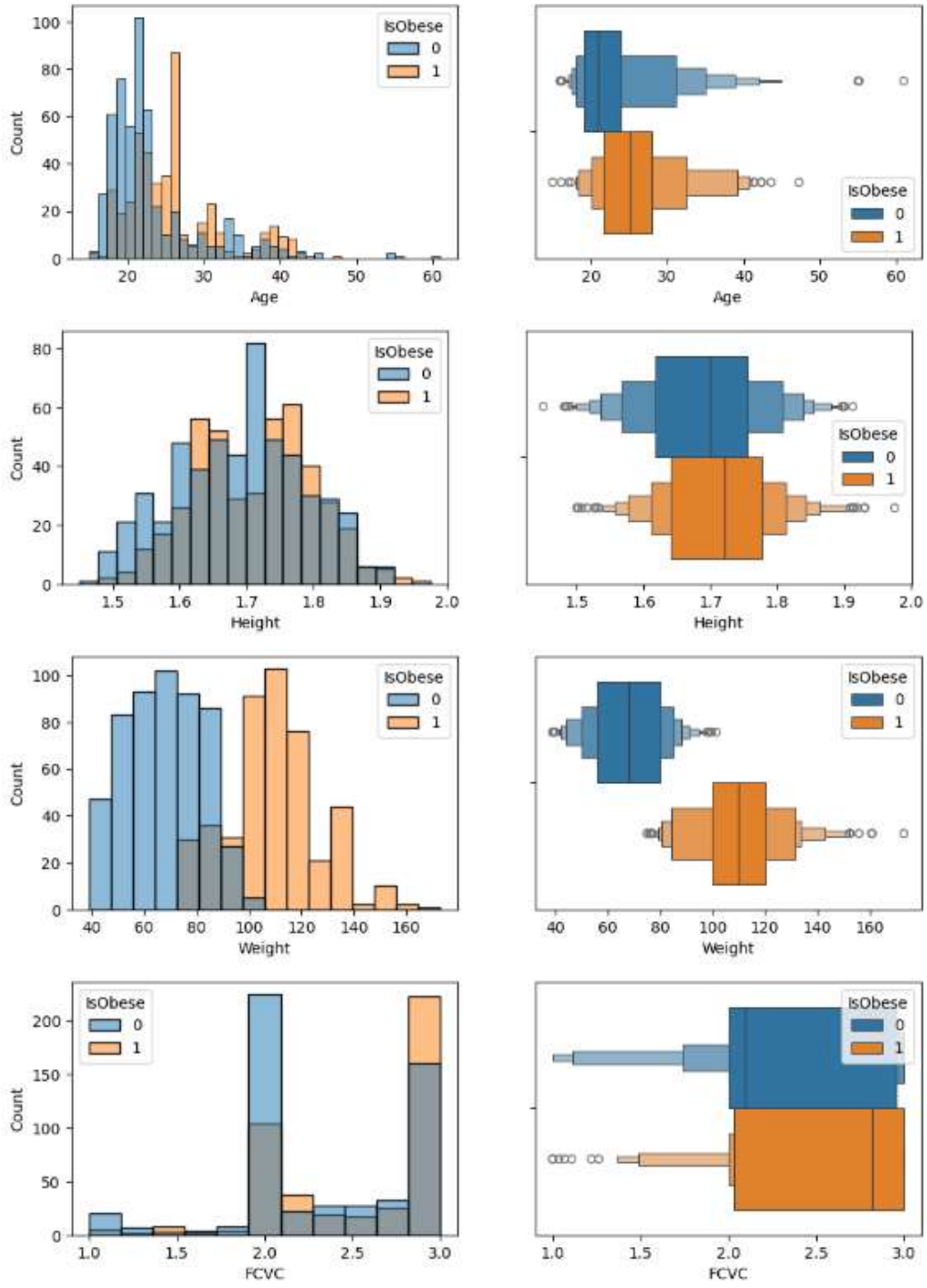
Una vez identificados los valores faltantes, se aplicaron técnicas de imputación específicas para variables numéricas y categóricas:

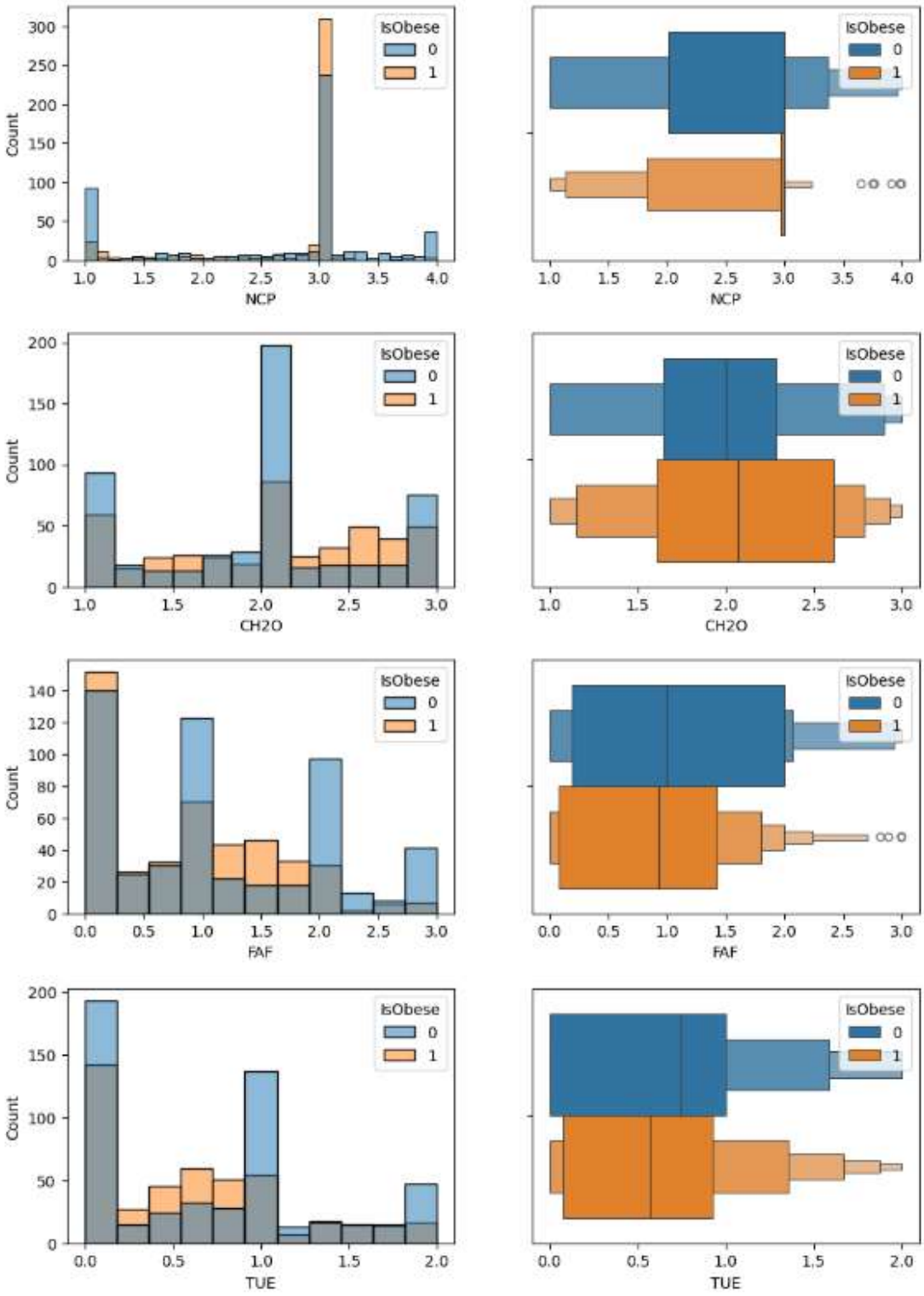
- **Variables numéricas**: Se utilizó el método de imputación mediante el algoritmo de KNNImputer. Este método imputa los valores faltantes utilizando los datos más cercanos a través de las distancias en el espacio de características. A diferencia de la simple imputación con la media o la mediana, el KNNImputer permite que las imputaciones respeten las relaciones complejas entre las variables, proporcionando así imputaciones más precisas.

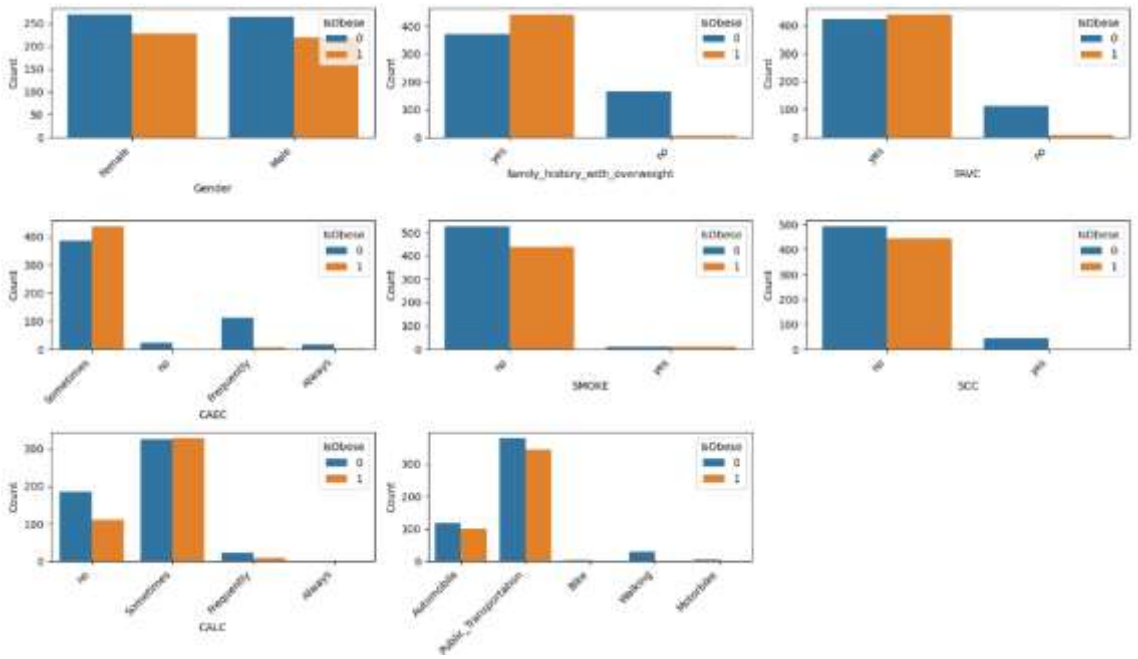
- **Variables categóricas**: Para las variables categóricas, se aplicó la imputación mediante la moda, utilizando el método SimpleImputer. Este enfoque sustituye los valores faltantes con el valor más frecuente en la columna correspondiente, lo que mantiene la distribución de las categorías y evita distorsionar los datos.

Relación de las variables con la variable objetivo

A continuación, se muestran las relaciones entre la variable objetivo y las variables numéricas y categóricas, rspectivamente:




IsObese
0
1
Count
NCP
CH2O
FAF
TUE



Creación de variables dummies

Para hacer que los datos sean compatibles con todos los modelos, convertimos las variables categóricas a variables dummies. También añadimos drop_first=True para que se creen tantas columnas como valores tenga cada variable menos uno, así mantenemos la información ahorrando tamaño y tiempo de ejecución a la hora de ejecutar modelos.

División en Conjunto de Entrenamiento y Prueba

Con los datos ya limpios, se procedió a dividir el conjunto en dos subconjuntos: uno de entrenamiento y otro de prueba. Esta división se realizó utilizando una proporción del 80% para entrenamiento y 20% para prueba, garantizando que el modelo pueda generalizar adecuadamente. La variable objetivo para esta tarea fue la columna correspondiente a la obesidad, que fue tratada como una variable binaria (obeso/no obeso).

1.- Análisis Comparativo de Modelos para Clasificación Binaria: Regresión Logística vs. Red Neuronal

Objetivo

El objetivo de este ejercicio es desarrollar modelos de clasificación para predecir si un individuo es obeso o no, utilizando regresión logística y redes neuronales. Inicialmente, se encuentra el mejor modelo de regresión logística mediante la optimización de sus hiperparámetros. Posteriormente, con las variables seleccionadas a partir de la regresión, se entrena una red neuronal optimizada para maximizar el accuracy.

A) Modelo de Regresión Logística

1. **Preprocesamiento y depuración de datos**: Dado que tanto la regresión logística como las redes neuronales son sensibles a la escala de las variables continuas, se realizó un preprocesamiento específico para mejorar la eficacia del modelo:

- **Normalización/Estandarización de Variables Continuas**: Se estandarizaron las variables continuas (Age, Height, Weight, NCP) utilizando StandardScaler de sklearn, lo cual asegura que todas las variables tengan media 0 y desviación estándar 1.
- **Transformación de variables categóricas**: Las variables categóricas (Gender, FAVC, CALC) fueron convertidas a variables dummies. Esto permite representar estas características como valores numéricos que pueden ser utilizados por los modelos.

```
from sklearn.preprocessing import StandardScaler

continuous_columns = ['Age', 'Height', 'Weight', 'FCVC', 'NCP', 'CH2O', 'FAF', 'TUE']
scaler = StandardScaler()
X_train[continuous_columns] = scaler.fit_transform(X_train[continuous_columns])
X_test[continuous_columns] = scaler.transform(X_test[continuous_columns])
```

2. **Optimización del modelo de regresión logística**: Se empleó GridSearchCV para optimizar los hiperparámetros de la regresión logística. Los siguientes parámetros fueron probados:
   - **C**: Este parámetro controla la regularización del modelo. Un rango amplio de valores (de 0.001 a 1000) fue probado para evitar el sobreajuste y encontrar un equilibrio entre la complejidad del modelo y su capacidad para generalizar.
   - **penalty**: Se probaron normas de penalización l1 y l2, que corresponden a las técnicas Lasso y Ridge.
   - **max_iter**: Este parámetro controla el número máximo de iteraciones que el solver puede realizar. Un rango entre 100 y 700 fue explorado para asegurar la convergencia.
   - **solver**: Se probaron distintos algoritmos de optimización (newton-cg, lbfgs, liblinear, sag, saga) para evaluar su rendimiento con los datos.

```
# Matriz de parametros para el modelo de regresion logistica
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
    'penalty': ['l1', 'l2'],
    'max_iter': list(range(100,800,100)),
    'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
}

logreg = LogisticRegression(random_state=seed)

# Grid search para encontrar los mejores parametros
grid_search = GridSearchCV(estimator=logreg, param_grid=param_grid, cv=3, n_jobs=-1, verbose=2)
grid_search.fit(X_train, y_train)
```

Resultados del modelo: La mejor configuración fue {'C': 1, 'max_iter': 100, 'penalty': 'l1', 'solver': 'liblinear'}, logrando una precisión (accuracy) de 0.995 en el conjunto de test. La matriz de confusión mostró solo un error de predicción, y el classification report arrojó un f1-score muy alto para ambas clases (0 y 1), indicando un excelente desempeño.

3. **Selección de variables más relevantes**: A partir de los coeficientes del modelo, se identificaron las características con mayor influencia para predecir obesidad. Las variables seleccionadas fueron utilizadas para entrenar la red neuronal.

Características seleccionadas: ['Age', 'Height', 'Weight', 'NCP', 'Gender_Male', 'FAVC_yes', 'CALC_Sometimes'].

```
# Mejores caracteristicas segun el modelo de regresion logistica
best_features = X_train.columns[best_logreg.coef_[0] != 0]

print(f'Best features: {best_features}')
```

```
Best features: Index(['Age', 'Height', 'Weight', 'NCP', 'Gender_Male', 'FAVC_yes',
       'CALC_Sometimes'],
      dtype='object')
```

B) Modelo de Red Neuronal

1. **Optimización de la red neuronal con GridSearchCV**: Con las características relevantes seleccionadas, se desarrolló una red neuronal utilizando MLPClassifier. Para ajustar su arquitectura y encontrar el mejor rendimiento, se realizó una búsqueda de cuadrícula (GridSearchCV) con los siguientes parámetros:
    - **hidden_layer_sizes**: Se probaron diferentes combinaciones de neuronas en una o dos capas ocultas para capturar la complejidad del patrón en los datos.
    - **activation**: Se utilizaron tanh y relu como funciones de activación para controlar cómo se propaga la información a través de la red.
    - **solver**: Se compararon sgd (descenso de gradiente estocástico) y adam (descenso de gradiente basado en momentum).
    - **alpha**: Este parámetro controla la regularización L2. Diferentes valores fueron probados para prevenir el sobreajuste y encontrar un equilibrio entre complejidad y generalización.
    - **learning_rate**: Para controlar la tasa de aprendizaje, se probaron configuraciones constantes (constant) y adaptativas (adaptive).

```
# Matriz de parametros para el modelo de red neuronal
param_grid_nn = {
    'hidden_layer_sizes': [(10,), (15,), (20,), (25,), (10,10), (15,15), (20,20), (25, 25)],
    'activation': ['tanh', 'relu'],
    'solver': ['sgd', 'adam'],
    'alpha': [0.0001, 0.01, 0.05],
    'learning_rate': ['constant','adaptive'],
}

mlp = MLPClassifier(max_iter=1000, random_state=seed)

# Grid search para encontrar los mejores parametros
grid_search_nn = GridSearchCV(estimator=mlp, param_grid=param_grid_nn, cv=5, n_jobs=-1, verbose=2, scoring='accuracy')
grid_search_nn.fit(X_train[best_features], y_train)

# Los mejores parametros obtenidos por el grid search
best_params_nn = grid_search_nn.best_params_
print(f'Best parameters for NN: {best_params_nn}')
```

Resultados del modelo: La red neuronal con la mejor arquitectura fue {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (15, 15), 'learning_rate': 'constant', 'solver': 'adam'}. Este modelo alcanzó una precisión (accuracy) de 0.995, con una matriz de confusión que muestra solo un error, y un classification report con precision, recall y f1-score cercanos a 1.

2. **Comparación de modelos y análisis de estabilidad**: La comparación entre el modelo de regresión logística y la red neuronal muestra que ambos ofrecen resultados similares y excepcionales en términos de precisión. Tanto la regresión logística como la red neuronal alcanzaron un accuracy de 0.995, lo que indica que ambos modelos capturan adecuadamente la estructura de los datos. La elección de solver, regularización (C para regresión y alpha para la red neuronal), y la arquitectura de la red (capas y neuronas) fueron críticos para lograr este rendimiento.

    - **Regresión Logística**: Accuracy de 0.995 con un único error de predicción.

```
Accuracy: 0.9949238578680203
---------------------------------
Confusion Matrix:
 [[110   0]
 [  1  86]]
---------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       110
           1       1.00      0.99      0.99        87

    accuracy                           0.99       197
   macro avg       1.00      0.99      0.99       197
weighted avg       0.99      0.99      0.99       197
```

    - **Red Neuronal**: Accuracy de 0.995 con un único error de predicción.

```
Accuracy for NN: 0.9949238578680203
---------------------------------
Confusion Matrix:
 [[110   0]
 [  1  86]]
---------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       110
           1       1.00      0.99      0.99        87

    accuracy                           0.99       197
   macro avg       1.00      0.99      0.99       197
weighted avg       0.99      0.99      0.99       197
```

Conclusión

Ambos modelos (regresión logística y red neuronal) alcanzaron un accuracy de 0.995, con solo un error de predicción. Aunque la red neuronal logra capturar relaciones complejas en los datos, la regresión logística muestra un desempeño comparable con menor complejidad computacional. La selección final del modelo depende del contexto de uso; si se busca simplicidad y rapidez, la regresión logística es la opción preferible. Sin embargo, si se busca capturar relaciones más complejas o generalizar mejor en otros escenarios, la red neuronal es una excelente opción.

2.- Optimización de la Red Neuronal mediante Selección de Variables y Ajuste para Maximizar el AUC

Objetivo

El objetivo de este ejercicio es aplicar un proceso de selección de características usando SelectKBest, eligiendo las cuatro variables más relevantes (k=4). Posteriormente, se entrenará una red neuronal con estas características seleccionadas, optimizando su arquitectura para obtener el mejor desempeño en términos de la métrica AUC.

Selección de Variables con SelectKBest

1. **Aplicación del método de selección SelectKBest**: Se utilizó SelectKBest con score_func=f_classif para seleccionar las cuatro mejores características del conjunto de datos (k=4). Este método calcula la importancia de cada variable en relación con la variable objetivo, y selecciona aquellas que aportan más información.

```
# Aplicamos SelectKBest para extraer las 4 mejores variables
bestfeatures = SelectKBest(score_func=f_classif, k=4)
bestfeatures.fit(X_train,y_train)

# 4 mejores
selected_features = X.columns[bestfeatures.get_support()]

X_selected = X[selected_features]

print("Selected Features:")
for f in selected_features:
    print(f)
```

Características seleccionadas: Las variables identificadas como más relevantes fueron Weight, family_history_with_overweight_yes, CAEC_Frequently y CAEC_Sometimes.

Esto reduce la dimensionalidad del conjunto de datos, manteniendo solo las variables más relevantes para el entrenamiento del modelo de red neuronal.

2. **Entrenamiento del modelo de red neuronal** Una vez seleccionadas las cuatro variables más relevantes, se entrenó un modelo de red neuronal (MLPClassifier) con diferentes arquitecturas para optimizar su desempeño en términos de AUC. Las siguientes configuraciones de capas ocultas fueron probadas:

- **Tamaño de capas ocultas**: Se probaron diferentes tamaños de neuronas, tanto en una capa oculta ((10,), (15,), (20,), (25,), (30,)) como en dos capas ocultas ((10, 10), (15, 15), (20, 20), (25, 25)).
- **Proceso de validación cruzada**: Para evaluar cada arquitectura, se utilizó KFold con 5 divisiones (n_splits=5), de manera que se pudiera medir la estabilidad y generalización del modelo.

```
hidden_layer_sizes = [(10,), (15,), (20,), (25,), (30,), (10,10), (15,15), (20,20), (25,25)]

results = []
for hl in hidden_layer_sizes:
    mlp = MLPClassifier(hidden_layer_sizes=hl, max_iter=1000, random_state=seed)
    cv_technique = KFold(n_splits=5, shuffle=True, random_state=seed)
    r = cross_validate(mlp, X_selected, y, cv=cv_technique, scoring=['accuracy', 'precision', 'recall', 'f1', 'roc_au

    print("--------------------------------")
    print(f"Score for hidden layer size: {hl}")
    print(f"Accuracy: {r['test_accuracy'].mean()}")
    print(f"Precision: {r['test_precision'].mean()}")
    print(f"Recall: {r['test_recall'].mean()}")
    print(f"F1 Score: {r['test_f1'].mean()}")
    print(f"ROC AUC Score: {r['test_roc_auc'].mean()}")

    results.append(r['test_roc_auc'])
```

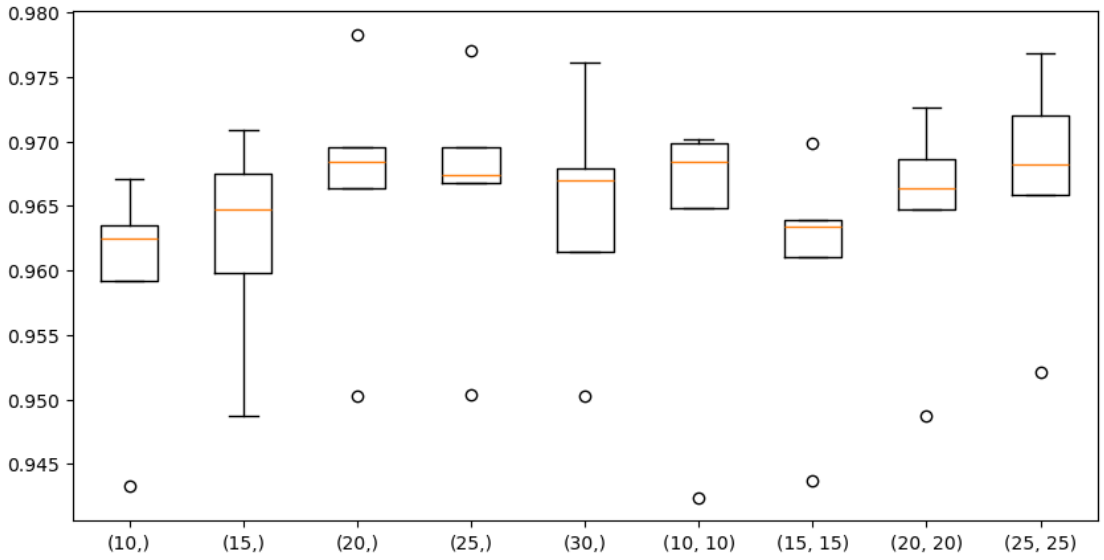
Los resultados de la validación cruzada para cada arquitectura fueron evaluados en términos de accuracy, precision, recall, f1-score, y principalmente, ROC AUC.

Resultados de las diferentes arquitecturas:

- Los modelos entrenados con una única capa oculta de tamaño (15,) y (30,) lograron altos valores de ROC AUC, pero con mayor variabilidad.
- Las arquitecturas con dos capas ocultas de tamaño (20, 20) y (25, 25) demostraron ser más estables, obteniendo valores de ROC AUC consistentemente altos, alrededor de 0.967.

```
---------------------------------
Score for hidden layer size: (10,)
Accuracy: 0.8910338754791256
Precision: 0.9089363475081985
Recall: 0.8485286160536077
F1 Score: 0.8762152141745421
ROC AUC Score: 0.9591288645393746
---------------------------------
Score for hidden layer size: (15,)
Accuracy: 0.8930591525950481
Precision: 0.9112710018477955
Recall: 0.8506119493869411
F1 Score: 0.8785780644780159
ROC AUC Score: 0.9623336758644262
---------------------------------
Score for hidden layer size: (20,)
Accuracy: 0.8859266549259299
Precision: 0.8910112466039048
Recall: 0.8569152493498627
F1 Score: 0.8729633890584566
ROC AUC Score: 0.9665920013404594
---------------------------------
Score for hidden layer size: (25,)
Accuracy: 0.8961203770848443
Precision: 0.9125967697020329
Recall: 0.8547647117154542
F1 Score: 0.8820175650714541
ROC AUC Score: 0.9662328614552651
---------------------------------
Score for hidden layer size: (30,)
Accuracy: 0.8981611934113747
Precision: 0.9206293086785646
Recall: 0.8505308407477121
F1 Score: 0.8833617956038106
ROC AUC Score: 0.9645353825256551
---------------------------------
Score for hidden layer size: (10, 10)
Accuracy: 0.8808297938464726
Precision: 0.8911184634700872
Recall: 0.8485286160536077
F1 Score: 0.8669736975782083
ROC AUC Score: 0.9631131507272481
---------------------------------
Score for hidden layer size: (15, 15)
Accuracy: 0.8859266549259297
Precision: 0.8982360354910263
Recall: 0.8506119493869411
F1 Score: 0.8719738133891634
ROC AUC Score: 0.960385769963761
---------------------------------
Score for hidden layer size: (20, 20)
Accuracy: 0.8798042059463379
Precision: 0.8983981354726129
Recall: 0.8391590861711371
F1 Score: 0.8657222226856138
ROC AUC Score: 0.9642045568352116
---------------------------------
Score for hidden layer size: (25, 25)
Accuracy: 0.8920387444317829
Precision: 0.9059324065707044
Recall: 0.8547647117154542
F1 Score: 0.8784811491970004
ROC AUC Score: 0.9669925427747446
```

A continuación, se muestra la representación gráfica de los resultados:



3. **Selección del modelo óptimo de red neuronal** La arquitectura seleccionada fue la red neuronal con dos capas ocultas de tamaño (20, 20), ya que ofreció un balance entre rendimiento (ROC AUC) y estabilidad.

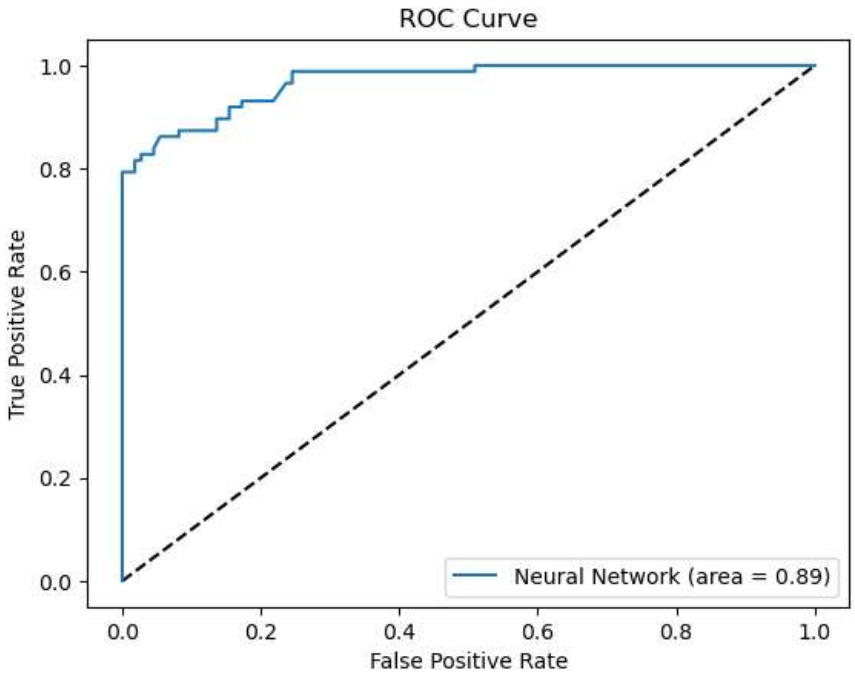
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=seed)

X_train_selected = X_train[selected_features]
X_test_selected = X_test[selected_features]

# El modelo con el mejor resultado fue con 2 capas ocultas de 20 neuronas cada una, utilziando las 4 mejores variables
mlp = MLPClassifier(hidden_layer_sizes=(20,20), max_iter=1000, random_state=seed)

# Entranamos el modelo
mlp.fit(X_train_selected, y_train)

# Validmos con test
y_pred = mlp.predict(X_test_selected)
y_pred_prob = mlp.predict_proba(X_test_selected)[:,1]

# Evaluacion del modelo
# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

Evaluación del modelo: El modelo fue evaluado en el conjunto de test, obteniendo los siguientes resultados:

- **Accuracy**: 0.89
- **ROC AUC**: 0.89, lo cual indica un buen rendimiento en la separación de las clases (obeso/no obeso).
- **Matriz de Confusión**: Mostró algunos errores de clasificación (10 falsos negativos y 11 falsos positivos), indicando una precisión similar para ambas clases.

```
# ROC AUC Score
roc_auc = roc_auc_score(y_test, y_pred)
print(f"ROC AUC Score: {roc_auc}")
```

A continuación, se muestra la curva del ROC AUC score:



Visualización de la ROC Curve: La ROC Curve muestra un área bajo la curva (AUC) de 0.89, lo que sugiere que el modelo tiene un buen poder de discriminación.

Conclusión

El proceso de selección de características con SelectKBest permitió identificar las variables más relevantes, lo que facilitó el entrenamiento de un modelo de red neuronal eficiente. La red neuronal con dos capas ocultas de tamaño (20, 20) fue la arquitectura que mejor balanceó estabilidad y rendimiento, logrando un ROC AUC de 0.89 en el conjunto de test. Aunque la accuracy es buena, la métrica AUC es la que guía la optimización del modelo, asegurando que la red neuronal sea capaz de discriminar adecuadamente entre ambas clases.

Con esto, se demuestra la importancia de la selección de características y la optimización cuidadosa de la arquitectura del modelo para maximizar el rendimiento en problemas de clasificación binaria.

3.- Comparación de modelos

Objetivo

El objetivo de este apartado es comparar los modelos candidatos ganadores de los ejercicios 1 y 2: el modelo de **Regresión Logística** y la **Red Neuronal**. Ambos modelos fueron seleccionados en base a su desempeño, principalmente en términos de accuracy y AUC. Esta comparación se llevará a cabo evaluando sus métricas de rendimiento, estabilidad, complejidad, y capacidad para generalizar.

Modelos a Comparar

1. **Regresión Logística**: Seleccionado como el mejor modelo en el ejercicio 1, con parámetros óptimos {'C': 1, 'max_iter': 100, 'penalty': 'l1', 'solver': 'liblinear'}. Este modelo alcanzó un accuracy de 0.995 y mostró un ROC AUC elevado, lo que indica una alta capacidad para discriminar entre clases (obeso/no obeso).
2. **Red Neuronal**: Seleccionada en el ejercicio 2 tras un proceso de SelectKBest para reducir el número de variables a las más relevantes y optimizada en términos de AUC. La mejor arquitectura encontrada fue (20, 20) (dos capas ocultas con 20 neuronas cada una), con un accuracy de 0.89 y un ROC AUC de 0.89.

Comparación en Términos de Métricas de Rendimiento

1. **Accuracy**:
    - La **Regresión Logística** logra un accuracy de 0.995, lo que significa que el modelo clasifica correctamente el 99.5% de las muestras. Esto indica que el modelo tiene un alto rendimiento general en términos de predicción de la clase correcta.
    - La **Red Neuronal** alcanzó un accuracy de 0.89. Aunque esto es un buen resultado, es significativamente menor que el de la regresión logística, lo

que sugiere que la red neuronal tuvo más dificultad para generalizar con las variables seleccionadas.

2. **ROC AUC**:
    - La **Regresión Logística** tiene un ROC AUC cercano a 1, lo que indica que el modelo tiene un poder de discriminación muy alto para distinguir entre individuos obesos y no obesos.
    - La **Red Neuronal** tiene un ROC AUC de 0.89, lo cual sigue siendo un valor aceptable y muestra que el modelo tiene una buena capacidad de discriminación, pero no tan fuerte como la regresión logística.
3. **Estabilidad del Modelo**:
    - La **Regresión Logística** es un modelo más simple y lineal, lo que significa que es menos probable que sea afectado por sobreajuste, especialmente con la penalización l1 seleccionada. Esto da como resultado un modelo más estable que funciona bien con diferentes subconjuntos de datos.
    - La **Red Neuronal**, aunque más compleja y capaz de capturar relaciones no lineales, mostró más variabilidad en su rendimiento a través de diferentes arquitecturas (como se evidenció en el boxplot del ROC AUC con diferentes capas y neuronas). Esto sugiere que, aunque puede ser un modelo potente, su estabilidad depende más de la arquitectura seleccionada y puede ser más sensible a cambios en los datos de entrada.

Comparación de la Complejidad y Capacidad de Generalización

1. **Complejidad**:
    - La **Regresión Logística** es inherentemente más sencilla, con solo unos pocos hiperparámetros para ajustar. Esto se traduce en un modelo más interpretable y menos costoso computacionalmente, ideal cuando se necesita simplicidad y transparencia.
    - La **Red Neuronal**, con múltiples capas y neuronas, es mucho más compleja. Aunque esto le da mayor capacidad para capturar patrones no lineales y complejos en los datos, también requiere más recursos computacionales y tiempo para ajustar los hiperparámetros y entrenar el modelo. Esta complejidad puede llevar a un mejor rendimiento, pero también implica un mayor riesgo de sobreajuste si no se manejan adecuadamente las regularizaciones y la arquitectura.
2. **Capacidad de Generalización**:
    - La **Regresión Logística** muestra una excelente capacidad de generalización con un accuracy y ROC AUC cercanos al máximo, lo que indica que puede generalizar bien a nuevos datos sin perder precisión.
    - La **Red Neuronal** también muestra buena generalización, pero su accuracy más bajo sugiere que, aunque es capaz de capturar relaciones complejas, no es tan eficaz con las cuatro variables seleccionadas como lo

es la regresión logística con su conjunto completo de variables. Además, la necesidad de un proceso de validación cruzada para ajustar la arquitectura resalta la necesidad de más cuidado para asegurar su capacidad de generalización.

Evaluación Final y Selección del Modelo Ganador

Ambos modelos tienen sus ventajas y desventajas. La elección del modelo ganador depende en gran medida del objetivo y del contexto de la aplicación:

- **Si se busca simplicidad, estabilidad y rapidez** en la implementación, la **Regresión Logística** es claramente la mejor opción. Su precisión superior (accuracy de 0.995) y su ROC AUC cercano a 1 muestran que no solo generaliza bien sino que también lo hace con una estructura de modelo simple y fácil de interpretar.
- **Si se busca capturar relaciones más complejas en los datos** y se dispone de recursos computacionales para ajustar cuidadosamente la arquitectura, la **Red Neuronal** puede ser una buena opción. Aunque su accuracy es menor, sigue mostrando un buen poder de discriminación (ROC AUC de 0.89).

Conclusión

En este caso particular, dado que la **Regresión Logística** muestra un rendimiento superior tanto en accuracy como en AUC, es más estable y computacionalmente más simple, se selecciona como el modelo ganador para esta tarea de clasificación. La **Red Neuronal** sigue siendo un modelo potente que podría considerarse para datos más complejos o no lineales, pero en este caso, no aporta suficiente mejora para superar a la regresión logística.

La comparación ilustra la importancia de evaluar diferentes aspectos de los modelos, incluyendo su rendimiento, complejidad y capacidad de generalización, para seleccionar la mejor solución para un problema de clasificación.

4.- Optimización y Análisis de un Modelo de Árbol de Decisión

Objetivo

El objetivo de este ejercicio es optimizar un modelo de árbol de decisión para lograr el mejor rendimiento posible según cuatro métodos de validación de la bondad de clasificación. Una vez obtenido el árbol de decisión óptimo, se representarán sus reglas en formato texto y se analizará la importancia de las variables.

Proceso de Búsqueda Paramétrica

1. **Selección de hiperparámetros**: Para encontrar el mejor árbol de decisión, se utilizó GridSearchCV con los siguientes parámetros:
    - **criterion**: Se probaron las funciones gini y entropy para calcular la pureza de los nodos.

- **max_depth**: Se evaluaron profundidades del árbol de 1 a 9 para controlar la complejidad del modelo y evitar el sobreajuste.
- **min_samples_split**: Se exploraron valores de 1 a 9 para definir el número mínimo de muestras requeridas para dividir un nodo.

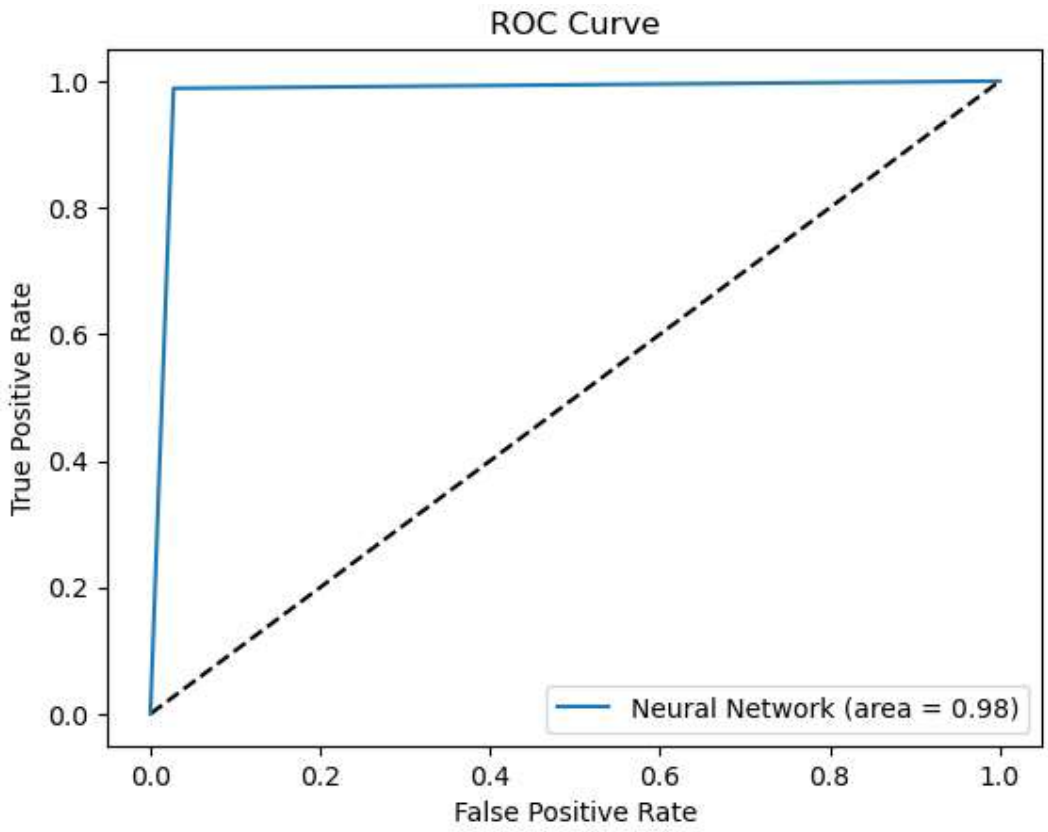
```
# Matriz de parámetros a probar
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': range(1, 10),
    'min_samples_split': range(1, 10)
}

# Realizamos la búsqueda de mejores parámetros
grid = GridSearchCV(DecisionTreeClassifier(random_state=seed), param_grid, cv=5)
grid.fit(X_train, y_train)
```

Resultados de la búsqueda: Los mejores parámetros encontrados fueron {'criterion': 'entropy', 'max_depth': 5, 'min_samples_split': 2}. Este modelo fue ajustado con los datos de entrenamiento y validado con datos de test.

2. **Evaluación del modelo**:
   - **Accuracy**: El modelo obtuvo un accuracy de 0.979, lo que indica un muy buen desempeño general.
   - **Matriz de confusión**: Mostró 3 falsos positivos y 1 falso negativo, evidenciando que el modelo clasifica correctamente la gran mayoría de las muestras.
   - **ROC AUC**: La curva ROC muestra un área bajo la curva cercana a 1, lo que indica una alta capacidad de discriminación entre las clases.

```
Accuracy: 0.9796954314720813
Precision: 0.9662921348314607
Recall: 0.9885057471264368
F1 Score: 0.9772727272727273
---------------------------------
Confusion Matrix:
 [[107   3]
 [  1  86]]
---------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.97      0.98       110
           1       0.97      0.99      0.98        87

    accuracy                           0.98       197
   macro avg       0.98      0.98      0.98       197
weighted avg       0.98      0.98      0.98       197
```

```
roc_auc = roc_auc_score(y_test, y_pred)
y_pred_prob = clf.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred)

# Representamos gráficamente la curva ROC
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr, label='Neural Network (area = %0.2f)' % roc_auc)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.show()
```



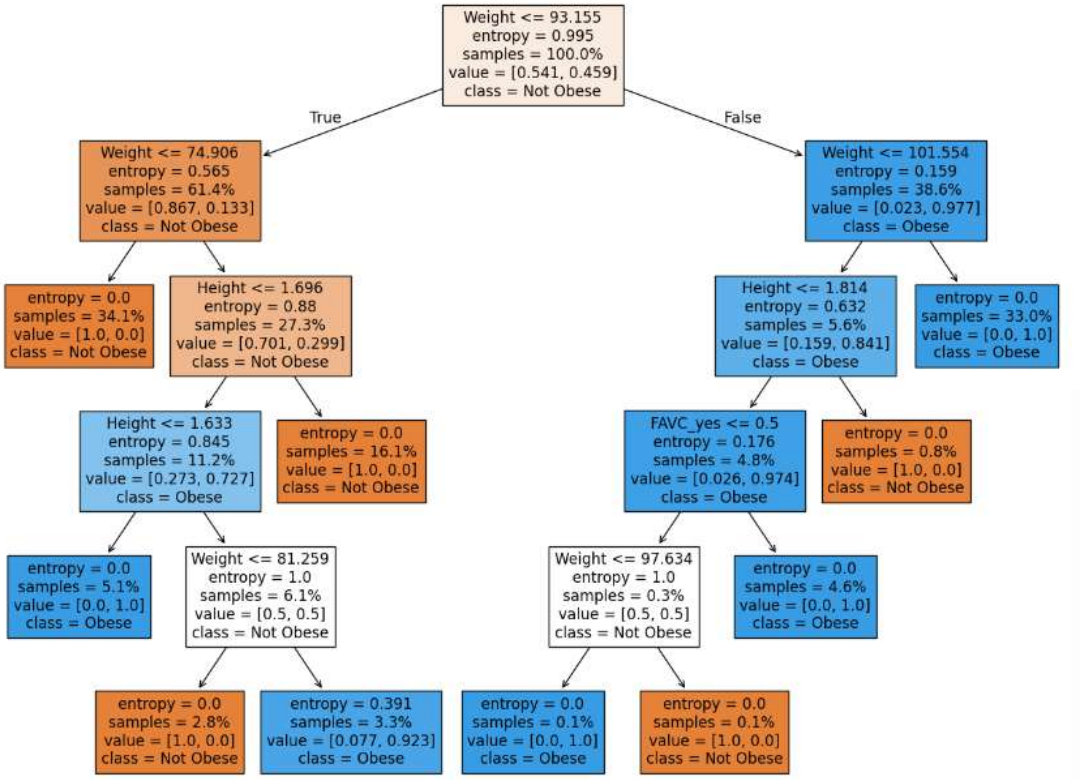
Análisis de Sobreajuste y Validación Cruzada

Una evaluación importante para los árboles de decisión es comparar la bondad de ajuste entre los datos de entrenamiento y los datos de test, ya que los árboles pueden sobreajustarse fácilmente. Se observó que el accuracy y otras métricas (como precision, recall y f1-score) son consistentes entre los datos de entrenamiento y test, lo que sugiere que el modelo generaliza bien y no muestra señales de sobreajuste.

Además, se empleó cross-validation para validar la robustez del modelo y confirmar que los resultados obtenidos son estables a través de diferentes divisiones del conjunto de datos.

Representación del Árbol Ganador

1. **Visualización gráfica del árbol**: El árbol de decisión fue representado gráficamente para mostrar cómo se realiza la clasificación en cada nodo. Se puede observar que el árbol realiza divisiones con base principalmente en la variable Weight, seguida por Height. Estas divisiones reflejan cómo el árbol separa las clases obesas y no obesas.



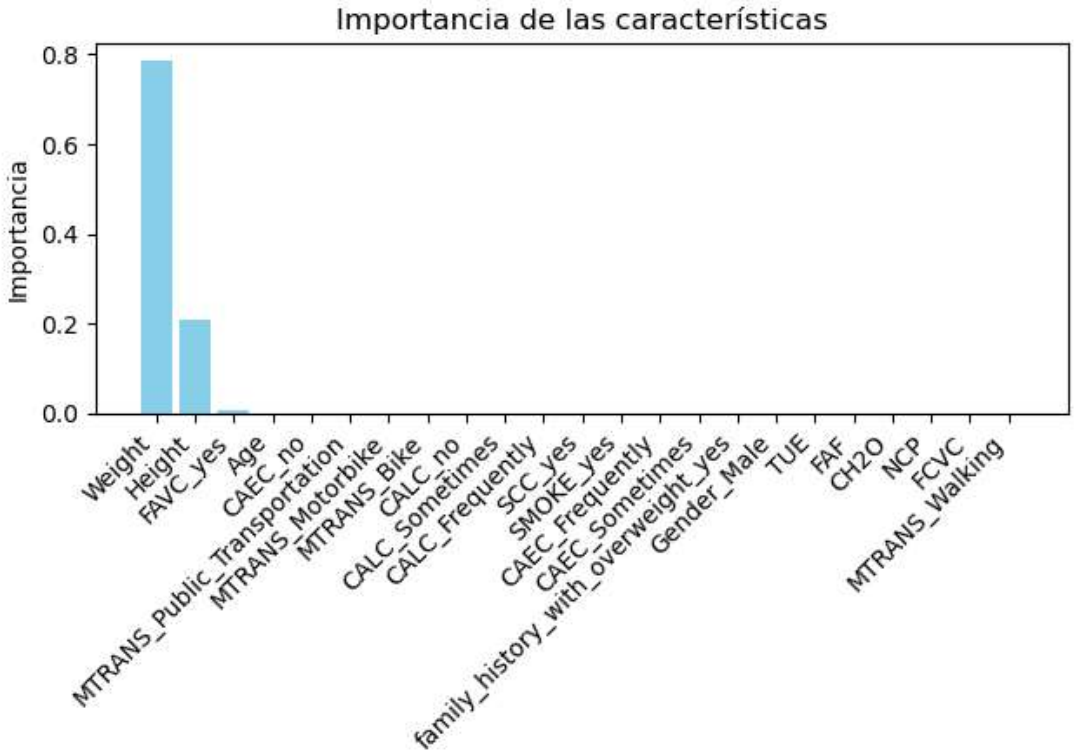
2. **Reglas del árbol en formato texto**: Para entender las decisiones tomadas por el árbol, sus reglas fueron representadas en formato texto. Esto muestra cómo las diferentes características y sus umbrales conducen a una clasificación de obeso o no obeso. La estructura del árbol permite observar cómo cada nodo toma una decisión binaria basada en un valor de una característica.

Importancia de las Variables

El árbol de decisión también permite analizar la importancia de las variables utilizadas en la clasificación:

- **Weight**: Fue la variable más importante, contribuyendo significativamente a la separación de las clases, con una importancia de 0.784.
- **Height**: Fue la segunda variable más relevante, aunque con una importancia significativamente menor (0.209).
- **Otras variables**: Variables como FAVC_yes mostraron una contribución marginal, y muchas otras tuvieron una importancia de 0, indicando que no influyeron en las decisiones del árbol.

A continuación, se muestra una representación gráfica de las importancia de las distintas variables:



Conclusión

El proceso de búsqueda paramétrica y ajuste del árbol de decisión resultó en un modelo con excelente capacidad de discriminación y precisión, al tiempo que mostró estabilidad y robustez a través de diferentes métodos de validación. El árbol de decisión final se basó principalmente en el Weight y Height de los individuos para realizar sus predicciones, reflejando la importancia de estas variables en la clasificación de obesidad.

Este ejercicio demuestra cómo la selección de parámetros y el análisis detallado de la estructura del árbol permiten desarrollar modelos interpretables y efectivos para problemas de clasificación.

5.- Afinamiento de Modelos de Ensamblado con Bagging y Random Forest

Objetivo

El objetivo de este ejercicio es encontrar el mejor modelo para Bagging y Random Forest en términos de accuracy, realizando una búsqueda exhaustiva de hiperparámetros. Para ambos modelos, se decidió forzar bootstrap=True durante la búsqueda para poder calcular el oob_score (Out-Of-Bag Score), lo que permite evaluar la estabilidad y capacidad de generalización del modelo.

A) Modelo de Bagging

1. **Selección de Hiperparámetros**: Para el **BaggingClassifier**, se realizó una búsqueda de parámetros utilizando GridSearchCV y se estableció bootstrap=True para habilitar el cálculo de oob_score. Los parámetros ajustados fueron:
    - **n_estimators**: Número de árboles en el ensamble, con valores de 10 a 200.
    - **max_samples**: Proporción de muestras utilizadas para entrenar cada árbol, probando valores de 0.5, 0.7 y 1.0.
    - **max_features**: Proporción de características seleccionadas para cada árbol.
    - **bootstrap_features**: Controla si las características son seleccionadas con reemplazo.

```
# Matriz de parametros para el BagggingClassifier
param_grid_bagging = {
    'n_estimators': [10, 50, 100, 200],
    'max_samples': [0.5, 0.7, 1.0],
    'max_features': [0.5, 0.7, 1.0],
    # 'bootstrap': [True, False],
    'bootstrap_features': [True, False]
}

# Buscamos los mejores parametros para el BaggingClassifier
grid_bagging = GridSearchCV(BaggingClassifier(random_state=seed), param_grid_bagging, cv=5, scoring='accuracy')
grid_bagging.fit(X_train, y_train)
```

2. **Resultados**: Los mejores parámetros obtenidos fueron:
    - {'bootstrap_features': False, 'max_features': 1.0, 'max_samples': 0.7, 'n_estimators': 100}
3. **Precisión (accuracy) y oob_score**:
    - **accuracy**: 0.985 en el conjunto de test, lo que indica un excelente rendimiento de clasificación.
    - **oob_score**: 0.986, muy cercano a la precisión en el test, lo que demuestra la estabilidad y capacidad de generalización del modelo.
4. **Evaluación del Modelo Final**:
    - La matriz de confusión mostró solo 3 errores de predicción (2 falsos positivos y 1 falso negativo).
    - El classification report indicó valores de precision, recall y f1-score cercanos a 1 para ambas clases, lo que refleja la gran capacidad del modelo para clasificar correctamente tanto individuos obesos como no obesos.

```python
# Entrenamos el BaggingClassifier con los mejores parametros
print(f'Best parameters for Bagging Classifier: {grid_bagging.best_params_}')
bagging_clf = BaggingClassifier(**grid_bagging.best_params_, random_state=seed, oob_score=True, bootstrap=True)
bagging_clf.fit(X_train, y_train)

# Evaluamos el BaggingClassifier
y_pred_bagging = bagging_clf.predict(X_test)
print(f'Bagging Classifier Accuracy: {accuracy_score(y_test, y_pred_bagging)}')
print(f'Bagging OOB Score: {bagging_clf.oob_score_}')

# Bagging model Validation
print("---------------------------------")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_bagging))
print("---------------------------------")
print("Classification Report:\n", classification_report(y_test, y_pred_bagging))
```

```
Bagging Classifier Accuracy: 0.9847715736040609
Bagging OOB Score: 0.9859872611464968
---------------------------------
Confusion Matrix:
 [[108   2]
 [  1  86]]
---------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.98      0.99       110
           1       0.98      0.99      0.98        87

    accuracy                           0.98       197
   macro avg       0.98      0.99      0.98       197
weighted avg       0.98      0.98      0.98       197
```

Resumen:

- El modelo de Bagging con bootstrap=True y oob_score demostró un equilibrio sólido entre precisión y generalización, con un rendimiento general muy alto.

B) Modelo de Random Forest

1. **Selección de Hiperparámetros**: Para el **RandomForestClassifier**, se utilizó GridSearchCV para ajustar los parámetros clave, manteniendo bootstrap=True para calcular el oob_score:
   - **n_estimators**: Número de árboles, con valores entre 50 y 200.
   - **max_depth**: Profundidad máxima de los árboles, para controlar la complejidad.
   - **min_samples_leaf** y **min_samples_split**: Tamaño mínimo de muestras requerido para dividir nodos y mantener hojas.
   - **criterion**: Métrica de pureza del nodo (gini o entropy).

```
# Matriz de parametros para el RandomForestClassifier
param_grid_rf = {
    'n_estimators' : [50,100,150,200],
    'max_depth': [3, 5, 10],
    'min_samples_leaf' : [3,10,30],
    'min_samples_split': [5, 10, 20, 50],
    'criterion': ["gini", "entropy"]
}

# Buscamos los mejores parametros para el RandomForestClassifier
grid_rf = GridSearchCV(RandomForestClassifier(random_state=seed), param_grid_rf, cv=5, scoring='accuracy')
grid_rf.fit(X_train, y_train)

print(f'Best parameters for Random Forest Classifier: {grid_rf.best_params_}')
```

2. **Resultados**: Los mejores parámetros obtenidos fueron:
    - {'criterion': 'entropy', 'max_depth': 10, 'min_samples_leaf': 3, 'min_samples_split': 5, 'n_estimators': 50}
3. **Precisión (accuracy) y oob_score**:
    - **accuracy**: 0.959 en el conjunto de test.
    - **oob_score**: 0.972, mostrando que el modelo generaliza bien, aunque con una ligera diferencia respecto al conjunto de test.
4. **Evaluación del Modelo Final**:
    - La matriz de confusión mostró 8 errores de predicción (2 falsos positivos y 6 falsos negativos), con un buen equilibrio entre ambas clases.
    - El classification report mostró una precisión alta para ambas clases, aunque ligeramente menor que el modelo de Bagging.

```
# Entrenamos el RandomForestClassifier con los mejores parametros
rf_clf = RandomForestClassifier(**grid_rf.best_params_, random_state=seed, oob_score=True, bootstrap=True)
rf_clf.fit(X_train, y_train)

# Evaluación de RandomForestClassifier
y_pred_rf = rf_clf.predict(X_test)
print(f'Random Forest Classifier Accuracy: {accuracy_score(y_test, y_pred_rf)}')
print(f'OOB Score: {rf_clf.oob_score_}')


# RandomForest model Validation
print("---------------------------------")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
print("---------------------------------")
print("Classification Report:\n", classification_report(y_test, y_pred_rf))
```

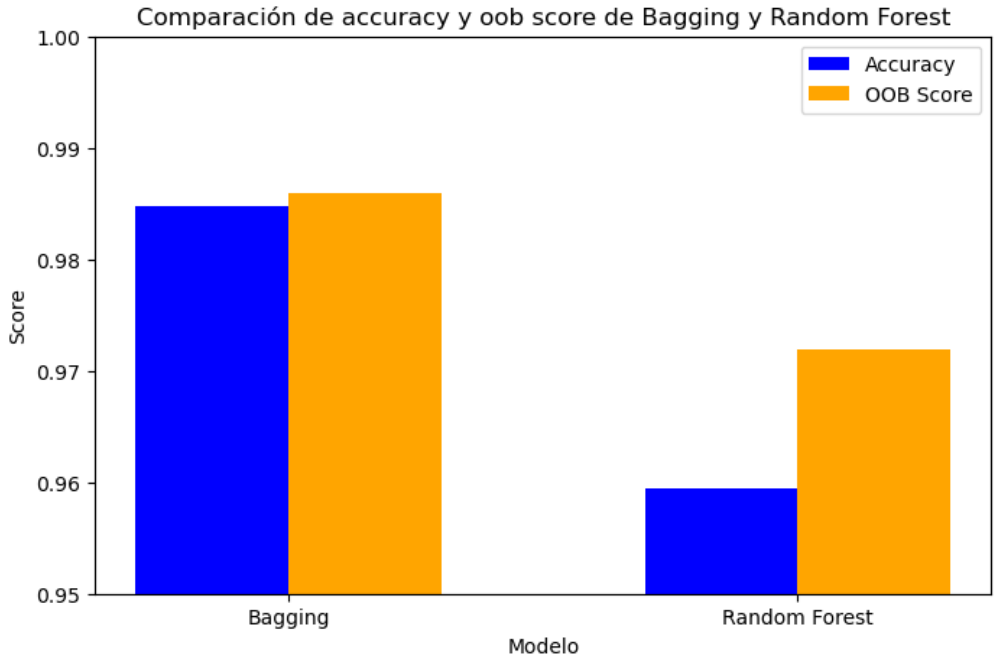
```
Random Forest Classifier Accuracy: 0.9593908629441624
OOB Score: 0.9719745222929936
--------------------------------
Confusion Matrix:
 [[108   2]
 [  6  81]]
--------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.98      0.96       110
           1       0.98      0.93      0.95        87

    accuracy                           0.96       197
   macro avg       0.96      0.96      0.96       197
weighted avg       0.96      0.96      0.96       197
```

Resumen:

- El modelo de Random Forest muestra un buen rendimiento y capacidad de generalización, con accuracy y oob_score consistentes. Aunque fue superado ligeramente por el modelo de Bagging, sigue siendo un modelo robusto y efectivo.

Comparación y Conclusión

1. **Rendimiento**:
   - **BaggingClassifier**: Logró un accuracy de 0.985 y un oob_score de 0.986, lo que indica que el modelo clasifica correctamente la gran mayoría de los datos y generaliza bien.
   - **RandomForestClassifier**: Alcanzó un accuracy de 0.959 y un oob_score de 0.972, también mostrando buen rendimiento, aunque un poco menor comparado con Bagging.



2. **Matriz de Confusión y Métricas**:
    - Ambos modelos tienen un accuracy alto, con métricas de precision, recall y f1-score cercanas a 1. Sin embargo, Bagging mostró una ligera ventaja en todos estos aspectos.
3. **Conclusión Final**:
    - Dado que el **BaggingClassifier** mostró un rendimiento ligeramente superior tanto en accuracy como en oob_score, es considerado el modelo ganador en este ejercicio.
    - La comparación demuestra la importancia de ajustar correctamente los hiperparámetros y de evaluar el oob_score para validar la capacidad de generalización de modelos basados en ensambles.

La inclusión del oob_score proporcionó una evaluación adicional del rendimiento y estabilidad de los modelos, asegurando que no solo funcionen bien con los datos de entrenamiento, sino que también generalicen adecuadamente a nuevos datos.

6.- Optimización de Modelos de Gradient Boosting y XGBoost

Objetivo

El objetivo de este ejercicio es ajustar modelos de **Gradient Boosting** y **XGBoost** para maximizar la precisión (accuracy). Durante la búsqueda de hiperparámetros, se amplió el rango de valores para explorar configuraciones más extremas y garantizar que los parámetros característicos de estas técnicas fueran evaluados con suficiente profundidad.

A) Modelo de Gradient Boosting

1. **Selección de Hiperparámetros**: Para el modelo de **Gradient Boosting**, se realizó una búsqueda de hiperparámetros con GridSearchCV. La matriz de hiperparámetros utilizada fue:
    - **n_estimators**: Número de árboles en el modelo (10, 50, 100) para controlar la complejidad y duración del entrenamiento.
    - **n_iter_no_change**: Número de iteraciones sin mejora en el score antes de detener el entrenamiento (None, 5, 10).
    - **max_depth**: Profundidad máxima de los árboles, probando con valores de 5 y 10.
    - **min_samples_split**: Número mínimo de muestras requeridas para dividir un nodo (5, 10, 15).

```
# Matriz de hiperparámetros para el GradientBoosting
param_grid_gb = {
    'n_estimators': [10, 50, 100],
    'n_iter_no_change': [None, 5, 10],
    'max_depth': [5, 10],
    'min_samples_split': [5, 10, 15]
}

# Búsqueda de hiperparámetros para el GradientBoosting
grid_gb = GridSearchCV(GradientBoostingClassifier(random_state=seed), param_grid_gb, cv=5, scoring='accuracy')
grid_gb.fit(X_train, y_train)

print(f'Best Parameters: {grid_gb.best_params_}')
```

2. **Resultados**: Los mejores parámetros encontrados fueron:
    - {'max_depth': 5, 'min_samples_split': 15, 'n_estimators': 50, 'n_iter_no_change': None}
3. **Evaluación del Modelo Final**:
    - **accuracy**: 0.975 en el conjunto de test, lo que indica un excelente rendimiento de clasificación.
    - La matriz de confusión mostró 5 errores de clasificación (2 falsos positivos y 3 falsos negativos), lo que sugiere un buen balance entre las predicciones de ambas clases.

```
# Entrenamiento del GradientBoosting
gb_clf = GradientBoostingClassifier(**grid_gb.best_params_, random_state=seed)
gb_clf.fit(X_train, y_train)

# Evaluación del GradientBoosting
y_pred_gb = gb_clf.predict(X_test)
print(f'Gradient Boosting Classifier Accuracy: {accuracy_score(y_test, y_pred_gb)}')

# Gradient Boosting validación
print("---------------------------------")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_gb))
print("---------------------------------")
print("Classification Report:\n", classification_report(y_test, y_pred_gb))
```

```
Gradient Boosting Classifier Accuracy: 0.9746192893401016
---------------------------------
Confusion Matrix:
 [[108   2]
 [  3  84]]
---------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.98      0.98       110
           1       0.98      0.97      0.97        87

    accuracy                           0.97       197
   macro avg       0.97      0.97      0.97       197
weighted avg       0.97      0.97      0.97       197
```

Resumen:

- El modelo de Gradient Boosting logró un balance entre precisión y capacidad de generalización, con métricas de precision, recall y f1-score cercanas a 1 para ambas clases.

B) Modelo de XGBoost

1. **Selección de Hiperparámetros**: Para el modelo **XGBoost**, se utilizó GridSearchCV con un rango ampliado de hiperparámetros clave:
   - **n_estimators**: Número de árboles (10, 50, 100).
   - **eta**: Tasa de aprendizaje, que controla el tamaño de los pasos dados en cada actualización (0.1, 0.4, 0.7).
   - **gamma**: Umbral para reducir la división de un nodo, que puede ayudar a controlar la complejidad del árbol (0.1, 0.5, 1).
   - **max_depth**: Profundidad máxima de los árboles (5, 10).

```
# Matriz de hiperparámetros para el XGBoost
param_grid_xgb = {
    'n_estimators': [10, 50, 100],
    'eta' : [0.1, 0.4, 0.7],
    'gamma' : [0.1, 0.5, 1],
    'max_depth': [5, 10]
}

grid_xgb = GridSearchCV(XGBClassifier(use_label_encoder=False, eval_metric='logloss'), param_grid_xgb, cv=5, scoring=
grid_xgb.fit(X_train, y_train)

print(f'Best Parameters: {grid_xgb.best_params_}')
```

2. **Resultados**: Los mejores parámetros encontrados fueron:
   - {'eta': 0.1, 'gamma': 1, 'max_depth': 5, 'n_estimators': 50}
3. **Evaluación del Modelo Final**:
   - **accuracy**: 0.975 en el conjunto de test, con un rendimiento muy similar al modelo de Gradient Boosting.
   - La matriz de confusión mostró 5 errores de clasificación (3 falsos positivos y 2 falsos negativos).

```python
# Entrenameinto del XGBoost
xgb_clf = XGBClassifier(**grid_xgb.best_params_)
xgb_clf.fit(X_train, y_train)

# Evaluación de XGBoostClassifier
y_pred_xgb = xgb_clf.predict(X_test)
print(f'XGBoost Classifier Accuracy: {accuracy_score(y_test, y_pred_xgb)}')

# Validación XGBoost
print("---------------------------------")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_xgb))
print("---------------------------------")
print("Classification Report:\n", classification_report(y_test, y_pred_xgb))
```

```
XGBoost Classifier Accuracy: 0.9746192893401016
---------------------------------
Confusion Matrix:
 [[107   3]
 [  2  85]]
---------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.97      0.98       110
           1       0.97      0.98      0.97        87

    accuracy                           0.97       197
   macro avg       0.97      0.97      0.97       197
weighted avg       0.97      0.97      0.97       197
```

Resumen:

- El modelo de XGBoost logró un rendimiento comparable al de Gradient Boosting, con una precisión y capacidad de generalización igualmente sólidas.

Análisis de Resultados y Conclusión

1. **Rendimiento de los Modelos**:
   - **Gradient Boosting**: Con un accuracy de 0.975 y pocos errores de clasificación, muestra un gran rendimiento al optimizar la clasificación de ambas clases.
   - **XGBoost**: Ofrece un rendimiento similar, con un accuracy también de 0.975, y logra una buena capacidad de discriminación en el conjunto de test.
2. **Comparación de Parámetros y Exploración de Valores Extremos**:
   - La exploración de un rango más amplio de hiperparámetros permitió identificar configuraciones óptimas que podrían no haber sido evidentes con una búsqueda más limitada.

- Es importante señalar que en ambas técnicas (Gradient Boosting y XGBoost), los mejores valores para los parámetros se encontraban en el extremo inferior del grid (max_depth=5, eta=0.1). Esto indica que el modelo podría estar beneficiándose de una mayor regularización y simplicidad.

3. **Conclusión Final**:
    - Ambos modelos muestran un accuracy sobresaliente y comparten un rendimiento muy similar. La elección entre Gradient Boosting y XGBoost dependerá de factores como tiempo de entrenamiento y requisitos específicos del problema.
    - La búsqueda paramétrica y la comparación detallada permiten ajustar finamente estos modelos de ensamble para maximizar su capacidad predictiva.

La revisión de hiperparámetros más extrema y detallada garantizó que se encontraran los modelos más precisos posibles para ambos métodos de boosting, asegurando la robustez y eficacia de la clasificación.

7.- Optimización y Evaluación de Modelo SVM con Diferentes Kernels

Objetivo

El objetivo de este ejercicio es ajustar un modelo de **Máquina de Vectores de Soporte (SVM)** utilizando al menos dos kernels diferentes para maximizar la precisión (accuracy). Para esto, se realiza una búsqueda exhaustiva de hiperparámetros con GridSearchCV para encontrar la mejor combinación de parámetros para el modelo SVM.

Selección de Hiperparámetros

1. **Matriz de Hiperparámetros**: Se probó una variedad de combinaciones de hiperparámetros para ajustar el modelo SVM:
    - **C**: Parámetro de regularización que controla la magnitud de las penalizaciones de clasificación incorrecta. Se probaron valores de [0.1, 1, 10, 100].
    - **gamma**: Parámetro de ajuste del kernel, que controla la influencia de un solo ejemplo de entrenamiento. Valores probados: [1, 0.1, 0.01, 0.001].
    - **kernel**: Tipo de kernel utilizado en el modelo. Se exploraron linear, rbf (radial basis function) y poly (polinómico).

```
# Matriz de parametros
param_grid_svc = {
    'C': [0.1, 1, 10, 100],
    'gamma': [1, 0.1, 0.01, 0.001],
    'kernel': ['linear', 'rbf', 'poly']
}

# Seleccionamos los mejores parametros
grid_svc = GridSearchCV(SVC(random_state=seed), param_grid_svc, cv=5, scoring='accuracy')
grid_svc.fit(X_train, y_train)

print(f'Best Parameters: {grid_svc.best_params_}')
```

Resultados: Los mejores parámetros encontrados fueron:

- {'C': 100, 'gamma': 1, 'kernel': 'linear'}

Esto indica que el kernel lineal, con una regularización alta (C=100) y un gamma=1, es la mejor configuración para el modelo SVM en este conjunto de datos.

2. **Entrenamiento y Evaluación del Modelo Final**:
    - Se entrenó el modelo SVM con los mejores parámetros obtenidos de la búsqueda.
    - El modelo fue evaluado en el conjunto de test, y los resultados son:
        - **accuracy**: 0.995, lo que indica una clasificación casi perfecta.
        - **Matriz de Confusión**: Un único error de predicción (1 falso negativo), lo que muestra un balance sólido en ambas clases (obeso y no obeso).

```
# Entrenamos al modelo con los mejores parametros
svc_clf = SVC(**grid_svc.best_params_, random_state=seed)
svc_clf.fit(X_train, y_train)

# Evaluamos el modelo SVC
y_pred_svc = svc_clf.predict(X_test)
print(f'Support Vector Machine Accuracy: {accuracy_score(y_test, y_pred_svc)}')

# Validacion del modelo
print("---------------------------------")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_svc))
print("---------------------------------")
print("Classification Report:\n", classification_report(y_test, y_pred_svc))
```

```
Support Vector Machine Accuracy: 0.9949238578680203
--------------------------------
Confusion Matrix:
 [[110   0]
 [  1  86]]
--------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       110
           1       1.00      0.99      0.99        87

    accuracy                           0.99       197
   macro avg       1.00      0.99      0.99       197
weighted avg       0.99      0.99      0.99       197
```

3. **Evaluación del Modelo SVM y Comparación de Kernels**: La búsqueda de hiperparámetros permitió comparar el rendimiento de diferentes kernels (linear, rbf, poly). Los resultados mostraron que el kernel linear fue el más efectivo en este caso particular, proporcionando la mejor precisión. Es importante considerar que en otros escenarios, kernels más complejos como rbf o poly podrían tener un mejor rendimiento si los datos contienen relaciones no lineales más complejas.

Conclusión

El modelo SVM ajustado con kernel lineal, C=100 y gamma=1, muestra un rendimiento excelente con un accuracy de 0.995 en el conjunto de test. Esto indica que el modelo es capaz de discriminar eficazmente entre individuos obesos y no obesos con una tasa de error mínima.

Este ejercicio demuestra la importancia de probar diferentes configuraciones de hiperparámetros y kernels para SVM, ya que la elección correcta puede tener un gran impacto en la capacidad de clasificación del modelo. Además, un kernel lineal puede ser más efectivo en escenarios donde las relaciones entre características y clases son lineales o se pueden aproximar bien con una separación lineal.

8.- Implementación de un Método de Ensamblado de Bagging con SVM

Objetivo

El objetivo de este ejercicio es aplicar el método de **Bagging** utilizando un clasificador base que no sea un árbol. En este caso, se utilizará el mejor modelo de **SVM** (Soporte Vector Machines) obtenido en el ejercicio anterior como clasificador base para el ensamblado.

Proceso de Construcción del Ensamblado con Bagging

1. **Configuración del Clasificador Base**: Dado que el clasificador SVM con kernel='linear', C=100, y gamma=1 mostró un gran rendimiento en el ejercicio

anterior (con un accuracy de 0.995), se utilizará como estimador base para el método de Bagging.

2. **Ensamblado con BaggingClassifier**: Se utiliza el BaggingClassifier con el siguiente ajuste:
    - **estimator**: El clasificador base es el SVC ajustado previamente (svc_clf).
    - **n_estimators**: Número de clasificaciones a ensamblar, establecido en 10.

```
# Como clasificador base usamos el mejor clasificador SVC obtenido en el ejercicio anterior
bagging_clf = BaggingClassifier(estimator=svc_clf, n_estimators=10, random_state=seed)
bagging_clf.fit(X_train, y_train)
```

3. **Entrenamiento y Evaluación del Modelo de Bagging**:
    - Se entrenó el modelo de Bagging con el conjunto de entrenamiento y se evaluó en el conjunto de test.
    - Los resultados son:
        - **accuracy**: 0.990, lo que indica un rendimiento ligeramente mejorado respecto al clasificador SVM individual.
        - **Matriz de Confusión**: Solo se cometieron 2 errores (2 falsos negativos), lo que muestra un excelente balance y capacidad de generalización.

```
# Evaluamos el modelo de Bagging
y_pred_bagging = bagging_clf.predict(X_test)
print(f'Bagging Classifier Accuracy: {accuracy_score(y_test, y_pred_bagging)}')

# Validación del modelo de Bagging ensamblado
print("---------------------------------")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_bagging))
print("---------------------------------")
print("Classification Report:\n", classification_report(y_test, y_pred_bagging))
```

```
Bagging Classifier Accuracy: 0.9898477157360406
---------------------------------
Confusion Matrix:
 [[110   0]
 [  2  85]]
---------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       110
           1       1.00      0.98      0.99        87

    accuracy                           0.99       197
   macro avg       0.99      0.99      0.99       197
weighted avg       0.99      0.99      0.99       197
```

4. **Análisis del Ensamblado de Bagging**: El uso de Bagging con un clasificador base distinto de árboles (SVM en este caso) permite mejorar ligeramente la

capacidad de generalización y estabilidad del modelo. Esto se debe a que el ensamblado reduce la varianza del modelo al combinar múltiples clasificaciones individuales, lo que contribuye a un mejor rendimiento global.

Conclusión

El modelo de Bagging basado en SVM logra un accuracy de 0.990, con un f1-score alto para ambas clases (0 y 1). Esto demuestra que el ensamblado puede ser una estrategia efectiva para mejorar la precisión de un clasificador base, incluso si este no es un árbol. En este caso, combinar varios modelos SVM en un ensamblado Bagging resultó en un modelo robusto y efectivo para la clasificación de obesidad.

El proceso de Bagging muestra su eficacia al reducir el sobreajuste y mejorar el rendimiento general, lo que es especialmente valioso cuando se utilizan clasificadores base más complejos, como SVM.

9.- Implementación de un Método de Stacking

Objetivo

El objetivo de este ejercicio es aplicar un método de **Stacking** que combine varios modelos previamente entrenados para mejorar el rendimiento de la clasificación. La idea es utilizar un ensamble de clasificadores de entrada con un modelo de ensamblaje final (meta-modelo) para obtener mejores resultados en términos de precisión (accuracy) y Área Bajo la Curva (AUC) de la curva ROC.

Proceso de Construcción del Ensamblado por Stacking

1. **Configuración de los Modelos Base**:
    - Se seleccionaron como clasificadores base varios de los modelos que ya se optimizaron en ejercicios anteriores:
        - **NN**: Modelo de red neuronal (MLPClassifier) con los mejores parámetros.
        - **DTC**: Árbol de decisión (DecisionTreeClassifier).
        - **BAG**: Modelo de Bagging con los mejores parámetros.
        - **RF**: Modelo de Random Forest (RandomForestClassifier).
        - **GBC**: Modelo de Gradient Boosting (GradientBoostingClassifier).
        - **XGB**: Modelo de XGBoost (XGBClassifier).
        - **SVC**: Modelo SVM con kernel lineal.
        - **BAG2**: Modelo de Bagging con SVC como clasificador base.

```python
# Mejores parámetros obtenidos para la Red Neuronal (MLPClassifier)
best_params_nn = {
    'activation': 'tanh',
    'alpha': 0.05,
    'hidden_layer_sizes': (15, 15),
    'learning_rate': 'constant',
    'solver': 'adam'
}

# Mejores parámetros obtenidos para el Árbol de Decisión (DecisionTreeClassifier)
best_params_dtc = {
    'criterion': 'entropy',
    'max_depth': 5,
    'min_samples_split': 2
}

# Mejores parámetros obtenidos para el modelo de Bagging
best_params_bagging = {
    'bootstrap': True,
    'bootstrap_features': True,
    'max_features': 1.0,
    'max_samples': 0.7,
    'n_estimators': 100
}

# Mejores parámetros obtenidos para el Random Forest (RandomForestClassifier)
best_params_rf = {
    'criterion': 'entropy',
    'max_depth': 10,
    'min_samples_leaf': 3,
    'min_samples_split': 5,
    'n_estimators': 50
}

# Mejores parámetros obtenidos para el Gradient Boosting (GradientBoostingClassifier)
best_params_gbc = {
    'max_depth': 5,
    'min_samples_split': 15,
    'n_estimators': 50,
    'n_iter_no_change': None
}

# Mejores parámetros obtenidos para el XGBoost (XGBClassifier)
best_params_xgb = {
    'eta': 0.1,
    'gamma': 1,
    'max_depth': 5,
    'n_estimators': 50
}

# Mejores parámetros obtenidos para el Support Vector Machine (SVC)
best_params_svc = {
    'C': 100,
    'gamma': 1,
    'kernel': 'linear'
}
```

```
# Usamos los modelos de los ejercicios anteriores para comparar
svc =  SVC(**best_params_svc, random_state=seed)
base_models = [
    ('NN', MLPClassifier(**best_params_nn, max_iter=1000, random_state=seed)),
    ('DTC', DecisionTreeClassifier(**best_params_dtc, random_state=seed)),
    ('BAG', BaggingClassifier(**best_params_bagging, random_state=seed)),
    ('RF', RandomForestClassifier(**best_params_rf, random_state=seed)),
    ('GBC', GradientBoostingClassifier(**best_params_gbc, random_state=seed)),
    ('XGB', XGBClassifier(**best_params_xgb, random_state=seed)),
    ('SVC', svc),
    ('BAG2', BaggingClassifier(estimator=svc, random_state=seed))
]
```

2. **Configuración del Ensamblaje con StackingClassifier**:
    - Los clasificadores base fueron ensamblados usando StackingClassifier, con una regresión logística como modelo meta (final_estimator), ya que esta suele funcionar bien para combinar predicciones de diferentes modelos.
    - La validación cruzada (cv=5) se aplicó para entrenar el modelo final de stacking.

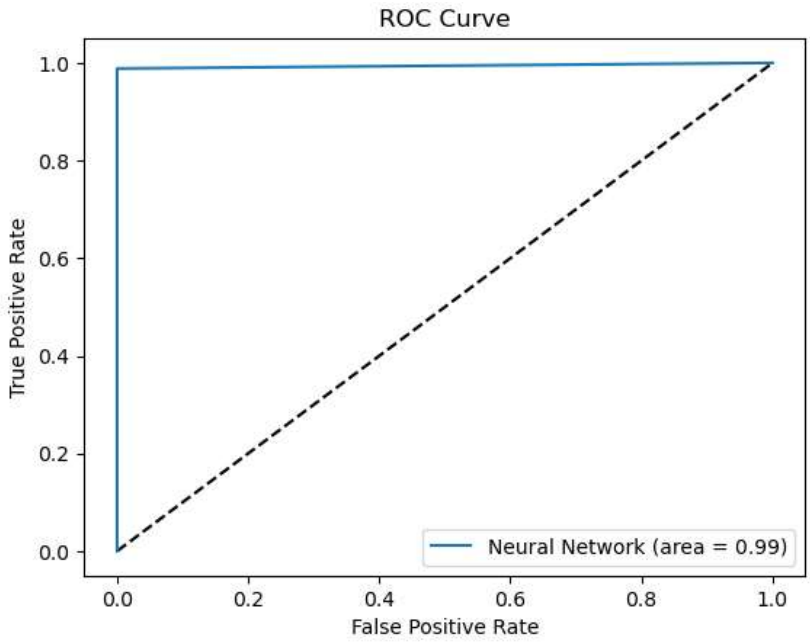
```
# Clasificador
stacking_clf = StackingClassifier(estimators=base_models, final_estimator=LogisticRegression(), cv=5)
stacking_clf.fit(X_train, y_train)
```

3. **Entrenamiento y Evaluación del Modelo Stacking**:
    - El modelo de stacking fue entrenado en el conjunto de entrenamiento y evaluado en el conjunto de test.
    - Los resultados fueron:
        - **accuracy**: 0.995, lo que demuestra un excelente rendimiento del ensamblado, logrando clasificar correctamente casi todas las instancias.
        - **ROC AUC Score**: 0.994, lo que indica que el modelo tiene una gran capacidad de discriminación para ambas clases.
        - **Matriz de Confusión**: Solo se cometió 1 error de clasificación (1 falso negativo), lo que muestra que el modelo tiene un balance sobresaliente en ambas clases.

```python
# Evaluamos el modelo
y_pred = stacking_clf.predict(X_test)
print('Accuracy: ', accuracy_score(y_test, y_pred))
print("ROC AUC Score:", roc_auc_score(y_test, y_pred))

# Validación del modelo
print("---------------------------------")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("---------------------------------")
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy:  0.9949238578680203
ROC AUC Score: 0.9942528735632183
---------------------------------
Confusion Matrix:
 [[110   0]
 [  1  86]]
---------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       110
           1       1.00      0.99      0.99        87

    accuracy                           0.99       197
   macro avg       1.00      0.99      0.99       197
weighted avg       0.99      0.99      0.99       197
```

4. **Curva ROC del Modelo Stacking**:
    - Para visualizar la capacidad de clasificación del modelo, se generó una curva ROC (Receiver Operating Characteristic), que mostró un área bajo la curva (AUC) de 0.994, confirmando el excelente rendimiento del modelo.



Conclusión

El método de stacking, que combina varios modelos con una regresión logística como meta-modelo, alcanzó un accuracy de 0.995 y un AUC de 0.994 en el conjunto de test. Esto demuestra que la combinación de múltiples clasificadores a través de stacking mejora el rendimiento de la clasificación, logrando un modelo más robusto y preciso. La capacidad de ensamblaje del modelo final ayuda a aprovechar las fortalezas individuales de cada clasificador base y a reducir los errores de clasificación.

La curva ROC resultante muestra una gran capacidad de discriminación entre las clases, lo que respalda la efectividad del modelo de stacking como estrategia de ensamble para mejorar la capacidad predictiva.