Machine Learning - Parte 1

Álvaro Vilela Nova

Marzo 2024

1. Introducción

La capacidad de predecir la obesidad a partir de patrones de comportamiento y factores de riesgo podría jugar un papel crucial en la intervención temprana y la implementación de estrategias preventivas personalizadas. En este trabajo, nos proponemos desarrollar un modelo predictivo de obesidad sin depender de medidas directas como la altura y el peso, enfocándonos en los hábitos alimenticios, la condición física y la edad de las personas.

El conjunto de datos que utilizamos incluye atributos relacionados con los hábitos alimenticios:

- Consumo frecuente de alimentos altos en calorías (FAVC).
- Frecuencia de consumo de vegetales (FCVC).
- Número de comidas principales (NCP).
- Consumo de alimentos entre comidas (CAEC).
- Consumo diario de agua (CH20).
- Consumo de alcohol (CALC).

Además, se consideran atributos vinculados con la condición física:

- Monitoreo del consumo de calorías (SCC).
- Frecuencia de actividad física (FAF).
- Tiempo de uso de dispositivos tecnológicos (TUE).
- Medio de transporte utilizado (MTRANS).

Completando la caracterización de los individuos, se incluyen la edad, mientras que hemos decidido excluir deliberadamente la altura y el peso de nuestro análisis. Este enfoque busca explorar la viabilidad de predecir la obesidad únicamente a través de los hábitos de vida y la edad, superando la limitación de tener que depender de la altura y el peso, los cuales permitirían calcular el índice de masa corporal (IMC) de manera directa. Este reto implica no solo un desafío técnico en términos de modelado de datos, sino también una oportunidad para identificar patrones menos evidentes pero significativamente relacionados con la obesidad. Nuestro objetivo es proporcionar insights valiosos que puedan ser utilizados para el desarrollo de intervenciones preventivas más efectivas y personalizadas en la lucha contra la obesidad.

2. Depuración de los datos

Para asegurar la calidad y la integridad de nuestro conjunto de datos, hemos llevado a cabo una serie de pasos de depuración y tratamiento de datos que sientan las bases para nuestro análisis predictivo de la obesidad. A continuación, detallamos el proceso seguido:

1. **Limpieza inicial de datos:** Eliminamos una columna que había sido leída incorrectamente, así como las columnas de altura y peso, para centrarnos en predecir la obesidad basándonos únicamente en hábitos de vida y edad. También renombramos la columna «NObeyesdad» a «Obesidad» para clarificar el objetivo de nuestra variable dependiente.

2. **Selección de muestras:** Optamos por una muestra representativa de 1000 datos, utilizando la semilla 8899 para garantizar la reproducibilidad de nuestra selección a lo largo del trabajo. Este paso es crucial para trabajar con un conjunto de datos manejable y representativo.

3. **Limpieza de la variable objetivo:** Borramos los registros con datos vacíos en nuestra variable objetivo, «Obesidad», para asegurar la integridad de nuestro análisis predictivo.

4. **Revisión y ajuste de tipos de datos:** Comprobamos que los tipos de datos sean los correctos y evaluamos la posibilidad de discretizar variables numéricas para simplificar nuestro modelo y mejorar la interpretación de los resultados.

5. **Discretización de variables numéricas:** Las variables 'CH2O', 'FAF', 'TUE', 'FCVC' y 'NCP' fueron redondeadas y discretizadas. Este paso permite manejar estas variables numéricas como categóricas, facilitando ciertos tipos de análisis estadísticos y predictivos.

6. **Agrupación de categorías:** Unificamos algunas categorías poco representadas en las variables 'MTRANS', 'CALC' y 'CAEC', mejorando así la robustez de nuestro análisis frente a la variabilidad de datos pequeños.

7. **Imputación de datos faltantes:** Empleamos un 'ColumnTransformer' para imputar los datos faltantes, aplicando estrategias diferenciadas para las variables numéricas y categóricas. Para las categóricas, utilizamos un imputado simple que asigna la categoría más común a los datos faltantes. Para las numéricas, recurrimos al método KNN Imputer, que estima los valores faltantes utilizando los vecinos más cercanos.

8. **Establecimiento final de los tipos de datos:** Tras realizar todos los pasos anteriores, ajustamos los tipos de datos de cada variable a lo que deberían ser, estableciendo una base sólida para el análisis posterior.

Garantizamos que nuestro conjunto de datos esté limpio, bien estructurado y listo para el desarrollo de modelos predictivos fiables. La estandarización de la variable numérica se realiza caso por caso a la hora de entrenar los modelos, igual que utilizaremos **OHE** para las variables categóricas. Esto lo haremos utilizando las Pipelines que ofrece **sklearn**.

```
# C digo utilizado para agrupar categor as
df['Obesidad'].replace({'Insufficient_Weight':'0',
             'Normal_Weight':'0',
             'Overweight_Level_I':'0',
             'Overweight_Level_II':'0',
```

```
                'Obesity_Type_I':'1',
                'Obesity_Type_II':'1',
                'Obesity_Type_III':'1'},
                inplace=True)

df['MTRANS'].replace({'Walking':'Active',
                'Bike':'Active',
                'Automobile':'Personal',
                'Motorbike':'Personal'},
                inplace=True)

df['CALC'].replace({
    'Always': 'Yes',
    'Sometimes': 'Yes',
    'Frequently': 'Yes',
    'no': 'No'},
    inplace=True)

df['CAEC'].replace({
    'Always': 'Always/Frequently',
    'Frequently': 'Always/Frequently'},
    inplace=True)

# C digo utilizado para imputar datos faltantes
# Define the transformation for categorical columns
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent'))
])

# Define the transformation for numerical columns
numerical_transformer = Pipeline(steps=[
    ('imputer', KNNImputer())
])

column_transformer = ColumnTransformer(
    transformers=[
        ('cat', categorical_transformer, cat_cols),
        ('num', numerical_transformer, num_cols)
    ], remainder='passthrough'
)

remainder_cols = [col for col in df.columns if col not in cat_cols +
    ↪ num_cols]
all_transformed_cols = cat_cols + num_cols
all_transformed_cols += remainder_cols

df = pd.DataFrame(column_transformer.fit_transform(df), columns=
    ↪ all_transformed_cols, index=df.index)

# Conversi n de categor as num ricas a categ ricas
```

```
df = df.round({"FCVC":0,
               "NCP":0,
               "CH2O":0,
               "FAF":0,
               "TUE":0})

df['FCVC'] = df['FCVC'].astype(str)
df['NCP'] = df['NCP'].astype(str)
df['CH2O'] = df['CH2O'].astype(str)
df['FAF'] = df['FAF'].astype(str)
df['TUE'] = df['TUE'].astype(str)
```

3. Análisis preliminar de las variables

Graficamos en el caso de la variable numérica la distribución y el gráfico de cajas; en el caso de las variables categóricas, simplemente graficamos el número de observaciones de cada categoría dividido en los casos de obesidad y no-obesidad. Basándonos en los gráficos obtenidos, se pueden hacer varias observaciones relacionadas con la incidencia de la obesidad en relación con distintos factores demográficos y de hábitos de vida:

1. **Distribución de la Edad:**

   La distribución de la edad muestra dos picos en el conteo de obesidad, indicando edades específicas donde la obesidad es más prevalente.

   En el gráfico de cajas, parece haber una mediana de edad más alta en el grupo con obesidad (1) comparado con el grupo sin obesidad (0), lo que sugiere que la obesidad podría ser más común en edades avanzadas dentro de este conjunto de datos.

2. **Hábitos Alimenticios (CAEC, CALC, CH2O, FAVC, FCVC):**

   El consumo de alimentos entre comidas (CAEC) muestra que aquellos que comen .ª veces.º "siempre/frecuentemente.enentre comidas tienen más incidencia de obesidad que aquellos que no lo hacen.

   El consumo de alcohol (CALC) presenta una distribución más uniforme entre los que consumen y los que no, aunque hay una mayor incidencia de obesidad en las personas que consumen alcohol.

   El consumo de agua (CH2O) revela que las personas que consumen 2 litros de agua al día tienen una mayor prevalencia de no obesidad en comparación con otros niveles de consumo.

   Los individuos que consumen frecuentemente alimentos altos en calorías (FAVC) muestran una alta prevalencia de obesidad.

   Un consumo más frecuente de vegetales (FCVC) está asociado con una menor incidencia de obesidad.

3. **Frecuencia de Actividad Física (FAF):**

   Los datos muestran que la falta de actividad física (FAF = 0) está asociada con una mayor incidencia de obesidad.

A medida que aumenta la frecuencia de actividad física, parece disminuir la incidencia de obesidad.

4. **Historia Familiar de Sobrepeso:**

   Una historia familiar de sobrepeso está fuertemente asociada con una mayor incidencia de obesidad en los individuos.

5. **Género:**

   La incidencia de obesidad parece ser ligeramente más alta en el género masculino en comparación con el femenino en este conjunto de datos.

6. **Método de Transporte (MTRANS):**

   Aquellos que usan transporte personal tienen una incidencia más alta de obesidad en comparación con aquellos que usan transporte público o medios activos de transporte.

   Número de Comidas Principales (NCP): Los individuos que tienen tres comidas principales al día presentan una mayor tendencia a no tener obesidad en comparación con aquellos que tienen solo una o dos comidas principales, mientras que no hay una diferencia clara entre aquellos con cuatro comidas principales.

7. **Monitoreo del Consumo de Calorías (SCC):**

   La mayoría de las personas que no monitorean su consumo de calorías tienden a no tener obesidad, mientras que hay una pequeña cantidad de individuos que sí monitorean su consumo de calorías y presentan obesidad. Esto podría indicar que quienes monitorean las calorías podrían hacerlo como parte de un esfuerzo por controlar su peso.

8. **Hábito de Fumar (SMOKE):**

   Hay una gran cantidad de individuos que no fuman en ambos grupos de obesidad. Sin embargo, hay una proporción relativamente mayor de personas con obesidad entre los fumadores, aunque el número total de fumadores es pequeño en comparación con los no fumadores.

9. **Tiempo de Uso de Dispositivos Tecnológicos (TUE):**

   La mayoría de las personas pasan 0 o 1 hora usando dispositivos tecnológicos diariamente. Parece que en general no hay una gran diferencia en las distintas categorías en tendencia a la obesidad. Aunque parece que las persona que pasan cerca de 2 horas al día tienen menos tendencia.

Vamos a buscar posibles correlaciones entre nuestros datos y nuestra variable objetivo utilizando un gráfico de barras con la métrica V de Cramer.

Vamos a eliminar las variables SMOKE y TUE pues parecen no ser muy buenas discriminando nuestros datos, además están excesivamente desequilibradas. El resto de variables vamos a mantenerlas bien porque son buenas discriminando o porque puede que nuestros modelos sean capaces de encontrar relaciones entre ellas que no son aparentes a simple vista.

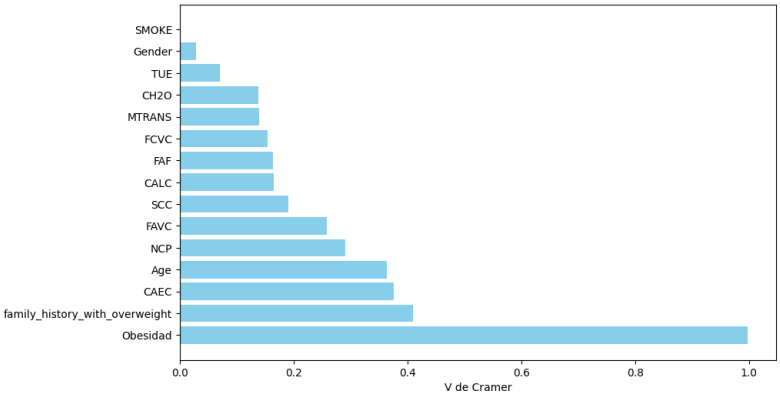

Figura 1: V de Cramer para cada variable

4. Regresión logística y red neuronal a partir de ella

Por lo general vamos a dividir siempre los datos en set de entrenamiento y de test con una proporción 80/20. Vamos a probar varias regresiones logísticas, aplicando distintos métodos para encontrar qué variables es mejor usar.

4.1. Regresiones logísticas

Para que las regresiones logísticas den los mejores datos que sean posibles hacemos un One Hot Encoding sobre las variables categóricas y estandarizamos los datos numéricos.

Probamos con una regresión logística incluyendo: todas las variables de nuestros datos; una regresión logística utilizando los datos con tan solo las 5 variables más relevantes (según el estadístico V de Cramer), [**'family_history_with_overweight','CAEC','Age','NCP','FAVC'**]; selección de variables clásica con el método backward; selección de variables clásica con el método forward. La métrica utilizada durante la selección de ambos modelos es accuracy.

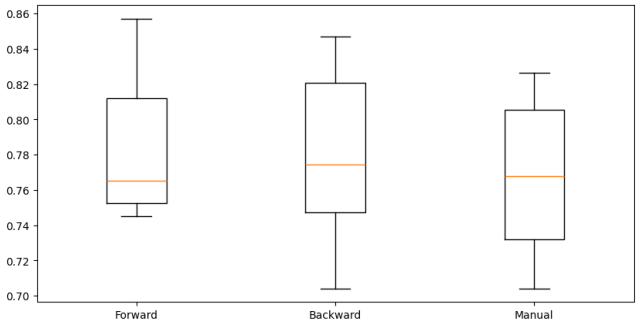

Figura 2: Gráfico de cajas de validación cruzada entre modelos logísticos

Tras entrenar todos los modelos, los comparamos utilizando el método de validación cruzada.

No comparamos el modelo con todas las variables puesto que el objetivo es seleccionar uno con menos variables para después entrenar una red neuronal.

El modelo de regresión logística con selección clásica de variables forward si bien parece tener una precisión mediana ligeramente menor que el modelo de regresión logística obtenido utilizando la selección clásica backward; vamos a sacrificar esa pequeña diferencia en accuracy a cambio de tener un modelo ligeramente más consistente, que no suela tener bajones de rendimiento grandes.

```
# Assuming cat_cols and num_cols are defined, as well as X, y, and seed

# Step 1: Transform the data using the ColumnTransformer
X_transformed = column_transformer.fit_transform(X)

# Step 2: Get transformed feature names
transformed_feature_names = column_transformer.get_feature_names_out()

# Step 3: Identify selected features
# Since SequentialFeatureSelector does not have get_support method
    ↪ directly accessible through the pipeline,
# we fit the ColumnTransformer and SequentialFeatureSelector separately
    ↪ .
# Note: This step assumes you have fit the model_fw to your data as
    ↪ shown above.

# Extract the SequentialFeatureSelector from the pipeline
feature_selector = model_fw.named_steps['feature_selection']

# Fit the feature selector on the transformed data to update the
    ↪ get_support() method
feature_selector.fit(X_transformed, y)

# Use get_support() to identify selected features
selected = feature_selector.get_support()

# Step 4: Filter for selected columns
selected_columns = transformed_feature_names[selected]

# Convert the transformed data (a numpy array) into a DataFrame
X_transformed_df = pd.DataFrame(X_transformed, columns=
    ↪ transformed_feature_names)

# Filter the DataFrame to keep only the selected features
selected_features_df = X_transformed_df[selected_columns]

# Now, selected_features_df is the DataFrame with the columns selected
    ↪ by the model
selected_features_df
```

Este código nos da el dataframe con tan solo las variables utilizadas por el modelo Forward. Las variables utilizadas son: **['cat__family_history_with_overweight_yes', 'cat__FAVC_yes', 'cat__CAEC_Sometimes', 'cat__CAEC_no', 'cat__SCC_yes', 'cat__MTRANS_Personal',**

'cat__CH2O_2.0', 'cat__FAF_3.0', 'cat__NCP_3.0', 'cat__NCP_4.0']

4.1.1. Red neuronal

Utilizando el dataframe obtenido en la sección anterior vamos a entrenar una red neuronal con unos parámetros que he creído convenientes y otra red neuronal con unos parámetros escogidos utilizando un grid search.

Posiblemente los parámetros más importantes a tener en cuenta a la hora de definir y entrenar una red neuronal son el número de nodos en las capas ocultas y el índice de aprendizaje (determinado por un parámetro alpha). Hay diferentes heurísticas distintas que nos ayudan a escoger el número de nodos en nuestras redes neuronales, por ejemplo, $h = \frac{k}{2} + o$, donde k es el número de nodos de la capa de entrada y o el número de nodos de la capa de salida, en nuestro caso el resultado sería 7.

Como no hay ninguna heurística ampliamente aceptada como la mejor vamos a tomar una variedad bastante amplia de valores que se adapten a algunos de estos resultados. El parámetro alpha que determina cómo cambiará el *learning rate* también recibirá una cantidad amplia de valores para capturar más redes neuronales distintas.

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

# Red con parametros manuales
red1 = MLPClassifier(random_state=seed, hidden_layer_sizes=(7),
    ↪ activation='tanh',
                     alpha=0.001,solver='adam',max_iter=1000)
```

Más tarde compararemos este modelo, el obtenido con el grid search y el modelo obtenido en el apartado 2 utilizando validación cruzada.

```
red = MLPClassifier(random_state=seed)
#definimos los parametros que queremos tunear
params = {
    'max_iter': [600, 1000],
    'hidden_layer_sizes': [4,5,6,7,8],
    'activation': ['tanh','relu'],
    'alpha': [0.001,0.0001,0.005,0.05,0.002,0.0005]
}
scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', '
    ↪ f1_macro']
# cv = crossvalidation con n folds con todas las combinaciones de
    ↪ parametros
grid_search = GridSearchCV(estimator=red,
                           param_grid=params,
                           cv=8, scoring = scoring_metrics, refit='
                               ↪ accuracy')
```

De entre los 5 mejores modelos según la precisión media que hemos encontrado tenemos el siguiente gráfico de su validación cruzada.

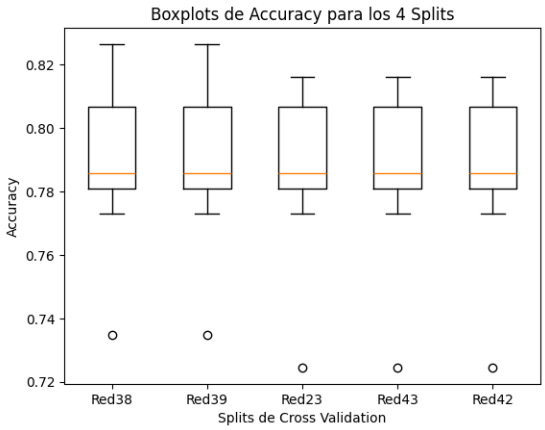

Figura 3: Gráfico de cajas de validación cruzada entre redes neuronales con grid search

De todas estas redes neuronales vamos a escoger la red 38 que guardamos sus parámetros:

```
red11 = MLPClassifier(random_state=seed, hidden_layer_sizes=(8),
    ↪ activation='tanh',
                      alpha=0.05,solver='adam',max_iter=600)
```

5. Selección de variables select k best, k=4. Mejor modelo de red neuronal en términos de AUC

Para utilizar el SelectKBest nuestros datos deben ser todos no-negativos. Así que antes de estandarizar nuestra variable numérica Age vamos a realizar la selección de variables.

```
# Define the transformation for categorical columns
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(drop='first'))
])

# # Define the transformation for numerical columns
numerical_transformer = Pipeline(steps=[
    ('imputer', KNNImputer()),
])

column_transformer = ColumnTransformer(
    transformers=[
        ('cat', categorical_transformer, cat_cols),
        ('num', numerical_transformer, num_cols)
    ]
)
```

```
X = transform_dataframe(X, column_transformer)

# Step 1: Fit SelectKBest to identify selected features
selector = SelectKBest(chi2, k=4).fit(X, y)

# Step 2: Get the boolean mask of selected features
selected_mask = selector.get_support()

# Alternatively, get the indices of selected features
# selected_indices = selector.get_support(indices=True)

# Step 3: Create a new DataFrame with the selected features
X_new = X.loc[:, selected_mask]
```

El nuevo DataFrame tiene como columnas las variables **family_history_with_overweight_yes, SCC_yes, NCP_4.0, Age.** Una vez han sido elegidas estas variables, estandarizamos la variable numérica.

```
red = MLPClassifier(random_state=seed)
#definimos los par metros que queremos tunear
params = {
    'max_iter': [600, 1000],
    'hidden_layer_sizes': [7,8,9],
    'activation': ['tanh','relu'],
    'alpha': [0.001,0.0001,0.005,0.05,0.002,0.0005]
}
scoring_metrics = ['roc_auc', 'precision_macro', 'recall_macro', '
    ↪ f1_macro']
# cv = crossvalidation con n folds con todas las combinaciones de
    ↪ par metros
grid_search = GridSearchCV(estimator=red,
                           param_grid=params,
                           cv=10, scoring = scoring_metrics, refit='
                               ↪ roc_auc')

#ajusta en entrenamiento con todas las combinaciones
grid_search.fit(X_train, y_train)
```

En este caso para seleccionar los parámetros a probar he utilizado la heurística que aparece en los apuntes del módulo.

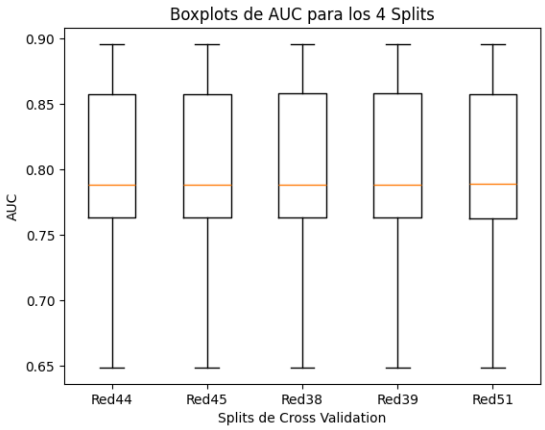

Figura 4: Gráfico de cajas de validación cruzada entre redes neuronales con grid search utilizando AUC

Todos los modelos candidatos tienen los mismos resultados, así que escogeremos uno de ellos, por ejemplo, el 44.

```
red2 = MLPClassifier(random_state=seed, hidden_layer_sizes=(8),
    ↪ activation='relu',
                     alpha=0.0001,max_iter=600)
```

6. Comparación de los modelos candidatos

Vamos a comparar los modelos que hemos obtenido hasta ahora para determinar cuál de ellos es el más adecuado. Debemos decidir una métrica para compararlos, entre sí, me parecería apropiado utilizar accuracy para compararlos. Puede que esto afecte en nuestra comparación a la red de este último apartado, pues ha sido escogida con el objetivo de maximizar el AUC y no el accuracy como en los otros casos. La métrica de accuracy parece ser la más apropiada de todas maneras.

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

logreg = LogisticRegression(random_state=seed)
logreg.fit(X_train, y_train)
print(logreg.score(X_test, y_test))

red1 = MLPClassifier(random_state=seed, hidden_layer_sizes=(7),
    ↪ activation='tanh',
                     alpha=0.001,solver='adam',max_iter=600)
red1.fit(X_train, y_train)
print(red1.score(X_test, y_test))
```

```
red11 = MLPClassifier(random_state=seed, hidden_layer_sizes=(8),
    ↪ activation='tanh',
                      alpha=0.05,solver='adam',max_iter=600)
red11.fit(X_train,y_train)
print(red11.score(X_test, y_test))

X_train, X_test, y_train, y_test = train_test_split(X_new, y, test_size
    ↪ =0.2, random_state=seed)

red2 = MLPClassifier(random_state=seed, hidden_layer_sizes=(8),
    ↪ activation='relu',
                      alpha=0.0001,max_iter=600)
red2.fit(X_train, y_train)
print(red2.score(X_test, y_test))
```

En general los resultados para los tres primeros modelos son buenos, la regresión logística y la primera red neuronal tienen resultados iguales de 0.83, la red obtenida con grid search (red11) apróximadamente 0.83 y la red del último apartado aprox 0.71.

```
# Cross-Validation
models = [('Logistic', logreg),
          ('Red 1', red1),
          ('Red 1 CV', red11)]
results = []
names = []

for name, model in models:
    cv_technique = KFold(n_splits=10, shuffle=True, random_state=seed)
    result = cross_val_score(model, selected_features_df, y, cv=
        ↪ cv_technique)

    names.append(name)
    results.append(result)

# El modelo con selecci n de variables manual tiene un X distinto
cv_technique = KFold(n_splits=10, shuffle=True, random_state=seed)
result = cross_val_score(red2, X_new, y, cv=cv_technique)
names.append('Red 2')
results.append(result)

plt.figure(figsize=(10,5))
plt.boxplot(results)
plt.xticks(range(1, len(names)+1), names)
plt.show()
```

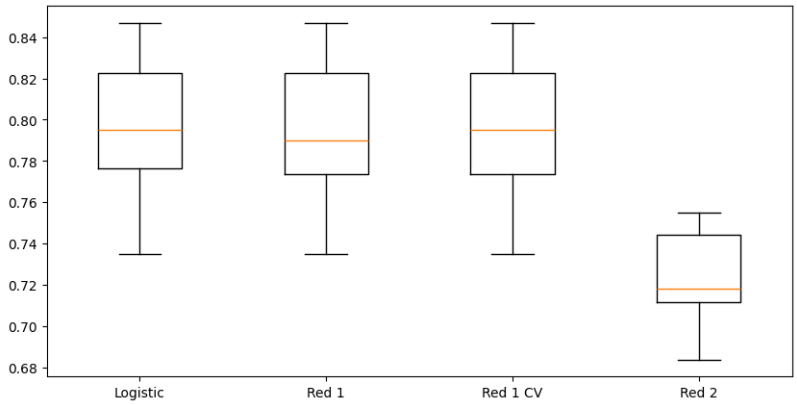

Figura 5: Gráfico de cajas de validación cruzada entre modelos hasta ahora

Observamos que la red obtenida en el primer apartado utilizando el grid search para ser el modelo más eficiente y efectivo aunque la regresión logística ganadora y la primera red neuronal no se quedan muy atrás. La red neuronal obtenida con SelectKBest claramente tiene menor rendimiento, esto puede darse debido a que hemos seleccionado tan solo $k = 4$ variables y que además ha sido seleccionada en términos de AUC.

7. Árbol de decisión

El árbol de decisiones lo evaluaremos utilizando una pipeline que llevamos utilizando hasta ahora y **DecisionTreeClassifier()**.

```
X = df.loc[:, df.columns != 'Obesidad']
y = df['Obesidad']

# Crear un conjunto de entrenamiento y uno de prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    ↪ =0.2, random_state=seed)

column_transformer = ColumnTransformer(
    transformers=[
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(drop='first'))
        ]), cat_cols),
        ('num', StandardScaler(), num_cols)
    ],
    remainder='passthrough'  # This specifies what to do with columns
        ↪ not explicitly selected
)

model = Pipeline([
```

```
    ('Transformer', column_transformer),
    ('Arbol', DecisionTreeClassifier(random_state=seed))
])
```

Ahora vamos a definir los árboles utilizando una búsqueda paramétrica que utilizará como métricas de evaluación: la tasa de aciertos, la precisión, la exhaustividad y la medida F o f1 score.

```
param_grid = {
    'Arbol__max_depth': [2, 4, 6, 8],
    'Arbol__min_samples_split': [10, 20, 30],
    'Arbol__criterion': ['gini', 'entropy']
}

from sklearn.metrics import make_scorer, precision_score, recall_score,
    ↪  f1_score

# Assuming your positive class label is '1' after converting y to
    ↪ integers
# If it remains as strings, use '1'

scoring_metrics = {
    'accuracy': 'accuracy',
    'precision': make_scorer(precision_score, pos_label='1'),  # Use
        ↪ string for pos_label
    'recall': make_scorer(recall_score, pos_label='1'),        # Use
        ↪ string for pos_label
    'f1': make_scorer(f1_score, pos_label='1')                 # Use
        ↪ string for pos_label
}

# Create the GridSearchCV object
grid_search = GridSearchCV(model, param_grid, cv=5, scoring=
    ↪ scoring_metrics, refit=False)

# Fit the GridSearchCV object
grid_search.fit(X_train, y_train)

# To access the results for each metric
results = grid_search.cv_results_
for metric in scoring_metrics:
    print(f"Best parameters for {metric}:")
    best_index = results[f'rank_test_{metric}'].argmin()
    print(results['params'][best_index])
    print(f"Best score for {metric}: {results[f'mean_test_{metric}'][
        ↪ best_index]}")
```

```
Best parameters for accuracy:
{'Arbol__criterion': 'gini', 'Arbol__max_depth': 8, 'Arbol__min_samples_split': 10}
Best score for accuracy: 0.8102564102564103
Best parameters for precision:
{'Arbol__criterion': 'gini', 'Arbol__max_depth': 8, 'Arbol__min_samples_split': 10}
Best score for precision: 0.8075714981136667
Best parameters for recall:
{'Arbol__criterion': 'entropy', 'Arbol__max_depth': 2, 'Arbol__min_samples_split': 10}
Best score for recall: 0.9728619029988893
Best parameters for f1:
{'Arbol__criterion': 'gini', 'Arbol__max_depth': 8, 'Arbol__min_samples_split': 20}
Best score for f1: 0.7983905434057165
```

Figura 6: Mejores modelos para cada medida de bondad de clasificación

Comprobamos que el mejor árbol en dos de los casos (para la accuracy y la precisión) es el que utiliza como criterio el coeficiente de Gini, tiene una profundidad máxima de 8 y un mínimo de observaciones para hacer una separación de 10. En el caso del f1 tenemos un árbol con criterios muy similares. Por todo esto, escogeremos como parámetros de nuestro árbol el primer conjunto.

7.1. Reglas del árbol en formato texto

```
model = Pipeline([
    ('Transformer', column_transformer),
    ('Arbol', DecisionTreeClassifier(criterion='gini', max_depth=8,
        ↪ min_samples_split=10, random_state=seed))
])

model.fit(X_train, y_train)

# Conocer los niveles de la variable a predecir
print(model.classes_)
# Conocer el nombre de las variables predictoras
print(model.feature_names_in_)
# Obtener informaci n detallada de cada nodo y las reglas de decisi n
decision_tree_model = model.named_steps['Arbol']
# Now, use export_text with the extracted decision tree
tree_rules = export_text(decision_tree_model, feature_names=list(
    ↪ column_transformer.get_feature_names_out()), show_weights=True)
print(tree_rules)
```

```
|--- cat__family_history_with_overweight_yes <= 0.50
|   |--- cat__Gender_Male <= 0.50
|   |   |--- cat__NCP_4.0 <= 0.50
|   |   |   |--- weights: [68.00, 0.00] class: 0
|   |   |--- cat__NCP_4.0 >  0.50
|   |   |   |--- cat__CALC_Yes <= 0.50
|   |   |   |   |--- weights: [1.00, 1.00] class: 0
```

```
|   |   |   |--- cat__CALC_Yes >  0.50
|   |   |   |   |--- weights: [9.00, 0.00] class: 0
|   |--- cat__Gender_Male >  0.50
|   |   |--- cat__NCP_3.0 <= 0.50
|   |   |   |--- weights: [23.00, 0.00] class: 0
|   |   |--- cat__NCP_3.0 >  0.50
|   |   |   |--- cat__CH2O_3.0 <= 0.50
|   |   |   |   |--- num__Age <= -0.92
|   |   |   |   |   |--- weights: [7.00, 2.00] class: 0
|   |   |   |   |--- num__Age >  -0.92
|   |   |   |   |   |--- weights: [20.00, 0.00] class: 0
|   |   |   |--- cat__CH2O_3.0 >  0.50
|   |   |   |   |--- weights: [5.00, 2.00] class: 0
|--- cat__family_history_with_overweight_yes >  0.50
|   |--- cat__CAEC_Sometimes <= 0.50
|   |   |--- num__Age <= -0.23
|   |   |   |--- cat__CH2O_2.0 <= 0.50
|   |   |   |   |--- cat__CALC_Yes <= 0.50
|   |   |   |   |   |--- weights: [9.00, 0.00] class: 0
|   |   |   |   |--- cat__CALC_Yes >  0.50
|   |   |   |   |   |--- weights: [6.00, 2.00] class: 0
|   |   |   |--- cat__CH2O_2.0 >  0.50
|   |   |   |   |--- weights: [28.00, 0.00] class: 0
|   |   |--- num__Age >  -0.23
|   |   |   |--- cat__FCVC_3.0 <= 0.50
|   |   |   |   |--- weights: [6.00, 3.00] class: 0
|   |   |   |--- cat__FCVC_3.0 >  0.50
|   |   |   |   |--- weights: [5.00, 0.00] class: 0
|   |--- cat__CAEC_Sometimes >  0.50
|   |   |--- num__Age <= -0.53
|   |   |   |--- cat__Gender_Male <= 0.50
|   |   |   |   |--- cat__NCP_3.0 <= 0.50
|   |   |   |   |   |--- weights: [18.00, 0.00] class: 0
|   |   |   |   |--- cat__NCP_3.0 >  0.50
|   |   |   |   |   |--- cat__FAVC_yes <= 0.50
|   |   |   |   |   |   |--- weights: [7.00, 0.00] class: 0
|   |   |   |   |   |--- cat__FAVC_yes >  0.50
|   |   |   |   |   |   |--- num__Age <= -0.54
|   |   |   |   |   |   |   |--- cat__SCC_yes <= 0.50
|   |   |   |   |   |   |   |   |--- weights: [4.00, 45.00] class: 1
|   |   |   |   |   |   |   |--- cat__SCC_yes >  0.50
|   |   |   |   |   |   |   |   |--- weights: [1.00, 0.00] class: 0
|   |   |   |   |   |   |--- num__Age >  -0.54
|   |   |   |   |   |   |   |--- weights: [4.00, 0.00] class: 0
|   |   |   |--- cat__Gender_Male >  0.50
|   |   |   |   |--- cat__FAF_2.0 <= 0.50
|   |   |   |   |   |--- cat__NCP_3.0 <= 0.50
|   |   |   |   |   |   |--- num__Age <= -0.89
|   |   |   |   |   |   |   |--- weights: [0.00, 7.00] class: 1
|   |   |   |   |   |--- num__Age >  -0.89
```

```
|   |   |   |   |   |   |   |--- num__Age <= -0.69
|   |   |   |   |   |   |   |   |--- weights: [6.00, 1.00] class: 0
|   |   |   |   |   |   |   |--- num__Age >  -0.69
|   |   |   |   |   |   |   |   |--- weights: [1.00, 4.00] class: 1
|   |   |   |   |   |--- cat__NCP_3.0 >  0.50
|   |   |   |   |   |   |--- num__Age <= -0.62
|   |   |   |   |   |   |   |--- num__Age <= -1.07
|   |   |   |   |   |   |   |   |--- weights: [2.00, 2.00] class: 0
|   |   |   |   |   |   |   |--- num__Age >  -1.07
|   |   |   |   |   |   |   |   |--- weights: [23.00, 1.00] class: 0
|   |   |   |   |   |   |--- num__Age >  -0.62
|   |   |   |   |   |   |   |--- num__Age <= -0.53
|   |   |   |   |   |   |   |   |--- weights: [0.00, 8.00] class: 1
|   |   |   |   |   |   |   |--- num__Age >  -0.53
|   |   |   |   |   |   |   |   |--- weights: [4.00, 0.00] class: 0
|   |   |   |   |--- cat__FAF_2.0 >  0.50
|   |   |   |   |   |--- num__Age <= -0.62
|   |   |   |   |   |   |--- weights: [22.00, 0.00] class: 0
|   |   |   |   |   |--- num__Age >  -0.62
|   |   |   |   |   |   |--- weights: [4.00, 1.00] class: 0
|   |   |--- num__Age >  -0.53
|   |   |   |--- cat__FCVC_3.0 <= 0.50
|   |   |   |   |--- cat__NCP_3.0 <= 0.50
|   |   |   |   |   |--- num__Age <= -0.32
|   |   |   |   |   |   |--- weights: [13.00, 0.00] class: 0
|   |   |   |   |   |--- num__Age >  -0.32
|   |   |   |   |   |   |--- num__Age <= -0.24
|   |   |   |   |   |   |   |--- weights: [0.00, 5.00] class: 1
|   |   |   |   |   |   |--- num__Age >  -0.24
|   |   |   |   |   |   |   |--- num__Age <= 2.02
|   |   |   |   |   |   |   |   |--- weights: [27.00, 14.00] class: 0
|   |   |   |   |   |   |   |--- num__Age >  2.02
|   |   |   |   |   |   |   |   |--- weights: [2.00, 8.00] class: 1
|   |   |   |   |--- cat__NCP_3.0 >  0.50
|   |   |   |   |   |--- num__Age <= 0.04
|   |   |   |   |   |   |--- cat__CH2O_3.0 <= 0.50
|   |   |   |   |   |   |   |--- cat__FAF_2.0 <= 0.50
|   |   |   |   |   |   |   |   |--- weights: [27.00, 12.00] class: 0
|   |   |   |   |   |   |   |--- cat__FAF_2.0 >  0.50
|   |   |   |   |   |   |   |   |--- weights: [1.00, 6.00] class: 1
|   |   |   |   |   |   |--- cat__CH2O_3.0 >  0.50
|   |   |   |   |   |   |   |--- cat__Gender_Male <= 0.50
|   |   |   |   |   |   |   |   |--- weights: [1.00, 0.00] class: 0
|   |   |   |   |   |   |   |--- cat__Gender_Male >  0.50
|   |   |   |   |   |   |   |   |--- weights: [2.00, 17.00] class: 1
|   |   |   |   |   |--- num__Age >  0.04
|   |   |   |   |   |   |--- cat__CALC_Yes <= 0.50
|   |   |   |   |   |   |   |--- cat__FAVC_yes <= 0.50
|   |   |   |   |   |   |   |   |--- weights: [0.00, 2.00] class: 1
|   |   |   |   |   |   |   |--- cat__FAVC_yes >  0.50
```

```
|   |   |   |   |   |   |   |   |--- weights: [9.00, 3.00] class: 0
|   |   |   |   |   |   |   |--- cat__CALC_Yes >  0.50
|   |   |   |   |   |   |   |   |--- num__Age <= 3.87
|   |   |   |   |   |   |   |   |   |--- weights: [11.00, 65.00] class: 1
|   |   |   |   |   |   |   |   |--- num__Age >  3.87
|   |   |   |   |   |   |   |   |   |--- weights: [1.00, 0.00] class: 0
|   |   |   |--- cat__FCVC_3.0 >  0.50
|   |   |   |   |--- cat__MTRANS_Personal <= 0.50
|   |   |   |   |   |--- cat__NCP_4.0 <= 0.50
|   |   |   |   |   |   |--- cat__FAVC_yes <= 0.50
|   |   |   |   |   |   |   |--- weights: [2.00, 1.00] class: 0
|   |   |   |   |   |   |--- cat__FAVC_yes >  0.50
|   |   |   |   |   |   |   |--- cat__Gender_Male <= 0.50
|   |   |   |   |   |   |   |   |--- weights: [2.00, 101.00] class: 1
|   |   |   |   |   |   |   |--- cat__Gender_Male >  0.50
|   |   |   |   |   |   |   |   |--- weights: [6.00, 36.00] class: 1
|   |   |   |   |   |--- cat__NCP_4.0 >  0.50
|   |   |   |   |   |   |--- weights: [2.00, 0.00] class: 0
|   |   |   |   |--- cat__MTRANS_Personal >  0.50
|   |   |   |   |   |--- num__Age <= 0.74
|   |   |   |   |   |   |--- weights: [7.00, 1.00] class: 0
|   |   |   |   |   |--- num__Age >  0.74
|   |   |   |   |   |   |--- cat__CALC_Yes <= 0.50
|   |   |   |   |   |   |   |--- num__Age <= 2.40
|   |   |   |   |   |   |   |   |--- weights: [7.00, 0.00] class: 0
|   |   |   |   |   |   |   |--- num__Age >  2.40
|   |   |   |   |   |   |   |   |--- weights: [1.00, 2.00] class: 1
|   |   |   |   |   |   |--- cat__CALC_Yes >  0.50
|   |   |   |   |   |   |   |--- num__Age <= 2.07
|   |   |   |   |   |   |   |   |--- weights: [2.00, 15.00] class: 1
|   |   |   |   |   |   |   |--- num__Age >  2.07
|   |   |   |   |   |   |   |   |--- weights: [5.00, 2.00] class: 0
```

7.2. Representar gráficamente el árbol ganador

La representación gráfica del árbol es una imagen muy grande, puede verse haciendo click aqui.

```
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (20,20), dpi
    ↪ =300)
plot_tree(decision_tree_model, filled=True)
```

7.3. Graficar la importancia de sus variables

Podemos obtener la importancia de las variables en nuestro árbol utilizando el parámetro .feature_importances_ de nuestro modelo.

```
# Ordenar el DataFrame por importancia en orden descendente
df_importancia = pd.DataFrame({'Variable': list(column_transformer.
    ↪ get_feature_names_out()), 'Importancia': decision_tree_model.
    ↪ feature_importances_}).sort_values(by='Importancia', ascending=
    ↪ False)

# Crear un grafico de barras
plt.bar(df_importancia['Variable'], df_importancia['Importancia'],
    ↪ color='skyblue')
plt.xlabel('Variable')
plt.ylabel('Importancia')
plt.title('Importancia de las caracteristicas')
plt.xticks(rotation=45, ha='right')  # Rotar los nombres en el eje x
    ↪ para mayor legibilidad
plt.tight_layout()

# Mostrar el grafico
plt.show()
```

Comprobamos que las variables más importantes de nuestro árbol de decisión son la edad y la historia familiar de sobrepeso. Parece cercano a lo observado en la gráfica del estadístico V de Cramer.

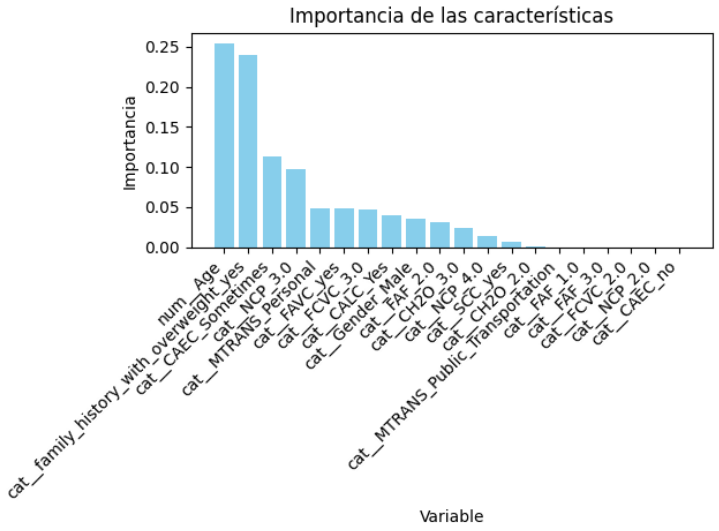

Figura 7: Gráfico de importancia de las variables en nuestro árbol de decisiones

8. Búsqueda paramétrica para lograr el mejor modelo de Bagging y Random Forest según Accuracy

Para calcular estos modelos utilizamos pipelines similares a los modelos anteriores, permitiremos a sklearn escoger el mejor modelo y que nos muestre los mejores parámetros en cada caso. Los parámetros pueden verse en el código dentro de cada grid search.

```
X = df.loc[:, df.columns != 'Obesidad']
y = df['Obesidad']

# Crear un conjunto de entrenamiento y uno de prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    ↪ =0.2, random_state=seed)

model = Pipeline([
    ("Transformer", column_transformer),
    ("Bagging", BaggingClassifier(random_state=seed))
])

# Parameters grid
param_grid_bagging = {
    'Bagging__n_estimators': [10, 50, 100],
    'Bagging__max_samples': [0.5, 0.7, 1.0]
}

# GridSearchCV
grid_search_bagging = GridSearchCV(model, param_grid_bagging, cv=5,
    ↪ scoring='accuracy')
grid_search_bagging.fit(X_train, y_train)

# Best parameters and score
print("Best parameters (Bagging):", grid_search_bagging.best_params_)
print("Best score (Bagging):", grid_search_bagging.best_score_)
```

Best parameters (Bagging): 'Bagging__max_samples': 0.7, 'Bagging__n_estimators': 100
Best score (Bagging): 0.8615384615384615

Vamos a utilizar como parámetros algunos parecidos a los que hemos utilizado a la hora de seleccionar y entrenar el árbol de decisiones.

```
X = df.loc[:, df.columns != 'Obesidad']
y = df['Obesidad']

# Crear un conjunto de entrenamiento y uno de prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    ↪ =0.2, random_state=seed)

model = Pipeline([
    ("Transformer", column_transformer),
    ("Random_Forest", RandomForestClassifier(random_state=seed))
])
```

```
# Parameters grid
param_grid_rf = {
    'Random_Forest__n_estimators': [100, 200, 300],
    'Random_Forest__max_depth': [None, 2, 4, 6, 8],
    'Random_Forest__min_samples_split': [10, 20]
}

# GridSearchCV
grid_search_rf = GridSearchCV(model, param_grid_rf, cv=5, scoring='
    ↪ accuracy')
grid_search_rf.fit(X_train, y_train)

# Best parameters and score
print("Best parameters (Random Forest):", grid_search_rf.best_params_)
print("Best score (Random Forest):", grid_search_rf.best_score_)
```

Best parameters (Random Forest): 'Random_Forest__max_depth': None, 'Random_Forest__min_samples_split': 5, 'Random_Forest__n_estimators': 100

Best score (Random Forest): 0.864102564102564

9. Búsqueda paramétrica para lograr el mejor modelo de Gradiente Bosting y XGBoost según Accuracy

Continuamos utilizando un código muy parecido al de los apartados anteriores:

```
model_gb = Pipeline([
    ("Transformer", column_transformer),
    ("Gradient_Boosting", GradientBoostingClassifier(random_state=seed)
        ↪ )
])

# Parameters grid for Gradient Boosting
param_grid_gb = {
    'Gradient_Boosting__n_estimators': [100, 200, 300],
    'Gradient_Boosting__learning_rate': [0.01, 0.1, 0.2],
    'Gradient_Boosting__max_depth': [3, 5, 7]
}

# GridSearchCV for Gradient Boosting
grid_search_gb = GridSearchCV(model_gb, param_grid_gb, cv=5, scoring='
    ↪ accuracy')
grid_search_gb.fit(X_train, y_train)

# Best parameters and score for Gradient Boosting
print("Best parameters (Gradient Boosting):", grid_search_gb.
    ↪ best_params_)
print("Best score (Gradient Boosting):", grid_search_gb.best_score_)
```

Best parameters (Gradient Boosting): 'Gradient_Boosting__learning_rate': 0.2, 'Gradient_Boosting__max_depth': 7, 'Gradient_Boosting__n_estimators': 200

Best score (Gradient Boosting): 0.8461538461538461

```
model_xgb = Pipeline([
    ("Transformer", column_transformer),
    ("XGBoost", XGBClassifier(use_label_encoder=False, eval_metric='
        ↪ logloss', random_state=seed))
])

# Parameters grid for XGBoost
param_grid_xgb = {
    'XGBoost__n_estimators': [100, 200, 300],
    'XGBoost__learning_rate': [0.01, 0.1, 0.2],
    'XGBoost__max_depth': [3, 5, 7],
    'XGBoost__min_child_weight': [1, 3, 5]
}

# GridSearchCV for XGBoost
grid_search_xgb = GridSearchCV(model_xgb, param_grid_xgb, cv=5, scoring
    ↪ ='accuracy')
grid_search_xgb.fit(X_train, y_train.astype(int))

# Best parameters and score for XGBoost
print("Best parameters (XGBoost):", grid_search_xgb.best_params_)
print("Best score (XGBoost):", grid_search_xgb.best_score_)
```

Best parameters (XGBoost): 'XGBoost__learning_rate': 0.2, 'XGBoost__max_depth': 7, 'XGBoost__min_child_weight': 1, 'XGBoost__n_estimators': 200

Best score (XGBoost): 0.8602564102564102

10. Búsqueda paramétrica para determinar el mejor modelo de SVM con al menos dos kernels diferentes

En el caso del SVM necesitamos cambiar ligeramente nuestra manera de hacer la búsqueda paramétrica para asegurarnos de que tiene dos kernels. Esto lo conseguimos añadiendo un segundo diccionario a la grid de parámetros que pasamos a grid search.

Vamos a seleccionar un tipo distinto para cada kernel.

```
from sklearn.svm import SVC

model_svc = Pipeline([
    ("Transformer", column_transformer),
    ("SVM", SVC(random_state=seed))
])

# Parameters grid for SVC
param_grid_svc = [
```

```
    {
        'SVM__kernel': ['rbf'],
        'SVM__gamma': [1e-3, 1e-4],
        'SVM__C': [1, 10, 100, 1000]
    },
    {
        'SVM__kernel': ['linear'],
        'SVM__C': [1, 10, 100, 1000]
    }
]

# GridSearchCV for SVC
grid_search_svc = GridSearchCV(model_svc, param_grid_svc, cv=5, scoring
    ↪ ='accuracy')
grid_search_svc.fit(X_train, y_train)

# Best parameters and score for SVC
print("Best parameters (SVM):", grid_search_svc.best_params_)
print("Best score (SVM):", grid_search_svc.best_score_)
```

Best parameters (SVM): 'SVM__C': 1000, 'SVM__gamma': 0.001, 'SVM__kernel': 'rbf'
Best score (SVM): 0.7679487179487179

Si tuviésemos una dimensionalidad menor podríamos representar este modelo gráficamente pero no es el caso.

11. Método de Ensamblado de Bagging con un mismo clasificador base que no sea un árbol

En este caso, vamos a elegir para hacer el ejemplo un modelo de clasificación SVM. Podríamos escoger otros ejemplos, pero como SVM es el último que hemos entrenado escojo este.

```
# Setup the pipeline with Bagging and SVC as the base estimator
model_bagging_svc = Pipeline([
    ("Transformer", column_transformer),
    ("Bagging_SVC", BaggingClassifier(base_estimator=SVC(probability=
        ↪ True, random_state=seed), random_state=seed))
])

# Parameters grid for Bagging with SVC
param_grid_bagging_svc = {
    'Bagging_SVC__n_estimators': [10, 20, 50],  # Number of base
        ↪ estimators
    'Bagging_SVC__max_samples': [0.5, 0.7, 1.0],  # Max number of
        ↪ samples to train each base estimator
    'Bagging_SVC__base_estimator__C': [0.1, 1, 10],  # Regularization
        ↪ parameter for the SVC
    'Bagging_SVC__base_estimator__kernel': ['linear', 'rbf']  # Kernel
        ↪ types for the SVC
```

```
}

# GridSearchCV for Bagging with SVC
grid_search_bagging_svc = GridSearchCV(model_bagging_svc,
    ↪ param_grid_bagging_svc, cv=5, scoring='accuracy')
grid_search_bagging_svc.fit(X_train, y_train)

# Best parameters and score for Bagging with SVC
print("Best parameters (Bagging with SVC):", grid_search_bagging_svc.
    ↪ best_params_)
print("Best score (Bagging with SVC):", grid_search_bagging_svc.
    ↪ best_score_)
```

Best parameters (Bagging with SVC): 'Bagging_SVC__base_estimator__C': 10, 'Bagging_SVC__base_estimator__kernel': 'rbf', 'Bagging_SVC__max_samples': 1.0, 'Bagging_SVC__n_estimators': 20

Best score (Bagging with SVC): 0.8538461538461538

12. Método de Stacking escogiendo varios algoritmos de entrada y el modelo que se prefiera como modelo de ensamblaje

Vamos a utilizar los tipos de modelos que hemos usado en el resto de apartados y como modelo de ensamblaje o meta-learner utilizaremos un modelo sencillo de regresión logística.

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier,
    ↪ GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from xgboost import XGBClassifier

# Define base models
base_models = [
    ('decision_tree', DecisionTreeClassifier(random_state=seed)),
    ('random_forest', RandomForestClassifier(random_state=seed)),
    ('svc', SVC(probability=True, random_state=seed)),
    ('gradient_boosting', GradientBoostingClassifier(random_state=seed)
        ↪ ),
    ('xgboost', XGBClassifier(use_label_encoder=False, eval_metric='
        ↪ logloss', random_state=seed)),
    ('mlp', MLPClassifier(random_state=seed))
]

# Define the meta-learner
meta_learner = LogisticRegression(random_state=seed)
```

```
# Setup the stacking classifier
model_stacking = StackingClassifier(
    estimators=base_models,
    final_estimator=meta_learner,
    cv=5,
    passthrough=False,
    stack_method='auto',
    n_jobs=-1
)

# Integrate the stacking classifier into a pipeline with preprocessing
pipeline_stacking = Pipeline([
    ("Transformer", column_transformer),
    ("Stacking", model_stacking)
])

# Example usage with train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    ↪ =0.2, random_state=seed)

# Fit the pipeline
pipeline_stacking.fit(X_train, y_train)

# Evaluate the model
accuracy = pipeline_stacking.score(X_test, y_test)
print(f"Stacking Model Accuracy: {accuracy}")
```

Stacking Model Accuracy: 0.9183673469387755

Hemos conseguido el modelo con mejor accuracy hasta ahora. Este modelo combina todos los tipos de modelo que hemos utilizado hasta el momento. Considero que hemos cumplido el objetivo que nos habíamos propuesto en la introducción.

Apéndice

En esta sección se incluyen los gráficos de la sección del análisis de variables y depuración de datos.

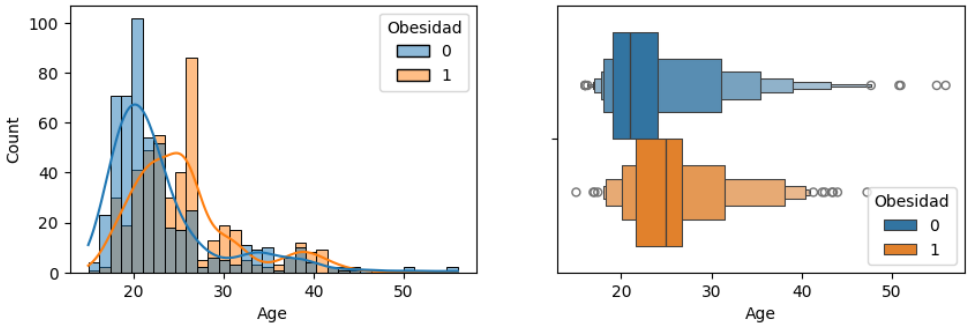

Figura 8

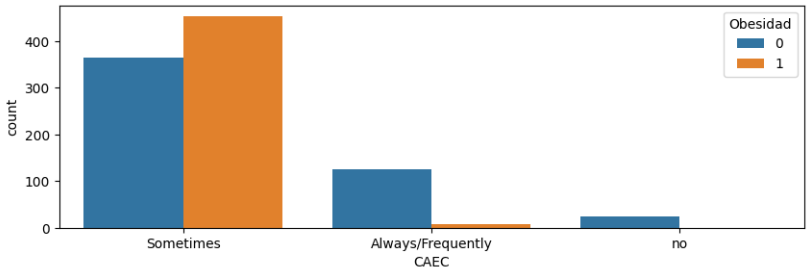

Figura 9

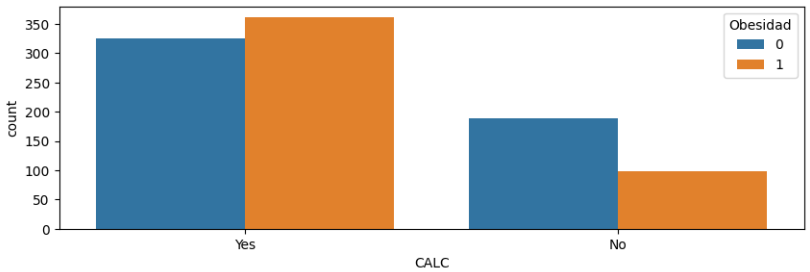

Figura 10

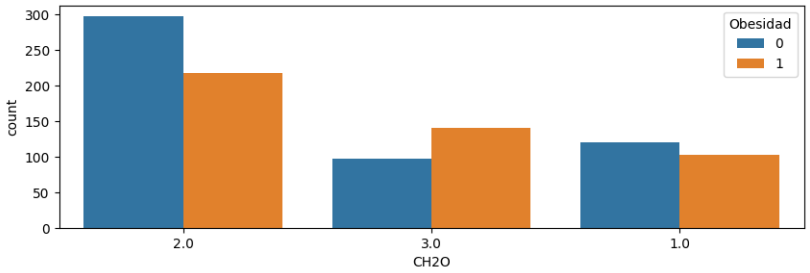

Figura 11

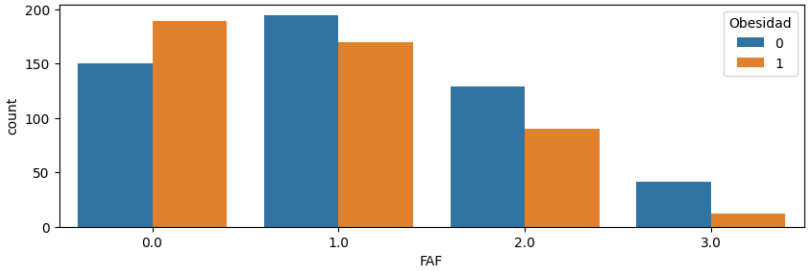

Figura 12

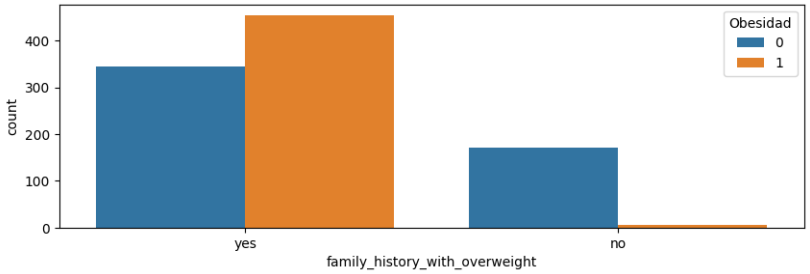

Figura 13

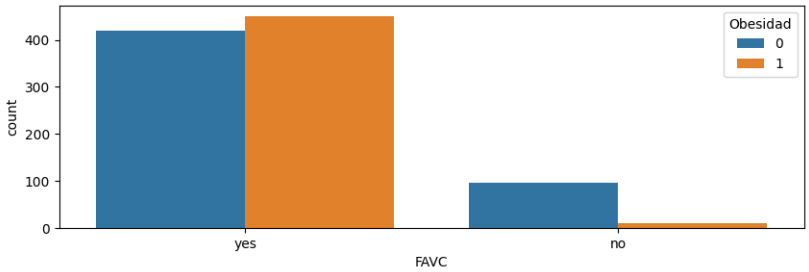

Figura 14

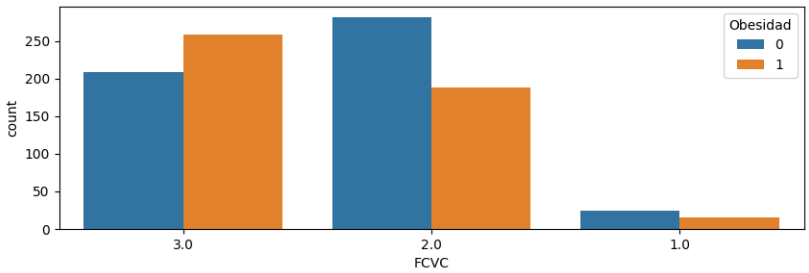

Figura 15

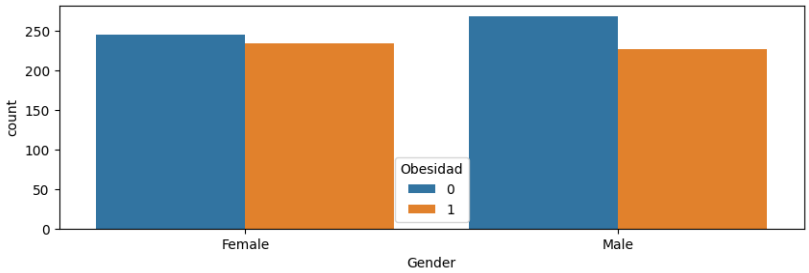

Figura 16

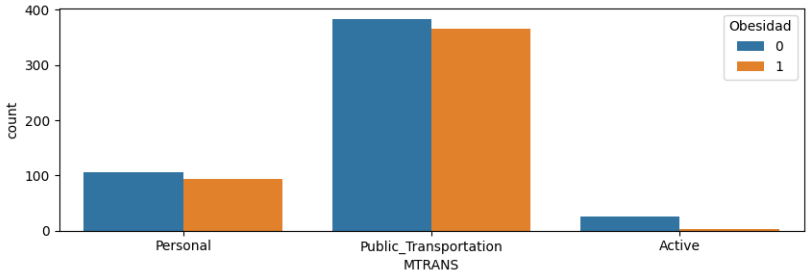

Figura 17

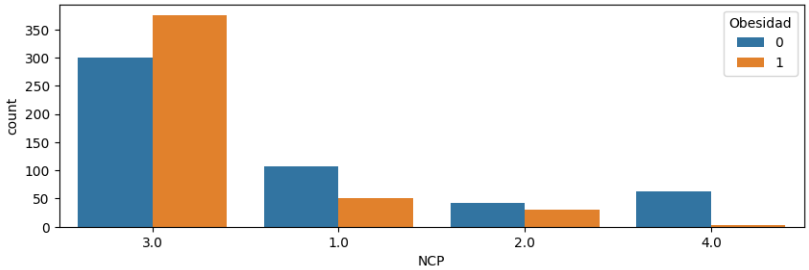

Figura 18

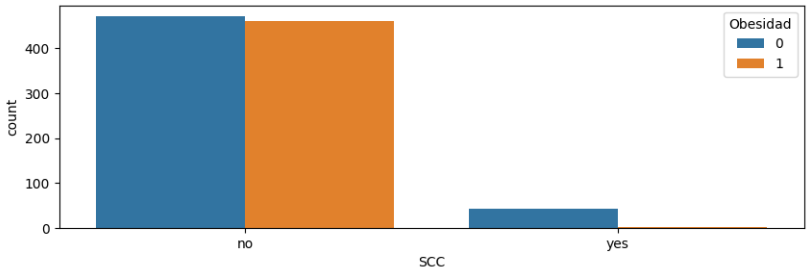

Figura 19

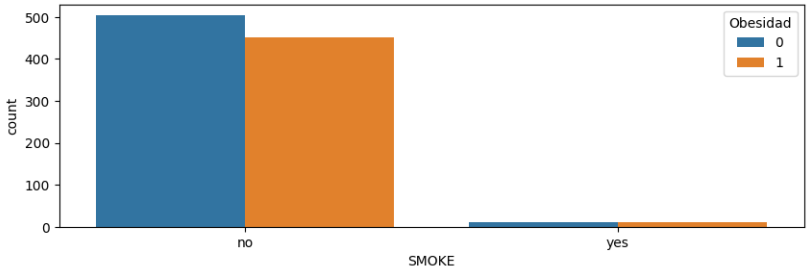

Figura 20

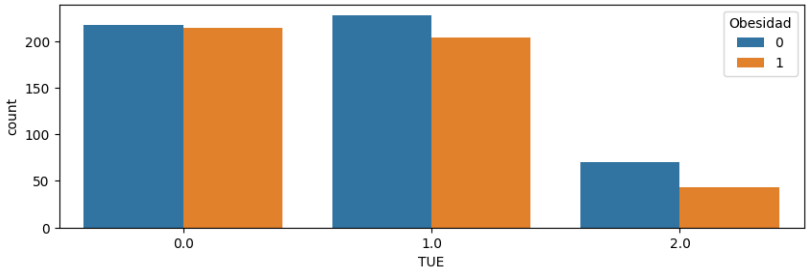

Figura 21