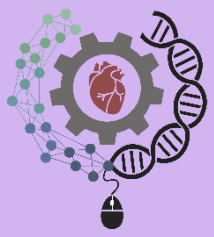
ESCUELA
POLITÉCNICA
SUPERIOR

Readmisión de paciente con diabetes.



Grado en Ingeniería Biomédica

Análisis de Datos Clínicos.

Esther Bermejo y Miguel Benito.
Ingeniería Biomédica. 2023-2024

Estudio sobre la readmisión de pacientes con diabetes.

Contenido

Índice Ilustraciones

Introducción

En esta práctica se realiza el estudio sobre la readmisión de pacientes con diabetes. A partir de la base de datos Diabetes 130-US Hospitals for Years 1999-2008 la cual representa 10 años de atención clínica en 130 hospitales y redes de distribución en los Estados Unidos. Cada fila en el conjunto de datos hará referencia a un hospital. Esta base está caracterizada por tener 101.766 instancias y 47 características.

Se utilizan herramientas de lenguaje Python como NumPy, Pandas, Scikit-learn para construir modelos predictivos. El estudio se divide en bloques primero se describen las variables, en segundo lugar, se realizará una búsqueda para seleccionar aquellos modelos que tienen mayor precisión tomando como estrategia la validación cruzada de 10 particiones y utilizando como medida el área bajo la curva.

Para realizar el estudio se revisan las características y se calcula el desequilibrio de la variable objetivo ("readmitted") con el fin de determinar el porcentaje de aciertos de un sistema trivial.

Objetivo

- Usar la herramienta el lenguaje Python (NumPy, Pandas, scikit-learn) para construir modelos predictivos;
- Buscar y seleccionar modelos con la mayor precisión, utilizando como medida el área bajo la Curva ROC (ROC Área) mediante la estrategia de validación cruzada de 10 particiones (10-CV, Cross Validation).
- La variable para predecir es readmitted (ubicada en última columna del fichero) compuesta por tres categorías (NO, <30 y >30)

Contenidos

- Revisar cada característica (Ej. numérica/categórica, valores desconocidos, etc.), calcular el desequilibrio de la variable objetivo ('readmitted') para determinar el porcentaje de aciertos de un sistema trivial.
- Utilizar el clasificador base sklearn. dummy.DummyClassifier() como referencia básica para obtener el valor del área bajo la curva ROC (AUC) y mejorarlo en las siguientes pruebas.
- Probar diferentes clasificadores con el objetivo de obtener el mayor AUC posible, ajustando hiperparámetros manualmente y evaluando diferentes algoritmos.
- Explorar la posibilidad de selección automática de atributos con diferentes algoritmos.

- Comparar los resultados (10-CV) usando la media aritmética, o con la prueba de contraste de hipótesis de Wilcoxon (signed-rank test).
- Tener en cuenta el elevado número de instancias del conjunto de datos y su impacto en el tiempo de procesamiento.

Desarrollo

Nivel Básico

Tratamiento de datos

Identificar las variables (columnas) y adaptar su tipo para que se pueda clasificar correctamente (Ej.: variables objetivas tienen que ser nominales y no numéricas, variables de texto que no se puede interpretar, etc.);

En primer lugar, se descarga la base de datos con recursos de las librerías 'requests' y 'zipfile'. Se lee el fichero .csv descargado y se almacenan los datos en la variable 'diabetic_data', esta será una de las variables principales.

Se ha realizado un estudio para conocer aquellas columnas que tienen mayor número de valores desconocidos, NaN y se ha decidido eliminarlas para que no interfieran en el resultado. En un principio se eliminaron las columnas *encounter_id* y *patient_nbr* por ser meramente identificadoras de los pacientes y no útiles para nuestro estudio. Además, también se eliminó la variable *examide* ya que se vio al analizar la base de datos que no aportaba ningún valor relevante. Al comenzar a realizar los clasificadores con toda la base de datos se observó que tardaban demasiado tiempo en ejecutarse, comprendimos que la base de datos necesitaba ser limpiada aún más. Para ello se buscaron los valores nulos que aparecían en cada una de ella y se vio que había tres columnas en las cuales había un gran número de ellos. Las siguientes columnas se han eliminado por este motivo y al hacerlo todos los clasificadores mejoraron notablemente su tiempo de entrenamiento.

Nombre de la columna	*Valores NaN*
weight	98569
max_glu_serum	96420
A1Cresult	84748
medical_specialty	49949
payer_code	40256
race	2273
diag_3	1423
diag_2	358
diag_1	21

Tabla 1. Variables y sus correspondientes valores NaN

Aunque se han eliminado las columnas que tienen valores desconocidos, no se puede eliminar todas las columnas que al menos contengan un valor desconocido. Cada característica está compuesta por 101.766 instancias, en comparación con el tamaño total del conjunto de datos la proporción de valores faltantes es insignificantes con relación a la totalidad, es por esto por lo que es muy poco probable que afecte significativamente a la calidad de los resultados del estudio.

```
La base de datos finalmente contiene las siguientes características:  Index(['race', 'gender', 'age', 'admission_type_id',
       'discharge_disposition_id', 'admission_source_id', 'time_in_hospital',
       'num_lab_procedures', 'num_procedures', 'num_medications',
       'number_outpatient', 'number_emergency', 'number_inpatient', 'diag_1',
       'diag_2', 'diag_3', 'number_diagnoses', 'metformin', 'repaglinide',
       'nateglinide', 'chlorpropamide', 'glimepiride', 'acetohexamide',
       'glipizide', 'glyburide', 'tolbutamide', 'pioglitazone',
       'rosiglitazone', 'acarbose', 'miglitol', 'troglitazone', 'tolazamide',
       'insulin', 'glyburide-metformin', 'glipizide-metformin',
       'glimepiride-pioglitazone', 'metformin-rosiglitazone',
       'metformin-pioglitazone', 'change', 'diabetesMed', 'readmitted'],
      dtype='object')

Headers of IDs
Index(['admission_type_id', 'description'], dtype='object')
```

Ilustración 1. Características de la base de datos.

En el conjunto de datos las características representan variables que almacenan tanto datos de tipo numérico, a las que llamamos variables numéricas, como datos de tipo categórico, a las que llamamos variables categóricas.

```
[36] numerical_columns = ['time_in_hospital','num_lab_procedures','num_procedures','num_medications','number_outpatient','number_emergency','number_inpatient','number_diagnoses']
     diabetic_data_numerical = diabetic_data[numerical_columns] #guardamos las columnas numericas
     diabetic_data_numerical.head()
```

	time_in_hospital	num_lab_procedures	num_procedures	num_medications	number_outpatient	number_emergency	number_inpatient	number_diagnoses
0	1	41	0	1	0	0	0	1
1	3	59	0	18	0	0	0	9
2	2	11	5	13	2	0	1	6
3	2	44	1	16	0	0	0	7
4	1	51	0	8	0	0	0	5

```
diabetic_data_numerical.describe()
```

	time_in_hospital	num_lab_procedures	num_procedures	num_medications	number_outpatient	number_emergency	number_inpatient	number_diagnoses
count	101766.000000	101766.000000	101766.000000	101766.000000	101766.000000	101766.000000	101766.000000	101766.000000
mean	4.395987	43.095641	1.339730	16.021844	0.369357	0.197836	0.635566	7.422607
std	2.985108	19.674362	1.705807	8.127566	1.267265	0.930472	1.262863	1.933600
min	1.000000	1.000000	0.000000	1.000000	0.000000	0.000000	0.000000	1.000000
25%	2.000000	31.000000	0.000000	10.000000	0.000000	0.000000	0.000000	6.000000
50%	4.000000	44.000000	1.000000	15.000000	0.000000	0.000000	0.000000	8.000000
75%	6.000000	57.000000	2.000000	20.000000	0.000000	0.000000	1.000000	9.000000
max	14.000000	132.000000	6.000000	81.000000	42.000000	76.000000	21.000000	16.000000

Ilustración 2. Creación de un subconjunto de variables numéricas.

Aunque todavía no es del todo cierto, los tipos de datos que almacena la base de datos son int64, float64, object y category, con el fin de tener las variables numéricas y categóricas correctamente codificadas se realiza una división creando un subconjunto de datos donde se guardan las variables numéricas y análogamente con las categóricas. Antes de codificar las variables categóricas se aparta la variable objetivo 'target' tal y como se puede apreciar en la siguiente ilustración.

```
target_name = "readmitted"
target = diabetic_data[target_name]

diabetic_data = diabetic_data.drop(columns=[target_name]) #se elimina la variable objetivo de la base de datos
#selecciona y almacena las tipo categorical
data_categorical = diabetic_data.select_dtypes([object])
categorical_columns = data_categorical.columns
data_categorical.head()
```

	race	gender	age	diag_1	diag_2	diag_3	metformin	repaglinide	nateglinide	chlorpropamide	...	troglitazone	tolazamide	insulin	glyburide-metformin	glipizide-metformin	glimepiride-pioglitazone	metformin-rosiglitazone	metformin-pioglitazone
0	Caucasian	Female	[0-10)	250.83	NaN	NaN	No	No	No	No	...	No	No	No	No	No	No	No	No
1	Caucasian	Female	[10-20)	276	250.01	255	No	No	No	No	...	No	No	Up	No	No	No	No	No
2	AfricanAmerican	Female	[20-30)	648	250	V27	No	No	No	No	...	No	No	No	No	No	No	No	No
3	Caucasian	Male	[30-40)	8	250.43	403	No	No	No	No	...	No	No	Up	No	No	No	No	No
4	Caucasian	Male	[40-50)	197	157	250	No	No	No	No	...	No	No	Steady	No	No	No	No	No

5 rows × 29 columns

Ilustración 3. Creación del subconjunto de variables categóricas.

Para realizar las conversiones del tipo de dato se crea un diccionario cuyas claves son los nombres de las columnas, utilizándolo se convierten las columnas categóricas, que ya se han almacenado previamente en una variable para conocer cuáles son, al tipo de dato 'categórico'. Se crea un DataFrame, con el mismo índice y columnas que la variable que almacena los datos categóricos. Este DataFrame almacena códigos numéricos de las categorías, se asignan mediante el mediante '`.cat.codes`'. Finalmente se representa el tipo de datos de cada variable.

```
dict_cat = {i:'category' for i in categorical_columns}  #creacción del diccionario.
diabetic_data_categorical = diabetic_data[categorical_columns].astype(dict_cat) #conversion columnas 'categóricas' al tipo de dato 'categórico.'

diabetic_data_categorical_codes = pd.DataFrame(   #Crea DataFrame
    index = diabetic_data_categorical.index,
    columns = diabetic_data_categorical.columns)

for i in diabetic_data_categorical.columns:#guarda los códigos numéricos
  diabetic_data_categorical_codes[i] = diabetic_data_categorical[i].cat.codes

#diabetic_data_categorical = diabetic_data_categorical_codes
print(diabetic_data_categorical_codes.head())
diabetic_data_categorical_codes.dtypes#imprime el tipo de datos

#creo una nueva base de datos con los nuevos datos
data= pd.concat([diabetic_data_numerical,diabetic_data_categorical_codes], axis=1)
```

```
   race  gender  age  diag_1  diag_2  diag_3  metformin  repaglinide  \
0     2       0    0     124      -1      -1          1            1
1     2       0    1     143      79     121          1            1
2     0       0    2     454      78     766          1            1
3     2       1    3     554      97     248          1            1
4     2       1    4      54      24      86          1            1

   nateglinide  chlorpropamide  ...  troglitazone  tolazamide  insulin  \
0            1               1  ...             0           0        1
1            1               1  ...             0           0        3
2            1               1  ...             0           0        1
3            1               1  ...             0           0        3
4            1               1  ...             0           0        2
```

Ilustración 4. Codificación de variables categóricas.

Por último, se realiza un mapeo de las etiquetas y valores de las variables categóricas convertidas.

```python
# Itera sobre cada columna categórica en el DataFrame de códigos numéricos y el DataFrame original
for columna in diabetic_data_categorical_codes:
    valores_unicos_codes = diabetic_data_categorical_codes[columna].unique()
    valores_unicos = diabetic_data_categorical[columna].unique()
    print(f"\nValores únicos de la columna '{columna}':")

    # Crea un diccionario de correspondencia entre códigos numéricos y etiquetas
    mapeo_correspondencia = {}
    for codigo, etiqueta in zip(valores_unicos_codes, valores_unicos):
        mapeo_correspondencia[codigo] = etiqueta

    print(f"Correspondencia entre códigos numéricos y etiquetas para la columna '{columna}':")
    for codigo, etiqueta in mapeo_correspondencia.items():
        print(f"{codigo}: {etiqueta}")
```

Ilustración 5. Mapeo de las etiquetas.

```
Valores únicos de la columna 'race':
Correspondencia entre códigos numéricos y etiquetas para la columna 'race':
2: Caucasian
0: AfricanAmerican
-1: nan
4: Other
1: Asian
3: Hispanic

Valores únicos de la columna 'gender':
Correspondencia entre códigos numéricos y etiquetas para la columna 'gender':
0: Female
1: Male
2: Unknown/Invalid

Valores únicos de la columna 'age':
Correspondencia entre códigos numéricos y etiquetas para la columna 'age':
0: [0-10)
1: [10-20)
2: [20-30)
3: [30-40)
4: [40-50)
5: [50-60)
6: [60-70)
7: [70-80)
8: [80-90)
9: [90-100)
```

Ilustración 6. Ejemplo del mapeo de las etiquetas 1.

```
Valores únicos de la columna 'acetohexamide':
Correspondencia entre códigos numéricos y etiquetas para la columna 'acetohexamide':
0: No
1: Steady

Valores únicos de la columna 'glipizide':
Correspondencia entre códigos numéricos y etiquetas para la columna 'glipizide':
1: No
2: Steady
3: Up
0: Down

Valores únicos de la columna 'glyburide':
Correspondencia entre códigos numéricos y etiquetas para la columna 'glyburide':
1: No
2: Steady
3: Up
0: Down

Valores únicos de la columna 'tolbutamide':
Correspondencia entre códigos numéricos y etiquetas para la columna 'tolbutamide':
0: No
1: Steady
```

Ilustración 7. Ejemplo del mapeo de las etiquetas 2.

Modelo Dummy Classifier

Usar un modelo como base (Dummy Classifier) y obtener sus resultados;

Se ha utilizado un modelo base para obtener los resultados con los que se podrá valorar si los siguientes clasificadores mejoran la predicción. Para ello se ha utilizado el fragmento de código que aparece a continuación:

```python
#DummyClasifier
from sklearn.model_selection import train_test_split
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score
#División de datos de entrenamiento y prueba.
data_train, data_test, target_train, target_test = train_test_split(diabetic_data, target, random_state=42, test_size=0.25)

# Mostrar el número de datos de entrenamiento y de test
print(f"Number of samples in testing: {data_test.shape[0]} => "
      f"{data_test.shape[0] / diabetic_data_numerical.shape[0] * 100:.1f}% of the"
      f" original set")
print(f"Number of samples in training: {data_train.shape[0]} => "
      f"{data_train.shape[0] /diabetic_data_numerical.shape[0] * 100:.1f}% of the"
      f" original set")

# Crear y ajustar el clasificador Dummy
dummy_clf = DummyClassifier(strategy="most_frequent")  # Puede ser "most_frequent", "stratified", "uniform", o "constant"
dummy_clf.fit(data_train, target_train)

# Realizar predicciones en el conjunto de prueba
y_pred = dummy_clf.predict(data_test)

# Calcular la precisión del clasificador Dummy
accuracy = accuracy_score(target_test, y_pred)
print("Predicción del Dummy Classifier:", y_pred)
print("Precisión del Dummy Classifier:", accuracy)
```

Ilustración 8. Código clasificador base.

Este conjunto de comandos nos devuelve los siguientes resultados:

- Number of samples in testing: 25442 => 25.0% of the original set
- Number of samples in training: 76324 => 75.0% of the original set
- Predicción del Dummy Classifier: ['NO' 'NO' 'NO' ... 'NO' 'NO' 'NO']

- Precisión del Dummy Classifier: 0.5374970521185441

Con ellos podemos afirmar que un 25% de la muestra de datos se utiliza para probar el modelo mientras que un 75% se utiliza para entrenarlo.

Se obtiene un valor de 0.53 de precisión que hace referencia a la medida de exactitud de un modelo predictivo o clasificador. Aproximadamente un 53.75% de las predicciones realizadas son correctas en comparación con el total de predicciones realizadas. Al obtener la proporción de cada clase de la variable objetivo en el conjunto de datos de vemos como el predictor falla en gran medida cuando el valor es diferente de NO.

Lineal Model y Decision Tree

Aplicar al menos 2 clasificadores distintos al base;

Por un lado, se ha creado un modelo lineal, para ello se crea una instancia del modelo de regresión logística, se entrena el modelo con los datos de entrenamiento 'data_train', y 'target_train' utilizando las etiquetas correspondientes mediante el método. fit (). Primero se divide el conjunto de datos en un conjunto de entrenamiento y otro de prueba, este clasificador se caracteriza por tener un número máximo de iteraciones de 5000 y 'penalty = None'. Después de entrenar el modelo se realizan predicciones sobre el conjunto de prueba utilizando el método 'predict' y se calcula la precisión del modelo. Por último, se utilizan las predicciones de probabilidad del modelo de regresión logística para calcular el Área bajo la curva para cada clase en comparación con el resto de las clases. La precisión del modelo es 0.569 y el área bajo la curva para cada clase es 0.6347.

```
#Muestreo aleatorio estratificado
data_train, data_test, target_train, target_test = train_test_split(data, target, test_size=0.2, stratify=target, random_state=42)

# Selección de características y entrenamiento del modelo
logistic_regression = make_pipeline(
    StandardScaler(),
    LogisticRegression(max_iter=5000, penalty=None)
)
logistic_regression.named_steps['logisticregression'].feature_names = data.columns.tolist()

logistic_regression.fit(data_train, target_train)

# Evaluación del modelo
predictions = logistic_regression.predict(data_test)
accuracy = accuracy_score(target_test, predictions)
print("Precisión del modelo:", accuracy)
```

Ilustración 9. Linear Model I

```
# Paso 7: Cálculo del AUC para cada clase versus el resto
predictions_auc = logistic_regression.predict_proba(data_test)
auc_scores_dummy = roc_auc_score(target_test, predictions_auc, multi_class='ovr')
# Imprime los AUC scores para cada clase
print("AUC scores for each class (Logistic Regression):", auc_scores_dummy)
```

Ilustración 10. Linear Model II

Por otro lado, se creado un árbol de decisión (Decission Tree) de la misma manera. En un inicio se realizaron de manera individual para más adelante juntar los modelos para realizar el siguiente apartado de una manera más sencilla. El modelo del árbol de decisión siguió la misma linea que el anterior modelo, en primer lugar, se tomaron los subconjuntos de datos de entrenamiento para entrenar al modelo, se hicieron las predicciones de la probabilidad en el conjunto de prueba para obtener la precisión del modelo. Este último tiene una precisión de 0.525, mejor que el anterior por lo que, aunque se tengan valores parecidos este modelo tiene más error que el anterior.

```
# Inicializar el clasificador Árbol de Decisión
tree_classifier = DecisionTreeClassifier(random_state=42)

# Entrenar el clasificador
tree_classifier.fit(data_train, target_train)

# Predecir las probabilidades en el conjunto de prueba
y_proba_tree = tree_classifier.predict_proba(data_test)
auc_scores = roc_auc_score(target_test, y_proba_tree, multi_class='ovr')
print(auc_scores)
```

Ilustración 11. Decission Tree

Un detalle para destacar en ambos códigos y en los próximos clasificadores es que para calcular el área bajo la curva ROC para un problema de clasificación multi-clase se puede enfrentar al desafío de cómo calcularlo adecuadamente ya que la curva ROC es una herramienta originalmente diseñada para problemas de clasificación binaria. Debido a que nuestro estudio es un problema de clasificación multi-clase se calcula el área bajo la curva ROC para cada clase individualmente comparando esa clase con todas las demás clases agrupadas. Por eso se especifica el parámetro 'multi_class = 'ovr'' en la función 'roc_auc_score', se calculan los AUC individuales para cada clase frente al resto y luego se promedian para obtener un único valor de AUC para el modelo multi-clase.

Esto implica que se está evaluando el rendimiento del modelo en la capacidad de distinguir cada clase de las demás considerando una clase a la vez como positiva y las demás como negativas, es decir, se evalúa el rendimiento del modelo en clasificación multi clase considerando cada clase como positiva en un análisis de uno contra todos.

Wilcoxon y validación cruzada

Aplicar al menos una vez el contraste de hipótesis con Wilcoxon signed-rank test a los resultados del 10-fold cross-validation (más información en scikit-learn módulo de validación cruzada);

Antes de aplicar el contraste de hipótesis con Wilcoxon signed-rank se hace 10-fold cross validation. La validación cruzada se ha hecho definiendo una función que

automatiza la evaluación de cada modelo y devuelve tanto la precisión como el valor de la curva ROC. Se ha realizado del siguiente conjunto de modelos.

```
classifiers = {
    "Dummy": DummyClassifier(),
    "Logistic Regression": LogisticRegression(max_iter=5000, penalty=None),
    "AdaBoost": AdaBoostClassifier(),
    "Naive Bayes": GaussianNB(),
    "Random Forest": RandomForestClassifier(),
    "Decision Tree": DecisionTreeClassifier(),
    "Gradient Boosting": GradientBoostingClassifier(),
    "KNN": KNeighborsClassifier()
}
```

Ilustración 12. Conjunto de modelos

Funciona de la siguiente manera se dividen los datos en conjuntos de entrenamiento y prueba según la estrategia de validación cruzada, asignándole el objeto 'StratifiedKFold'. Se entrena el modelo en los datos de entrenamiento, y posteriormente utilizando la siguiente función se evalúan.

```
# Función para evaluar los modelos en cada pliegue
def evaluate_model(model, data, target):
    accuracy_scores = cross_val_score(model, data, target, cv=skf, scoring='accuracy')
    auc_scores = cross_val_score(model, data, target, cv=skf, scoring='roc_auc_ovr')
    return accuracy_scores, auc_scores
```

Ilustración 13. Función auxiliar de 10-fold-validation

Se han almacenado los clasificadores es un diccionario, y se ha recorrido este para la evaluación. Después de esto se han almacenado los resultados en un DataFrame tal y como se puede ver en la siguiente ilustración.

Los resultados de la validación cruzada de los métodos anteriormente comentados son:

	Mean Accuracy	Mean AUC
Dummy	0.539660	0.500000
Logistic Regression	0.569415	0.635452
AdaBoost	0.572454	0.636290
Naive Bayes	0.119622	0.580356
Random Forest	0.568838	0.624442
Decision Tree	0.458060	0.529609
Gradient Boosting	0.577027	0.648528
KNN	0.484867	0.542807

Ilustración 14. Resultados validación cruzada

1. Dummy Classifier: Este modelo tiene una precisión media de 0.539 y AUC medio de 0.500, predice siempre la clase más frecuente en los datos de entrenamiento. El modelo predice correctamente alrededor del 53.9% de las

muestras, y por otro lado el AUC medio indica que el modelo no es capaz de distinguir entre las clases. Su capacidad para distinguir entre las clases es muy baja, ya que su AUC es cercano al azar 0.5, por lo que es similar a un clasificador aleatorio por su baja tasa de acierto.

2. Logistic Regression: La precisión media de la regresión logística es de 0.569 y un AUC medio de 0.635. Aunque es más preciso que el Dummy Classifier, su rendimiento podría ser mejor. Sin embargo, su capacidad para distinguir entre las clases es mejor que la del Dummy Classifier, ya que su AUC medio es mayor.

3. AdaBoost Classifier: El clasificador AdaBoost tiene una precisión media de 0.572 y un AUC medio de 0.636, es ligeramente mejor que la regresión logística tanto en precisión como en AUC medio, por lo que es más preciso y tiene un rendimiento mejor que los dos anteriores.

4. Naive Bayes: Este modelo es menos preciso que el anterior, es el de menor precisión de todos, aunque tiene un rendimiento parecido, pero ligeramente menor. Se caracteriza por una precisión media de 0.119 y un AUC medio de 0.580.

5. Random Forest: El modelo Random Forest tiene una precisión media del 56.8% y un AUC medio de 0.629, tal y como se puede observar son valores parecidos a los del modelo de Logistic Regression y AdaBoost Classifier tanto para la precisión como para el rendimiento del modelo.

6. Decission Tree: El Árbol de Decisión tiene la precisión media más baja de todos los modelos evaluados en concreto la precisión media es de 0.458 y un AUC medio de 0.529, este último valor indica un rendimiento modesto en la distinción entre las clases.

7. Gradiente Boosting esta caracterizado por tener una precisión de 0.577 y una capacidad de discriminación de 0.648. Es la mejor capacidad de discriminación de todos los modelos utilizados. En términos de capacidad de discriminación, el modelo Gradient Boosting alcanza un valor de 0.648, medido mediante el área bajo la curva ROC (AUC). Esta métrica cuantifica la capacidad del modelo para distinguir entre clases positivas y negativas. Un AUC más alto, como en este caso, indica una mejor capacidad de discriminación.

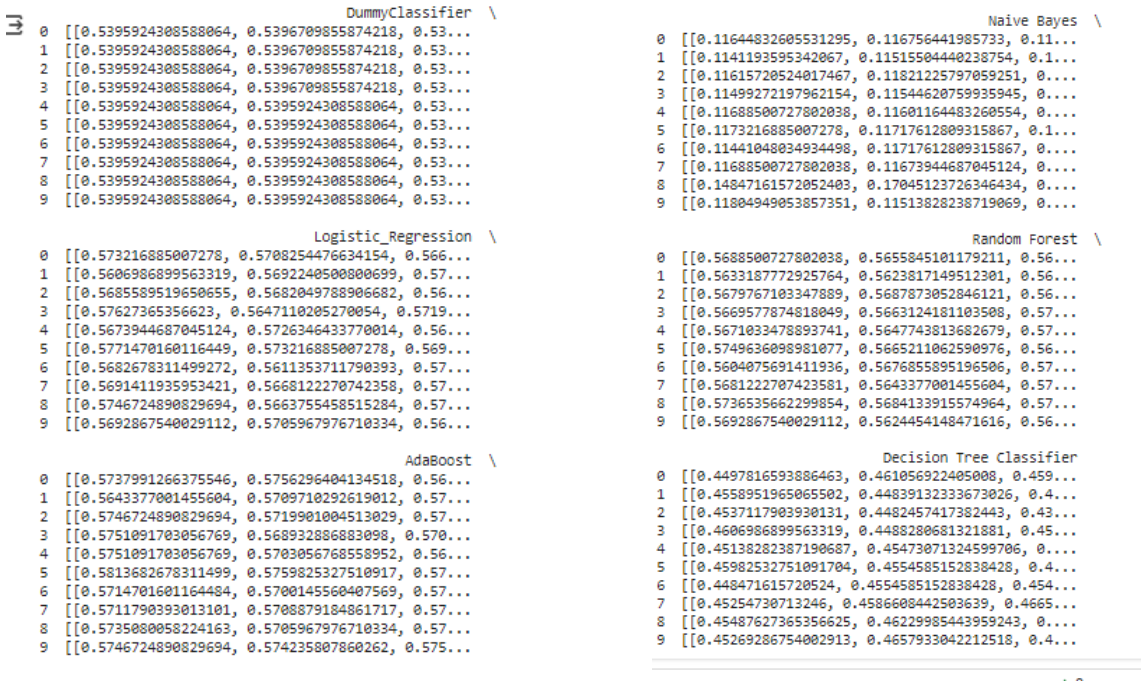
8. KNeighborsClassifier muestra una precisión del 48.4% y un área bajo la curva ROC (AUC) de 0.542. Aunque la precisión es relativamente baja, sugiere que alrededor del 48.4% de las predicciones son correctas. El AUC de 0.542 indica una capacidad de discriminación moderada del modelo para distinguir entre clases positivas y negativas. Estos resultados resaltan la

necesidad de considerar cuidadosamente el rendimiento del modelo en relación con el contexto del problema y las necesidades específicas del análisis.

Se puede observar que los modelos de mejor rendimiento son 'Gradient Boosting', 'AdaBoost', y 'Logistic Regression', son los que presentan valores más altos tanto de AUC como accuracy. Estos hechos significan que estos modelos generalizan de forma correcta a partir de los datos de entrenamiento a nuevos datos de prueba.

Asimismo, implica que estan capturando de manera efectiva las relaciones subyacentes en los datos y utilizando esa información para hacer predicciones precisas, una de las causas podría ser su capacidad para capturar relaciones no lineales como el Gradient Boosting o para modelar la probabilidad de las clases como el caso de la regresión logística. Los dos primeros modelos son los que mejores resultados obtienen debido a su funcionamiento intrínseco, ya que para eso utilizan múltiples modelos débiles combinados y esto resulta beneficioso para muchos escenarios, como es el caso del estudio de la readmisión de un paciente.

Antes de hacer una simplificación del modelo se hicieron distintas iteraciones de validación cruzada para obtener una estimación más precisa del rendimiento. Tal y como se puede observar en las siguientes imágenes los datos no solo corresponden y son coherentes, sino que también reflejan la exactitud de los modelos y siguen la tendencia marcada por los anteriores.



	DummyClassifier
0	[[0.5395924308588064, 0.5396709855874218, 0.53...
1	[[0.5395924308588064, 0.5396709855874218, 0.53...
2	[[0.5395924308588064, 0.5396709855874218, 0.53...
3	[[0.5395924308588064, 0.5396709855874218, 0.53...
4	[[0.5395924308588064, 0.5395924308588064, 0.53...
5	[[0.5395924308588064, 0.5395924308588064, 0.53...
6	[[0.5395924308588064, 0.5395924308588064, 0.53...
7	[[0.5395924308588064, 0.5395924308588064, 0.53...
8	[[0.5395924308588064, 0.5395924308588064, 0.53...
9	[[0.5395924308588064, 0.5395924308588064, 0.53...

	Logistic_Regression
0	[[0.573216885007278, 0.5708254476634154, 0.566...
1	[[0.5606986899563319, 0.5692240500800699, 0.57...
2	[[0.5685589519650655, 0.5682049788906682, 0.56...
3	[[0.576273653566623, 0.5647110205270054, 0.5719...
4	[[0.5673944687045124, 0.5726346433770014, 0.56...
5	[[0.5771470160116449, 0.573216885007278, 0.569...
6	[[0.5682678311499272, 0.5611353711790393, 0.57...
7	[[0.5691411935953421, 0.5668122270742358, 0.57...
8	[[0.5746724890829694, 0.5663755458515284, 0.57...
9	[[0.5692867540029112, 0.5705967976710334, 0.56...

	AdaBoost
0	[[0.5737991266375546, 0.5756296404134518, 0.56...
1	[[0.5643377001455604, 0.5709710292619012, 0.57...
2	[[0.5746724890829694, 0.5719901004513029, 0.57...
3	[[0.5751091703056769, 0.568932886883098, 0.570...
4	[[0.5751091703056769, 0.5703056768558952, 0.56...
5	[[0.5813682678311499, 0.5759825327510917, 0.57...
6	[[0.5714701601164484, 0.5700145560407569, 0.57...
7	[[0.5711790393013101, 0.5708879184861717, 0.57...
8	[[0.5735080058224163, 0.5705967976710334, 0.57...
9	[[0.5746724890829694, 0.574235807860262, 0.575...

	Naive Bayes
0	[[0.11644832605531295, 0.116756441985733, 0.11...
1	[[0.1141193595342067, 0.11515504440238754, 0.1...
2	[[0.11615720524017467, 0.11821225797059251, 0....
3	[[0.11499272197962154, 0.11544620759935945, 0....
4	[[0.11688500727802038, 0.11601164483260554, 0....
5	[[0.1173216885007278, 0.11717612809315867, 0.1...
6	[[0.11441048034934498, 0.11717612809315867, 0....
7	[[0.11688500727802038, 0.11673944687045124, 0....
8	[[0.14847161572052403, 0.17045123726346434, 0....
9	[[0.11804949053857351, 0.11513828238719069, 0....

	Random Forest
0	[[0.5688500727802038, 0.5655845101179211, 0.56...
1	[[0.5633187772925764, 0.5623817149512301, 0.56...
2	[[0.5679767103347889, 0.5687873052846121, 0.56...
3	[[0.5669577874818049, 0.5663124181103508, 0.57...
4	[[0.5671033478893741, 0.5647743813682679, 0.57...
5	[[0.5749636098981077, 0.5665211062590976, 0.56...
6	[[0.5604075691411936, 0.5676855895196506, 0.57...
7	[[0.5681222707423581, 0.5643377001455604, 0.57...
8	[[0.5736535662299854, 0.5684133915574964, 0.57...
9	[[0.5692867540029112, 0.5624454148471616, 0.56...

	Decision Tree Classifier
0	[[0.4497816593886463, 0.461056922405008, 0.459...
1	[[0.4558951965065502, 0.44839132333673026, 0.4...
2	[[0.4537117903930131, 0.4482457417382443, 0.43...
3	[[0.4606986899563319, 0.4488280681321881, 0.45...
4	[[0.45138282387190687, 0.45473071324599706, 0....
5	[[0.45982532751091704, 0.4554585152838428, 0.4...
6	[[0.448471615720524, 0.4554585152838428, 0.454...
7	[[0.45254730713246, 0.4586608442503639, 0.4665...
8	[[0.45487627365356625, 0.46229985443959243, 0....
9	[[0.45269286754002913, 0.4657933042212518, 0.4...

Una vez que se tienen los resultados de la validación cruzada se eligen dos algoritmos, se puede aplicar el test de Wilcoxon signed-rank para determinar si hay una diferencia significativa en su rendimiento. El test se encarga de evaluar si las diferencias observadas en el rendimiento entre los dos algoritmos son estadísticamente significativas o simplemente debido a la variabilidad aleatoria en los datos de entramiento y prueba.

Se representan los resultados de la validación cruzada tanto del valor de AUC como de la precisión de cada modelo.

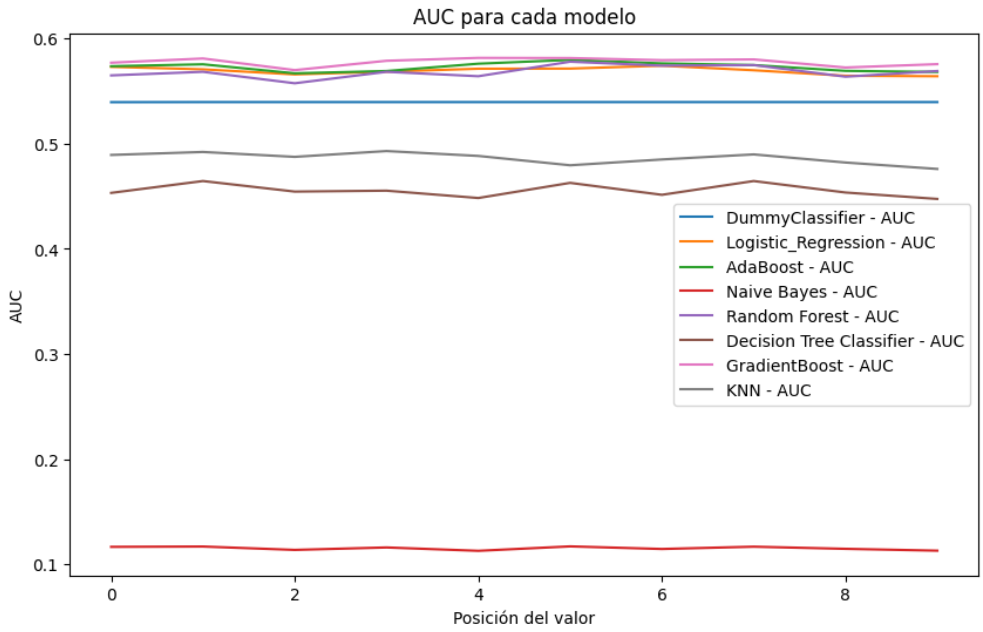


Ilustración 15. Comparación de AUC.

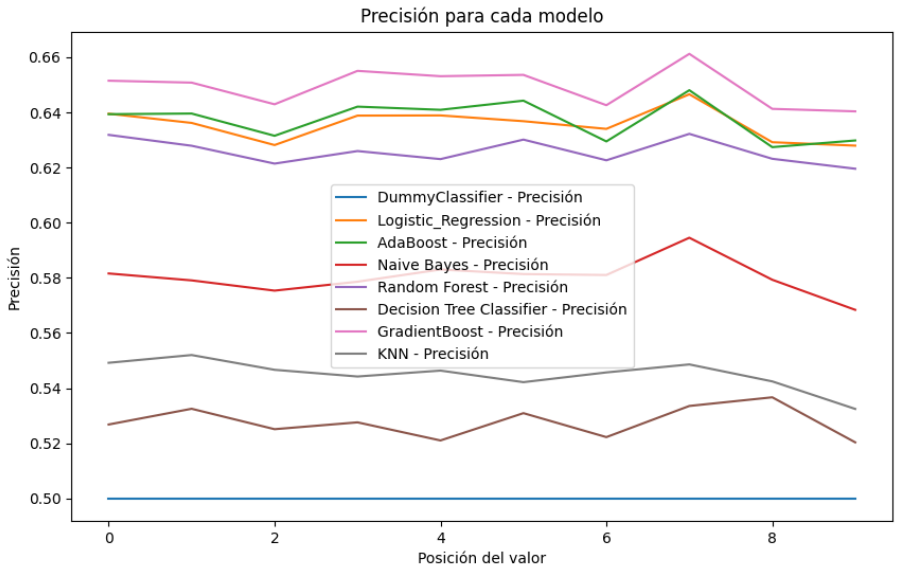


Ilustración 16. Resultados de la precisión de cada modelo.

Primero, el código extrae los datos de AUC y precisión para cada par de modelos a comparar. Luego, realiza el test de Wilcoxon para determinar si hay una diferencia significativa en el rendimiento entre los modelos. Se realiza el test para los siguientes pares de modelos:

1. Dummy Classifier y Regresión Logística
2. AdaBoost y Regresión Logística
3. Random Forest y Regresión Logística
4. Random Forest y Gradient Boost
5. Gradient Boost y Regresión Logística

Para cada par de modelos, el código imprime los resultados del test de Wilcoxon, incluyendo el valor de Wilcoxon (V) y el valor p. Estos resultados permiten evaluar si hay una diferencia significativa en el rendimiento entre los modelos en términos de AUC y precisión.

```python
print('DummyClassifier y regresión logistica')
# Extraer los datos de AUC y precisión para cada modelo
auc_scores = [modelo[1] for modelo in resultado_1]
precision_scores = [modelo[2] for modelo in resultado_1]

# Realizar el test de Wilcoxon
wilcox_V_auc, p_value_auc = wilcoxon(auc_scores[0], auc_scores[1], alternative='greater', zero_method='wilcox', correction=False)
wilcox_V_precision, p_value_precision = wilcoxon(precision_scores[0], precision_scores[1], alternative='greater', zero_method='wilcox', correction=False)

# Imprimir resultados
print('Resultados completos del test de Wilcoxon para AUC:')
print(f'Wilcox V: {wilcox_V_auc}, p-value: {p_value_auc:.2f}')

print('\nResultados completos del test de Wilcoxon para precisión:')
print(f'Wilcox V: {wilcox_V_precision}, p-value: {p_value_precision:.2f}')
```

Ilustración 17. Código fuente de ejecución de Wilcoxon

```
DummyClassifier y regresión logistica
Resultados completos del test de Wilcoxon para AUC:
Wilcox V: 0.0, p-value: 1.00

Resultados completos del test de Wilcoxon para precisión:
Wilcox V: 0.0, p-value: 1.00

Adaboost y regresión logistica
Resultados completos del test de Wilcoxon para AUC:
Wilcox V: 55.0, p-value: 0.00

Resultados completos del test de Wilcoxon para precisión:
Wilcox V: 42.0, p-value: 0.08

Random Forest y regresión logistica
Resultados completos del test de Wilcoxon para AUC:
Wilcox V: 15.0, p-value: 0.81

Resultados completos del test de Wilcoxon para precisión:
Wilcox V: 0.0, p-value: 1.00
```

Ilustración 18. Resultados Wilcoxon 1

```
Random Forest y Gradient Boost
Resultados completos del test de Wilcoxon para AUC:
Wilcox V: 0.0, p-value: 1.00

Resultados completos del test de Wilcoxon para precisión:
Wilcox V: 55.0, p-value: 0.00

Gradient Boost y regresión logistica
Resultados completos del test de Wilcoxon para AUC:
Wilcox V: 55.0, p-value: 0.00

Resultados completos del test de Wilcoxon para precisión:
Wilcox V: 55.0, p-value: 0.00
```

Ilustración 19. Resultados Wilcoxon 2

Los resultados presentados son el producto de pruebas de hipótesis realizadas mediante el test de Wilcoxon, las cuales tienen como objetivo comparar el desempeño de diferentes pares de modelos en términos de AUC y precisión. En cada comparación, se analiza si existe una diferencia significativa entre los modelos evaluados.

Primero, se compararon el Dummy Classifier y la regresión logística. Tanto para AUC como para precisión, los resultados indicaron que no hay una diferencia significativa en el rendimiento entre estos dos modelos, ya que los valores de p obtenidos fueron mayores que el nivel de significancia comúnmente utilizado ($\alpha = 0.05$). Posteriormente, se evaluó la comparación entre AdaBoost y la regresión logística. En términos de AUC, se encontró que AdaBoost tiene un rendimiento significativamente mejor que la regresión logística, con un valor de p igual a 0.00. Sin embargo, para la precisión, aunque se observó una diferencia en el rendimiento, el valor de p fue mayor que 0.05, lo que indica que la diferencia no fue estadísticamente significativa.

Otra comparación importante fue entre Random Forest y la regresión logística. Los resultados mostraron que no hay una diferencia significativa entre estos dos modelos en términos de AUC o precisión, ya que los valores de p fueron mayores que 0.05. Luego, se compararon Random Forest y Gradient Boost. Para ambos aspectos evaluados, no se encontraron diferencias significativas en el rendimiento entre estos dos modelos, ya que los valores de p fueron iguales a 1.00.

Finalmente, se evaluó la comparación entre Gradient Boost y la regresión logística. Los resultados indicaron que Gradient Boost tiene un rendimiento significativamente mejor que la regresión logística, tanto en términos de AUC como de precisión, con valores de p iguales a 0.00.

Más adelante a lo largo de la práctica nos dimos cuenta de que este método no comparaba realmente todos los modelos entre ellos por lo que decidimos realizar un bucle for que analizará mediante wilcoxon todos los métodos. Gracias al equipo

de Blanca Bel y Pablo Candela conseguimos mostrar los datos en formato tabla para que fueran mucho más fáciles de comprender.

Resultados del test de Wilcoxon

	Dummy	Logistic Regression	AdaBoost	Naive Bayes
Dummy	-	x	x	x
Logistic Regression	✓	-	x	✓
AdaBoost	✓	✓	-	✓
Naive Bayes	✓	x	x	-
Random Forest	✓	x	x	x
Decision Tree	✓	x	x	x
Gradient Boosting	✓	✓	✓	✓
KNN	✓	x	x	x

	Random Forest	Decision Tree	Gradient Boosting	KNN
Dummy	x	x	x	x
Logistic Regression	✓	✓	x	✓
AdaBoost	✓	✓	x	✓
Naive Bayes	x	✓	x	✓
Random Forest	-	✓	x	✓
Decision Tree	x	-	x	x
Gradient Boosting	✓	✓	-	✓
KNN	x	✓	x	-

Ilustración 20. Resultados Wilconxon 3

En resumen, estos análisis proporcionan una comprensión detallada del rendimiento relativo de diferentes modelos, lo que puede ser útil para la selección y la toma de decisiones en aplicaciones prácticas de aprendizaje automático.

Hiperparametros

Modificar alguno de los hiperparámetros de los clasificadores (de forma manual) para intentar mejorar el AUC y si es posible también su tasa de aciertos;

Se han modificado los hiperparámetros del modelo linear, del AdaBoost y del Gradient Boosting con el objetivo de mejorar aún más su rendimiento y el valor de la curva ROC. En primer lugar, se hacen las modificaciones del modelo linear, el cual tiene un valor de precisión de 0.569 y de AUC para cada clase de 0.6347. Se modifica el parámetro '*penalty*' este puede afectar al rendimiento del modelo, por lo general se prefiere 'l2' por su capacidad para manejar mejor la multicolinealidad entre las características y 'l1' para la selección de características y reducción de dimensionalidad.

También se ha modificado el parámetro '*solver*' este hecho puede influir en la convergencia y el rendimiento del modelo, algunos son más adecuados para problemas de regularización como '*liblinear*' y 'saga' mientras que '*lbgs*' y 'newton-cg' son más adecuados para la regularización.

```
Resultados del modelo con solver='liblinear', penalty='l1', max_iter=100:
Precisión del modelo:0.5657
AUC score del modelo:0.6346
Resultados del modelo con solver='liblinear', penalty='l1', max_iter=1000:
Precisión del modelo:0.5657
AUC score del modelo:0.6346
Resultados del modelo con solver='liblinear', penalty='l1', max_iter=10000:
Precisión del modelo:0.5657
AUC score del modelo:0.6346
Resultados del modelo con solver='saga', penalty='l1', max_iter=100:
Precisión del modelo:0.5690
AUC score del modelo:0.6367
Resultados del modelo con solver='saga', penalty='l1', max_iter=1000:
Precisión del modelo:0.5690
AUC score del modelo:0.6367
Resultados del modelo con solver='saga', penalty='l1', max_iter=10000:
Precisión del modelo:0.5690
AUC score del modelo:0.6367
Resultados del modelo con solver='lbfgs', penalty='l2', max_iter=100:
Precisión del modelo:0.5691
AUC score del modelo:0.6367
Resultados del modelo con solver='lbfgs', penalty='l2', max_iter=1000:
Precisión del modelo:0.5691
AUC score del modelo:0.6367
Resultados del modelo con solver='lbfgs', penalty='l2', max_iter=10000:
Precisión del modelo:0.5691
AUC score del modelo:0.6367
Resultados del modelo con solver='newton-cg', penalty='l2', max_iter=100:
Precisión del modelo:0.5690
AUC score del modelo:0.6367
Resultados del modelo con solver='newton-cg', penalty='l2', max_iter=1000:
Precisión del modelo:0.5690
AUC score del modelo:0.6367
```

Ilustración 21. Modificación Linear Model

Tal y como se puede observar los mejores hiperparámetros de los probados anteriormente son las siguientes combinaciones.

```
Resultados del modelo con solver='lbfgs', penalty='l2', max_iter=100:
Precisión del modelo:0.5691
AUC score del modelo:0.6367
Resultados del modelo con solver='lbfgs', penalty='l2', max_iter=1000:
Precisión del modelo:0.5691
AUC score del modelo:0.6367
Resultados del modelo con solver='lbfgs', penalty='l2', max_iter=10000:
Precisión del modelo:0.5691
AUC score del modelo:0.6367
```

Ilustración 22. Mejores hiperparámetros para el Linear Model

En segundo lugar, se modifican los hiperparámetros del modelo AdaBoost, los principales son:

- **n_estimators**: Especifica el número de estimadores bases (por defecto, árboles de decisión débiles) en el ensemble. Un valor más alto de *n_estimators* puede mejorar el rendimiento del modelo, pero también aumenta el tiempo de entrenamiento y el riesgo de sobreajuste.
- **learning_rate**: Controla la contribución de cada clasificador base al modelo final. Un valor más bajo de *learning_rate* significa que los clasificadores

contribuyen menos al modelo final, lo que puede ayudar a evitar el sobreajuste. Sin embargo, esto generalmente requiere un valor más alto de *n_estimators*.

- **base_estimator**: Especifica el tipo de clasificador débil que se utilizará en el ensemble. Por defecto, se utiliza un árbol de decisión con profundidad 1 (un árbol de decisión débil). Se puede especificar cualquier otro clasificador débil para adaptarse mejor a los datos.

Se han combinados distintos valores de n_estimators, 200 y 350 y de learning_rate, 0.8 y 0.5. Dando lugar a los siguientes resultados:

```
Resultados del clasificador con n_estimators = 200 y learning_rate = 0.8:
Precisión del modelo AdaBoost: 0.5739
AUC scores for each class (AdaBoost): 0.6403
Resultados del clasificador con n_estimators = 200 y learning_rate = 0.5:
Precisión del modelo AdaBoost: 0.5760
AUC scores for each class (AdaBoost): 0.6412
Resultados del clasificador con n_estimators = 350 y learning_rate = 0.8:
Precisión del modelo AdaBoost: 0.5742
AUC scores for each class (AdaBoost): 0.6404
Resultados del clasificador con n_estimators = 350 y learning_rate = 0.5:
Precisión del modelo AdaBoost: 0.5751
AUC scores for each class (AdaBoost): 0.6409
```

Ilustración 23. Mejores hiperparámetros para AdaBoost

La mejor modificación es n_estimators = 200 y learning_rate = 0.5. Sin embargo, la mejoría no era significativa y se probó con más valores, para n_estimators = 350, 450, 500 y 1000, mientras que para learning_rate = 0.5, 0.4, y 0.2. De todas las combinaciones resultantes a continuación se muestra la final.

```
Resultados del clasificador con n_estimators = 500 y learning_rate = 0.5:
Precisión del modelo AdaBoost: 0.5749
AUC scores for each class (AdaBoost): 0.6431
Resultados del clasificador con n_estimators = 500 y learning_rate = 0.2:
Precisión del modelo AdaBoost: 0.5746
AUC scores for each class (AdaBoost): 0.6438
Resultados del clasificador con n_estimators = 1000 y learning_rate = 0.5:
Precisión del modelo AdaBoost: 0.5742
AUC scores for each class (AdaBoost): 0.6426
Resultados del clasificador con n_estimators = 1000 y learning_rate = 0.2:
Precisión del modelo AdaBoost: 0.5746
AUC scores for each class (AdaBoost): 0.6436
```

Ilustración 24. Segundas modificaciones del clasificador.

La mejor combinación fue n_estimators = 500 y learning_rate = 0.2, obteniendo este resultado accuracy = 0.5746 y AUC = 0.6438. La modificación aumenta los resultados no de manera significativa, pero si se obtienen mejores que anteriormente.

Por último, se realizó la modificación para el modelo Gradient Boosting. Se modificaron los parámetros n_estimators y learning_rate para el doble del valor utilizado en el clasificador.

El mejor resultado para n_estimators = 200 y learning_rate = 0.2 es una precisión de 0.5777 y auc_score = 0.6540.

```
Resultados del clasificador con n_estimators = 200 y learning_rate = 0.2
Precisión del clasificador Gradient Boosting modificado: 0.5777
Roc AUC score modificado : 0.6540
Resultados del clasificador con n_estimators = 200 y learning_rate = 0.05
Precisión del clasificador Gradient Boosting modificado: 0.5759
Roc AUC score modificado : 0.6519
Resultados del clasificador con n_estimators = 50 y learning_rate = 0.2
Precisión del clasificador Gradient Boosting modificado: 0.5765
Roc AUC score modificado : 0.6518
Resultados del clasificador con n_estimators = 50 y learning_rate = 0.05
Precisión del clasificador Gradient Boosting modificado: 0.5724
Roc AUC score modificado : 0.6430
```

Ilustración 25. Resultados hiperparámetros para Gradient Boosting.

En la siguiente tabla se recoge los mejores resultados de todas las modificaciones a los hiperparámetros.

Modelo			Accuracy	AUC
Linear Model		Sin modificación	0.5690	0.6347
		Con modificación	0.5691	0.6367
AdaBoost		Sin modificación	0.5720	0.6373
	Con primera modificación		0.5760	0.6412
		Con segunda modificación	0.5746	0.6438
Gradient Boosting	Sin modificación		0.5549	0.6073
	Con modificación		0.5777	0.6540

Se puede observar que el modelo con mejores resultados de rendimiento y de valor de la curva ROC, tras hacerle las modificaciones pertinentes es el 'Gradient Boosting' para el que se obtiene una precisión de 0.5777 y AUC de 0.6540. Este hecho significa que en el 60% de los casos el modelo realizará las predicciones precisas. Un AUC de 0.6540 sugiere que el modelo tiene una capacidad moderada para distinguir entre las clases positivas y negativas, este modelo es mejor que una predicción aleatoria que es la línea base de la que se parte por el modelo Dummy Classifier, aunque todavía queda margen de mejora para su capacidad de discriminación.

Oversampling y undersampling

Para equilibrar el número de instancias para cada clase se pueden usar métodos denominados de *undersampling* u *oversampling* para mejorar el aprendizaje de los modelos. Prestar atención a que se aplique únicamente al conjunto de entrenamiento y NO al total de datos (entrenamiento y test), ya que el conjunto de test tiene que ser desconocido mientras se entrena el sistema.

En un inicio se utilizó este código para realizar el undersampling y oversampling, pero no se va a comentar ya que nos dimos cuenta de que se estaba realizando de manera errónea, ya que los datos de entrenamiento y test estaban siendo los mismos con lo que se aplicaban estas técnicas sin diferencias. El resultado eran estimaciones muy altas pero que no eran reales.

```python
# Define the oversampler and undersampler
oversampler = RandomOverSampler()
undersampler = RandomUnderSampler()

# Initialize lists to store results
columns_names = list(classifiers.keys())
rows_data = []

# Function to evaluate model with oversampling or undersampling
def evaluate_model_with_sampling(model, data_train, target_train, sampler):
    data_resampled, target_resampled = sampler.fit_resample(data_train, target_train)
    accuracy_scores = cross_val_score(model, data_resampled, target_resampled, cv=5, scoring='accuracy')
    target_predicted_proba = cross_val_predict(model, data_resampled, target_resampled, cv=5, method='predict_proba')
    target_binarized = label_binarize(target_resampled, classes=np.unique(target_resampled))
    auc_scores = roc_auc_score(target_binarized, target_predicted_proba, average='macro')
    return accuracy_scores, auc_scores

# Evaluate models with oversampling and undersampling
for name, model in classifiers.items():
    accuracy_scores_over, auc_scores_over = evaluate_model_with_sampling(model, data_train, target_train, oversampler)
    accuracy_scores_under, auc_scores_under = evaluate_model_with_sampling(model, data_train, target_train, undersampler)

    row_data_over = [accuracy_scores_over.mean(), auc_scores_over.mean()]  # Scores with oversampling
    row_data_under = [accuracy_scores_under.mean(), auc_scores_under.mean()]  # Scores with undersampling

    rows_data.append(row_data_over + row_data_under)  # Append both sets of scores

    print(f"Model: {name}")
    print(f"Mean Accuracy (Oversampling): {accuracy_scores_over.mean():.4f}")
    print(f"Mean AUC (Oversampling): {auc_scores_over.mean():.4f}")
    print(f"Mean Accuracy (Undersampling): {accuracy_scores_under.mean():.4f}")
    print(f"Mean AUC (Undersampling): {auc_scores_under.mean():.4f}")
    print()

# Create the DataFrame of cross-validation results
df_cross_validation = pd.DataFrame(rows_data, columns=["Mean Accuracy (Oversampling)", "Mean AUC (Oversampling)",
                                                       "Mean Accuracy (Undersampling)", "Mean AUC (Undersampling)"],
                                   index=columns_names)

print(df_cross_validation)
```

Ilustración 26. Código Fuente Over Sampling y Under Sampling

Por lo que se abordó de otra manera, en este caso el código evalúa múltiples modelos de clasificación, utilizando *oversampling* y *undersampling*, para abordar el desbalance de clases en un conjunto de datos. Itera sobre un conjunto de modelos, cada uno asociado a un nombre específico. Para cada modelo, aplica *oversampling* al conjunto de entrenamiento con *RandomOverSampler* para aumentar la representación de las clases minoritarias y equilibrar el conjunto de datos. Luego, entrena el modelo con el conjunto de entrenamiento aumentado y realiza predicciones en el conjunto de prueba. Calcula precisión, precisión ponderada y Área bajo la Curva ROC (AUC) utilizando los datos de prueba y funciones de puntuación correspondientes. Posteriormente, repite el proceso aplicando *undersampling* al conjunto de entrenamiento con *RandomUnderSampler* para reducir la representación de las clases mayoritarias. Entrena nuevamente el modelo con el conjunto de entrenamiento reducido y calcula las mismas métricas de evaluación. Finalmente, muestra los resultados de precisión, precisión ponderada y AUC para ambos métodos de muestreo para cada modelo de clasificación.

```
# Iterate over classifiers
for name, clf in classifiers.items():
    print(f"Classifier: {name}")

    # Apply oversampling
    ros = RandomOverSampler()
    X_train_resampled, y_train_resampled = ros.fit_resample(data_train, target_train)

    # Train the classifier with oversampled data
    clf.fit(X_train_resampled, y_train_resampled)

    # Make predictions with oversampled data
    y_pred_oversampled = clf.predict(data_test)

    # Calculate accuracy with oversampled data
    accuracy_oversampled = accuracy_score(target_test, y_pred_oversampled)
    print("Results with Oversampling:")
    print(f"Accuracy: {accuracy_oversampled:.4f}")

    # Calculate precision with oversampled data
    precision_oversampled = precision_score(target_test, y_pred_oversampled, average='weighted')
    print(f"Precision: {precision_oversampled:.4f}")

    # Calculate AUC with oversampled data
    y_test_bin = label_binarize(target_test, classes=clf.classes_)
    if len(clf.classes_) == 2:
        auc_oversampled = roc_auc_score(y_test_bin, clf.predict_proba(data_test)[:, 1])
    else:
        auc_oversampled = roc_auc_score(y_test_bin, clf.predict_proba(data_test), average='macro', multi_class='ovr')
    print(f"AUC: {auc_oversampled:.4f}")
    print()

    # Apply undersampling
    rus = RandomUnderSampler()
    X_train_resampled, y_train_resampled = rus.fit_resample(data_train, target_train)

    # Train the classifier with undersampled data
    clf.fit(X_train_resampled, y_train_resampled)

    # Make predictions with undersampled data
    y_pred_undersampled = clf.predict(data_test)

    # Calculate accuracy with undersampled data
    accuracy_undersampled = accuracy_score(target_test, y_pred_undersampled)
    print("Results with Undersampling:")
    print(f"Accuracy: {accuracy_undersampled:.4f}")

    # Calculate precision with undersampled data
    precision_undersampled = precision_score(target_test, y_pred_undersampled, average='weighted')
    print(f"Precision: {precision_undersampled:.4f}")

    # Calculate AUC with undersampled data
    if len(clf.classes_) == 2:
        auc_undersampled = roc_auc_score(y_test_bin, clf.predict_proba(data_test)[:, 1])
    else:
        auc_undersampled = roc_auc_score(y_test_bin, clf.predict_proba(data_test), average='macro', multi_class='ovr')
    print(f"AUC: {auc_undersampled:.4f}")
    print()
```

Conclusión del nivel básico

En síntesis, aquí finaliza el nivel básico de evaluación en el que se han adecuado los datos de entrada para poder trabajar con ellos, se han codificado las variables necesarias, se han eliminado los datos irrelevantes para el problema y también aquellos que eran inconsistentes y no contribuían con información de valor. Se ha establecido una linea base del problema con un clasificador base Dummy Classifier. A la vista de sus resultados se parte de una precisión cuyo valor medio representa la aleatoriedad, por lo que los resultados no eran fiables. A partir de este hecho se aplicaron otros dos clasificadores distintos Decission Tree y Linear Model

con el objetivo de mejorar la precisión del modelo y la capacidad de discriminación, los resultados reflejan una pequeña mejora, aunque no muy significativa en la precisión del modelo para realizar predicciones sobre los datos de prueba gracias al entrenamiento con el conjunto de datos de entramiento. Observando los resultados presentados en la tabla de los resultados finales de los clasificadores Decission Tree es el clasificador con capacidad de discriminación más baja y sin embargo Linear Model es el clasificador con el tercer mejor resultado para el valor de AUC, es el modelo que ofrece en tercer lugar la mejor capacidad de discriminación para los datos.

Conociendo los valores de la predicción y AUC de los modelos se aplica la validación cruzada se aplica para evaluar también el rendimiento de un modelo de aprendizaje, pero de un forma más robusta y confiable. Se evita con la utilización de esta técnica el sobreajuste del modelo al conjunto de datos. Al hacer una división con un objeto de tipo 'Stratified KFold' se divide el conjunto en 10 pliegues y se evalua con una función específica implementada por nosotros el modelo en cada pliegue. Se obtiene así una estimación mucho más estable del rendimiento general de cada modelo. Después se aplica el Wilcoxon signed-rank test, se aplica para comparar dos muestras relacionadas y determinar si la distribución de diferencias es simétrica, en otras palabas, se utiliza para comparar rendimiento de dos modelos y determinar si hay una diferencia significativa entre ellos.

Conociendo la precisión de cada modelo y su capacidad de discriminación se busca todavía mejorar los más. Esto es posible debido a que cada modelo esta caracterizado por sus hiperparámetros, cuanto más adecuados sea los hiperparámetros teniendo en cuenta las características del modelo y del conjunto de datos mejor serán los resultados de los modelos. Para ello se modifica de forma manual buscando mejorar los resultados.

Los modelos de aprendizaje multiclase como son los de la base de datos tienen tendencia a sesgos por la gran disparidad en el número de instancias entre las clases. Los modelos se sesgan hacia la clase dominante e ignoran la minoritaria. Con el objetivo de evitar esto y equilibrar las instancias se aumenta el número de instancia en la clase minoritaria para equilibrarlas con la clase dominante, oversampling, y se reduce el número de instancia en la clase dominante para igualar el numero de la clase minoritaria, undersampling.

Nivel Medio

Selección automática de atributos

Probar algoritmos de selección automática de atributos (feature selection) relevantes según la variable a clasificar;

En este punto se han probado seis algoritmos basados en diferentes estrategias para intentar mejorar los resultados

Selección de características basada en modelos: Algunos algoritmos de aprendizaje automático, como los árboles de decisión y las máquinas de vectores de soporte (SVM), proporcionan importancias de características incorporadas que se pueden utilizar para seleccionar las características más relevantes. En este caso se ha utilizado un árbol de decisión para después con los valores que se obtienen de él transformar los datos y entrenar con ellos los diferentes modelos.

```
# Entrenar un árbol de decisión para obtener importancias de características
tree_classifier = DecisionTreeClassifier()
tree_classifier.fit(data_train, target_train)

# Crear un selector de características basado en la importancia del árbol de decisión
selector_tree = SelectFromModel(tree_classifier, prefit=True)

# Transformar los datos de entrenamiento y prueba
data_train_selected_tree = selector_tree.transform(data_train)
data_test_selected_tree = selector_tree.transform(data_test)

# Entrenar el modelo con las características seleccionadas mediante árboles de decisión
selected_classifiers_tree = {name: clone(model).fit(data_train_selected_tree, target_train) for name, model in classifiers.items()}
```

Ilustración 27. Código Fuente para la selección basada en modelos

Selección de características mediante regulación: Algunos modelos, como la regresión logística con regularización L1 (Lasso), penalizan automáticamente las características menos importantes, lo que puede llevar a la selección automática de características.

```
# Entrenar un modelo de regresión logística con regularización L1 (Lasso)
logistic_regression = LogisticRegression(penalty='l1', solver='liblinear')
logistic_regression.fit(data_train, target_train)

# Crear un selector de características basado en los coeficientes del modelo
selector_lasso = SelectFromModel(logistic_regression, prefit=True)

# Transformar los datos de entrenamiento y prueba
data_train_selected_lasso = selector_lasso.transform(data_train)
data_test_selected_lasso = selector_lasso.transform(data_test)

# Entrenar el modelo con las características seleccionadas mediante regularización L1
selected_classifiers_lasso = {name: clone(model).fit(data_train_selected_lasso, target_train) for name, model in classifiers.items()}
```

Ilustración 28. Código Fuente para la selección mediante regulación

Selección de características utilizando técnicas de embbeding: Estas técnicas aprenden automáticamente las características más relevantes durante el entrenamiento del modelo. Ejemplos incluyen métodos de reducción de dimensionalidad como PCA (Análisis de Componentes Principales) y métodos de selección de características basados en modelos como L1 regularization en modelos lineales.

Primero, se instancia un objeto PCA con n_components=10, lo que significa que se conservarán solo las 10 mejores componentes principales. Luego, se ajusta este modelo a los datos de entrenamiento (data_train) y se transforman tanto los datos de entrenamiento como los de prueba utilizando estas nuevas características

seleccionadas (data_train_selected_pca y data_test_selected_pca, respectivamente).

Después, se utiliza un conjunto de modelos de aprendizaje automático (definidos previamente en el diccionario *classifiers*) para entrenarlos utilizando las características seleccionadas mediante PCA. Cada modelo se clona y se entrena con los datos transformados (data_train_selected_pca).

```
# Aplicar PCA para reducción de dimensionalidad y selección de características
pca = PCA(n_components=10)  # Selecciona las 10 mejores componentes principales
data_train_selected_pca = pca.fit_transform(data_train)
data_test_selected_pca = pca.transform(data_test)

# Entrenar el modelo con las características seleccionadas mediante PCA
selected_classifiers_pca = {name: clone(model).fit(data_train_selected_pca, target_train) for name, model in classifiers.items()}
```

Ilustración 29. Código Fuente para la selección mediante PCA

Selección de características mediante técnicas de empaquetado: Estas técnicas evalúan diferentes subconjuntos de características utilizando un algoritmo de aprendizaje automático y seleccionan aquellos subconjuntos que producen los mejores resultados. Ejemplos incluyen la eliminación recursiva de características (RFE) y los algoritmos genéticos, en este caso se ha utilizado RFE.

Primero, se crea un modelo de regresión logística con ciertos hiperparámetros (*max_iter*=5000 y *penalty*=None). Luego, se utiliza este modelo como estimador en RFE para seleccionar las 10 mejores características.

Una vez que se ajusta RFE a los datos de entrenamiento (*data_train y target_train*), se identifican las características seleccionadas y se extraen de los conjuntos de datos de entrenamiento y prueba (*data_train_selected y data_test_selected, respectivamente*).

Finalmente, se entrenan varios modelos de aprendizaje automático (definidos en el diccionario classifiers) utilizando las características seleccionadas.

```
logistic_regression = LogisticRegression(max_iter=5000, penalty=None)
rfe = RFE(estimator=logistic_regression, n_features_to_select=10)  # Selecciona las 10 mejores características
rfe.fit(data_train, target_train)
selected_features = rfe.support_
data_train_selected = data_train.iloc[:, selected_features]
data_test_selected = data_test.iloc[:, selected_features]
```

Ilustración 30. Código Fuente para la selección mediante empaquetado

Selección de características utilizando métodos de agrupamiento: Estos métodos agrupan características similares y seleccionan un representante de cada grupo como características finales.

Para ello se aplica KMeans para agrupar las características en clusters. Luego, calcula la distancia entre cada característica y los centros de los clusters. Selecciona las 10 características más cercanas a cada centro de Cluster y las utiliza para crear conjuntos de datos de entrenamiento y prueba reducidos.

Finalmente, entrena modelos de aprendizaje automático con estas características seleccionadas utilizando una variedad de clasificadores definidos previamente.

```
# Aplicar KMeans para agrupar características
kmeans = KMeans(n_clusters=37)  # Selecciona 10 clusters
kmeans.fit(data_train.T)  # Transponemos los datos para que cada muestra sea una característica

# Obtener los centros de los clusters
cluster_centers = kmeans.cluster_centers_
# Ajustar la forma de los datos de entrenamiento y los centros de los clusters
data_train_reshaped = data_train.values.reshape(data_train.shape[0], 1, data_train.shape[1])
cluster_centers_reshaped = cluster_centers[:, None, :]

# Transponer el primer eje de los datos de entrenamiento
data_train_reshaped = np.transpose(data_train_reshaped, (2, 1, 0))

# Calcular la distancia entre los datos de entrenamiento y los centros de los clusters
distances = np.linalg.norm(data_train_reshaped - cluster_centers_reshaped, axis=2)

# Obtener los índices de las características más cercanas a los centros de los clusters
sorted_indices = np.argsort(distances)

# Seleccionar las características más cercanas a los centros de los clusters
selected_indices = sorted_indices[:, :10]

# Seleccionar las características en los datos de entrenamiento y prueba
data_train_selected_kmeans = data_train.iloc[:, selected_indices.flatten()]
data_test_selected_kmeans = data_test.iloc[:, selected_indices.flatten()]


# Entrenar y evaluar los modelos con las características seleccionadas mediante KMeans...
selected_classifiers_kmeans = {name: clone(model).fit(data_train_selected_kmeans, target_train) for name, model in classifiers.items()}
```

Selección de características basada en la información: La información mutua entre dos variables mide la dependencia entre ellas y puede ser utilizada para seleccionar características relevantes para la variable objetivo.

Primero, se utiliza SelectKBest con la función de puntuación de información mutua (mutual_info_classif) para seleccionar estas características tanto en los datos de entrenamiento como en los de prueba. Luego, entrena modelos de aprendizaje automático utilizando las características seleccionadas mediante información mutua.

```
# Aplicar la información mutua para seleccionar características
selector_mi = SelectKBest(score_func=mutual_info_classif, k=10)  # Selecciona las 10 mejores características
data_train_mi = selector_mi.fit_transform(data_train, target_train)
data_test_mi = selector_mi.transform(data_test)

# Entrenar los modelos con las características seleccionadas mediante información mutua
selected_classifiers_mi = {name: clone(model).fit(data_train_mi, target_train) for name, model in classifiers.items()}
```

A pesar de haber realizado todas estas pruebas en ninguna de ellas se observó una mejora significativa ni de la presión ni de la AUC tras el entrenamiento y la validación cruzada por lo que se abordaron otras líneas de trabajo.

Aplicar al menos un total de 5 clasificadores distintos al base;

Además de los tres clasificadores previamente mencionados hemos desarrollado otros tres: Naive Bayes, Random Forest y AdaBoost.

Para el clasificador Naive Bayes se crea una instancia de la clase GaussianNB(), asumiendo que las características siguen una distribución gaussiana normal e independiente. A continuación, se entrena el clasificador con los datos de entrenamiento y se calcula la precisión comparando las predicciones con el

conjunto de prueba. Se calcula el área bajo la curva ROC para cada clase frente al resto utilizando las probabilidades predichas.

```
# Inicializar el clasificador Naive Bayes
nb_classifier = GaussianNB()

# Entrenar el clasificador
nb_classifier.fit(data_train, target_train)

# Predecir en el conjunto de prueba
y_pred = nb_classifier.predict(data_test)

# Calcular la precisión del clasificador
accuracy = accuracy_score(target_test, y_pred)
print("Accuracy:", accuracy)

# Mostrar el reporte de clasificación
print("Classification Report:")
print(classification_report(target_test, y_pred))

# Predecir las probabilidades en el conjunto de prueba
y_proba = nb_classifier.predict_proba(data_test)

# Calcular el AUC para cada clase versus el resto
auc_scores = roc_auc_score(target_test, y_proba, multi_class='ovr')

# Imprimir los AUC scores para cada clase
print("AUC scores")
print(auc_scores)
```

Ilustración 31. Clasificador Naive Bayes.

Este clasificador obtiene una precisión de 0.116 y un valor de la curva ROC de 0.5811. Este modelo tiene una alta capacidad de discriminación considerando las de los otros modelos, pero no preciso, su precisión es menor que la de un modelo aleatorio por lo que sus resultados no son fiables.

```
Accuracy: 0.11609511643902919
Classification Report:
              precision    recall  f1-score   support

         <30       0.11      0.99      0.20      2285
         >30       0.53      0.01      0.02      7117
          NO       0.64      0.00      0.00     10952

    accuracy                           0.12     20354
   macro avg       0.43      0.34      0.08     20354
weighted avg       0.54      0.12      0.03     20354

AUC scores
0.5810355461514009
```

Ilustración 32. Resultados Naive Bayes.

El segundo clasificador que forma parte de esta lista es el Random Forest. Random Forest se basa en la construcción de múltiples árboles de decisión durante el

proceso de entrenamiento. Cada árbol de decisión se entrena con una muestra aleatoria de los datos y una selección aleatoria de características.

```
# Inicializar el clasificador Random Forest
rf_classifier = RandomForestClassifier(random_state=42)

# Entrenar el clasificador
rf_classifier.fit(data_train, target_train)

# Predecir en el conjunto de prueba
y_proba_rf = rf_classifier.predict_proba(data_test)
```

Ilustración 33. Random Forest

```
auc_scores = roc_auc_score(target_test, y_proba_rf, multi_class='ovr')
print(auc_scores)
```

Ilustración 34. AUC Random Forest

La precisión del Random Forest es 0.5649 y el valor de AUC es de 0.6285.

Por último, se va a presentar el último modelo utilizado, el modelo AdaBoost. AdaBoost pertenece a la categoría de algoritmos de conjunto, que combinan múltiples modelos de aprendizaje para mejorar la precisión predictiva. En lugar de depender de un solo modelo, AdaBoost construye una secuencia de modelos de aprendizaje débil, donde cada modelo sucesivo se enfoca más en los casos clasificados incorrectamente por los modelos anteriores.

Se realizan las particiones del conjunto de datos, se entrena el modelo, se realizan las predicciones pertinentes, y se obtienen los pesos de los estimadores que utilizan internamente el clasificador AdaBoost.

Se calcula la precisión del modelo AdaBoost y el valor de AUC, respectivamente los valores son 0.57207 y 0.63739, estos valores son de los más altos que los modelos han obtenido.

```
# División de datos de entrenamiento y prueba
data_train, data_test, target_train, target_test = train_test_split(data, target, test_size=0.2, random_state=42)

# Inicializar el clasificador AdaBoost
adaboost_classifier = AdaBoostClassifier()
print("Los hiperparametros que utiliza este modelo son:")
print("base_estimator: ",adaboost_classifier.base_estimator)
print("n_estimators: ",adaboost_classifier.n_estimators)
print("learning_rate: ",adaboost_classifier.learning_rate)
# Entrenar el clasificador
adaboost_classifier.fit(data_train, target_train)

# Obtener los pesos de los estimadores base utilizados internamente por el clasificador AdaBoost
base_estimator_weights = np.array(adaboost_classifier.estimator_weights_)

print("Pesos de los estimadores base para AdaBoost:")
print(base_estimator_weights)

# Predecir en el conjunto de prueba
y_pred = adaboost_classifier.predict(data_test)

# Calcular la precisión del clasificador
accuracy = accuracy_score(target_test, y_pred)
print("Precisión del modelo AdaBoost:", accuracy)

predictions_auc = adaboost_classifier.predict_proba(data_test)
auc_scores_adabost = roc_auc_score(target_test, predictions_auc, multi_class='ovr')

# Imprime los AUC scores para cada clase
print("AUC scores for each class (AdaBoost):", auc_scores_adabost)
```

Finalmente se muestra en la siguiente tabla los resultados de la precisión y el valor de la curva ROC para cada modelo.

Modelo	Precisión del modelo	Valor AUC x clase
Linear Model	0.5690	0.6367
AdaBoost	0.5720	0.6373
Naive Bayes	0.1160	0.5810
Random Fores	0.5649	0.6285
Decision Tree	0.4541	0.5255
Gradient Boosting	0.5758	0.6519

Tabla 1. Resultados finales de la precisión y el valor AUC de los modelos.

Hiperparametros con GridSearchCV

Estudiar la modificación de sus hiperparámetros por defecto de los clasificadores usados para mejorar los resultados (Python/scikit-learn tiene funciones específicas como GridSearchCV para realizar pruebas de hiperparámetros);

Los clasificadores mostrados anteriormente resultan en un valor para la precisión de cada modelo y para el valor de AUC, que representa la capacidad de discriminación de cada modelo. Los modelos estan caracterizados por sus hiperparámetros, cuanto más adecuados sean según las características del conjunto de datos, más preciso será el modelo y mejor capacidad de discriminación tendrá. Con el objetivo de conocer cuáles son los mejores hiperparámetros para cada modelo se utiliza Grid Search.

Esta técnica es una técnica de optimización de hiperparámetros utilizada para encontrar la combinación de valores de hiperparámetros de un modelo de aprendizaje automático. Realiza para cumplir con su objetivo una búsqueda exhaustiva en todas las combinaciones posibles de valores de hiperparámetros dentro del espacio definido. Cada combinación se evalúa por validación cruzada para conocer su rendimiento en los datos no vistos. En concreto se divide el conjunto de datos en varios pliegues, en este código en concreto se divide en 5 folds y se utiliza uno como conjunto de validación y el resto se utiliza como conjunto de entrenamiento. Una vez que se encuentra la mejor combinación de hiperparámetros, Grid Search devuelve el modelo entrenado con esa combinación de hiperparámetros.

```python
from sklearn.model_selection import GridSearchCV

# Definimos los diccionarios de parámetros para cada modelo
param_grid = {
    "Logistic Regression": {'penalty': ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]},
    "AdaBoost": {'n_estimators': [50, 100, 200], 'learning_rate': [0.001, 0.01, 0.1, 1]},
    "Random Forest": {'n_estimators': [50, 100, 200], 'max_depth': [None, 10, 20, 30]},
    "Decision Tree": {'max_depth': [None, 10, 20, 30], 'min_samples_split': [2, 5, 10]},
    "Gradient Boosting": {'n_estimators': [50, 100, 200], 'learning_rate': [0.001, 0.01, 0.1, 1], 'max_depth': [3, 5, 7]},
    "KNN": {'n_neighbors': [3, 5, 7, 9], 'weights': ['uniform', 'distance']},
}

# Creamos un diccionario para almacenar los mejores parámetros de cada modelo
best_params = {}

# Iteramos sobre los modelos y realizamos la búsqueda de hiperparámetros
for name, model in classifiers.items():
    if name in param_grid:
        print(f"Searching best parameters for {name}...")
        grid_search = GridSearchCV(model, param_grid[name], cv=5, scoring='accuracy')
        grid_search.fit(data_train, target_train)
        best_params[name] = grid_search.best_params_
        print("Best parameters found:")
        print(grid_search.best_params_)
        print()

print("Best parameters for each model:")
print(best_params)
```

Ilustración 35. Estudio hecho con Grid Search.

Los mejores hiperparámetros para cada modelo se calculan iterando sobre cada modelo en el diccionario 'param_grid', este diccionario contiene los parámetros para cada modelo, la clave es el nombre del modelo y a su vez el contenido es otro diccionario cuyas claves son los parámetros conteniendo una lista de valores de la que se quiere extraer la mejor combinación.

El primer modelo, la Regresión Logística, necesita una combinación especifica de hiperparámetros, es por este motivo que, al ejecutar este código, con ciertas combinaciones salta un mensaje de *warning*, son combinaciones no permitas por el modelo. Teniendo esto en cuenta, los mejores hiperparámetros según Grid Search son {'C': 0.01, 'penalty': 'l2'}.

El mismo proceso se lleva a cabo para cada modelo, los mejores parámetros para el modelo AdaBoost son {'learning_rate': 1, 'n_estimators': 200}, para Random

Forest son {'*max_depth*': 10, 'n_estimators': 100}, para Decission Tree son {'*max_depth*': 10, '*min_samples_split*': 10} y por último para Gradient Boosting son:

```
Best parameters found:
{'C': 0.01, 'penalty': 'l2'}

Searching best parameters for AdaBoost...
Best parameters found:
{'learning_rate': 1, 'n_estimators': 200}

Searching best parameters for Random Forest...
Best parameters found:
{'max_depth': 10, 'n_estimators': 100}

Searching best parameters for Decision Tree...
Best parameters found:
{'max_depth': 10, 'min_samples_split': 10}
```

Ilustración 36. Best Parameters Found I

Se vuelven a cargar los modelos con los hiperparámetros que según GridSearchCV son los mejores, para los modelos con rendimiento más alto, y se obtienen los siguientes resultados.

	Hiperparámetros Iniciales		**Hiperparámetros según Grid Search**	
Modelo	Precisión del modelo	Valor AUC x clase	Precisión del modelo	Valor AUC x clase
Linear Model	0.5690	0.6367	0.5690	0.6367
AdaBoost	0.5720	0.6373	0.5734	0.6400
Gradient Boosting	0.5758	0.6519	0.5788	0.6399

Tabla 2. Comparación auc y accuracy según hiperparámetros

Aplicar el test estadístico sobre todos los resultados obtenidos. Al tener más clasificadores hay que comparar más resultados que en el nivel básico;

Se proporciona una comparación rápida del rendimiento de los siguientes modelos: Regresión Logística, AdaBoost y Gradient Boosting. Todos ellos se han utilizado con los mejores hiperparámetros obtenidos según GridSearchCV.

Se ha llevado a cabo incluyendo una partición de 10-KFold estratificada para la validación cruzada. Se ha utilizado la función definida anteriormente '*evaluate model*' para que devuelva las puntuaciones de precisión y AUC mediante validación cruzada. Por último, imprime el DataFrame obteniendo los siguientes resultados.

```
Model: Logistic Regression
Mean Accuracy: 0.5696
Mean AUC: 0.6364

Model: AdaBoost
Mean Accuracy: 0.5746
Mean AUC: 0.6400

Model: Gradient Boosting
Mean Accuracy: 0.5774
Mean AUC: 0.6496
```

	Mean Accuracy	Mean AUC
Logistic Regression	0.569560	0.636413
AdaBoost	0.574559	0.639978
Gradient Boosting	0.577372	0.649561

Ilustración 37. Resultados test estadístico de los principales clasificadores.

La capacidad de discriminación respecto a las capacidades anteriores con los hiperparámetros de base ha aumentado, ha habido un aumento de alrededor de un punto para cada valor. Por otro lado, la precisión de cada modelo aumenta, aunque en caso de la regresión logística es imperceptible, y en los otros clasificadores es de apeas medio punto.

Los resultados del test de Wilcoxon proporcionan información sobre la comparación entre los modelos de clasificación AdaBoost, Logistic Regression y Gradient Boosting en términos de su desempeño en la predicción de los datos.

Para la comparación entre AdaBoost y Logistic Regression, el test de Wilcoxon para Accuracy y AUC muestra que no hay una diferencia significativa entre estos dos modelos. Tanto para el Accuracy como para el AUC, el valor de Wilcox V es 1.0 y el p-value es 0.50. Esto indica que no hay evidencia suficiente para rechazar la hipótesis nula de que no hay diferencia en el desempeño entre AdaBoost y Logistic Regression en términos de Accuracy y AUC. En el caso de la comparación entre AdaBoost y Gradient Boosting, así como entre Gradient Boosting y Logistic Regression, los resultados son consistentes con los obtenidos en la comparación anterior. Para ambas comparaciones, tanto para Accuracy como para AUC, el valor de Wilcox V es 0.0 y el p-value es 1.00. Esto sugiere que no hay una diferencia significativa en el desempeño entre los modelos evaluados en ninguna de las métricas consideradas.

En resumen, según los resultados del test de Wilcoxon, no se observa una diferencia estadísticamente significativa en el desempeño entre AdaBoost, Logistic Regression y Gradient Boosting en términos de Accuracy y AUC en los datos evaluados.

Clasificadores sensibles al coste

Estudiar el uso clasificador sensibles al coste (scikit-learn se puede usar el parámetro *class_weight* de la mayoría de los clasificadores) y razonar sus resultados;

Este estudio se ha realizado para los clasificadores

Random Forest

Para su correcta interpretación hay que tener en cuenta que cada fila muestra una característica junto con su importancia, medida como el peso asignado por el modelo.

A partir de los datos obtenidos podemos ver que las características numéricas como *time_in_hospital, num_lab_procedures, num_medications, y number_diagnoses* tienen importancias relativamente altas. Esto sugiere que la duración de la hospitalización, el número de procedimientos de laboratorio, el número de medicamentos recetados y el número de diagnósticos tienen un impacto significativo en las predicciones del modelo.

Pesos de las características:

	Feature	Importance
0	time_in_hospital	8.094796e-02
1	num_lab_procedures	1.311633e-01
2	num_procedures	4.812151e-02
3	num_medications	1.125908e-01
4	number_outpatient	2.148963e-02
5	number_emergency	1.669865e-02
6	number_inpatient	4.643993e-02
7	number_diagnoses	5.057978e-02
8	race	3.103350e-02
9	gender	2.542141e-02
10	age	6.502537e-02
11	diag_1	6.964788e-02
12	diag_2	7.716934e-02
13	diag_3	7.897082e-02
14	metformin	1.788419e-02
15	repaglinide	4.010486e-03
16	nateglinide	2.132066e-03
17	chlorpropamide	3.969850e-04
18	glimepiride	9.464126e-03
19	acetohexamide	9.038761e-06
20	glipizide	1.610392e-02
21	glyburide	1.491833e-02
22	tolbutamide	8.512965e-05
23	pioglitazone	1.093198e-02
24	rosiglitazone	9.815236e-03
25	acarbose	9.165905e-04
26	miglitol	1.395688e-04
27	troglitazone	1.313936e-05
28	tolazamide	1.654639e-04
29	insulin	3.279444e-02
30	glyburide-metformin	2.077809e-03
31	glipizide-metformin	5.492742e-05
32	glimepiride-pioglitazone	5.350137e-06
33	metformin-rosiglitazone	6.934050e-07
34	metformin-pioglitazone	2.160280e-07
35	change	1.459801e-02
36	diabetesMed	8.182432e-03

Ilustración 38. Resultados weights según Random Forest

En cuanto a las características categóricas, como los diferentes diagnósticos *(diag_1, diag_2, diag_3*) y los medicamentos prescritos (*metformin, glimepiride, insulin,* etc.), tienen importancias significativas. Esto indica que los diagnósticos específicos y los medicamentos recetados también son predictores importantes para el modelo.

Sin embargo, hay características como *tolbutamide, troglitazone y metformin-rosiglitazone,* que tienen importancias muy bajas. Esto sugiere que estas características tienen poco o ningún impacto discernible en las predicciones del modelo. Es posible que estas características tengan valores poco comunes o estén poco correlacionadas con los resultados de interés en el conjunto de datos.

Regresión logística

En este caso los resultados hay que interpretarlos de otra forma:

- Positivos: Un peso positivo indica que a medida que el valor de esa característica aumenta, también aumenta la probabilidad de readmisión. Por ejemplo, "*number_inpatient*" tiene un peso positivo relativamente alto (0.251984), lo que sugiere que un mayor número de ingresos hospitalarios previos aumenta la probabilidad de readmisión.
- Negativos: Un peso negativo indica que a medida que el valor de esa característica aumenta, disminuye la probabilidad de readmisión. Por ejemplo, "*troglitazone*" tiene un peso negativo (-0.035585), lo que sugiere que el uso de este medicamento está asociado con una menor probabilidad de readmisión.
- Cercanos a cero: Los pesos cercanos a cero indican que la característica no tiene un impacto significativo en la predicción.

Por ejemplo, los ingresos hospitalarios previos parecen ser un factor importante para predecir la readmisión, mientras que algunos medicamentos tienen poco impacto en la predicción.

```
Pesos de las características:
time_in_hospital            0.041602
num_lab_procedures         -0.002608
num_procedures             -0.024513
num_medications             0.025516
number_outpatient           0.000035
number_emergency            0.062322
number_inpatient            0.251984
number_diagnoses            0.041873
race                        0.000146
gender                      0.007373
age                         0.038728
diag_1                     -0.012693
diag_2                      0.003868
diag_3                      0.008618
metformin                  -0.034297
repaglinide                -0.000600
nateglinide                 0.002563
chlorpropamide             -0.006697
glimepiride                -0.019759
acetohexamide              -0.013085
glipizide                  -0.003300
glyburide                  -0.007283
tolbutamide                -0.008517
pioglitazone               -0.011015
rosiglitazone              -0.013507
acarbose                   -0.012730
miglitol                   -0.006244
troglitazone               -0.035585
tolazamide                  0.001835
insulin                    -0.021595
glyburide-metformin        -0.015211
glipizide-metformin        -0.079107
glimepiride-pioglitazone   -0.013834
metformin-rosiglitazone    -0.012877
metformin-pioglitazone     -0.011736
change                     -0.017744
diabetesMed                 0.051044
dtype: float64
```

Ilustración 39. Resultados weights según regresión logística

AdaBoost

Si todos los pesos de los estimadores bases para AdaBoost son iguales a 1, significa que cada estimador base contribuye de manera uniforme con el mismo peso a la predicción final del modelo AdaBoost. Es decir, cada estimador base tiene la misma influencia en la toma de decisiones del modelo conjunto.

Este hecho puede ser a causa de que los estimadores bases estan teniendo un desempeño similar en términos de su capacidad para mejorar la predicción del modelo. Cada estimador base tiene la misma importancia y contribuye de manera equitativa a la predicción final del modelo.

```
# Obtener los pesos de los estimadores base utilizados internamente por el clasificador AdaBoost
base_estimator_weights = np.array(adaboost_classifier.estimator_weights_)

print("Pesos de los estimadores base para AdaBoost:")
print(base_estimator_weights)
```

```
Pesos de los estimadores base para AdaBoost:
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1.]
```

Ilustración 40. Resultados weights para AdaBoost

Conclusión nivel medio

En primer lugar, cabe destacar el fallo en la linea de la búsqueda de mejorar los resultados mediante la selección automática de atributos. Este fallo puede surgir por diferentes motivos y por ello se han elaborado diferentes estrategias. En definitiva, podemos decir que no es un error en la elección del método o en la elección de los atributos, sino que puede estar relacionado con el conjunto de datos. Nuestra hipótesis es que este resultado es debido al ruido que existe en los datos, esto hace que los algoritmos no trabajen de manera adecuada. Esta hipótesis también justifica los resultados de las técnicas de oversampling y undersampling que se han utilizado previamente.

Además de los clasificadores aplicados en el nivel básico, se han utilizado el Naive Bayes, AdaBoost y Random Forest con el objetivo de tener predicciones más precisas que los anteriores. El Random Forest y AdaBoost destacan sobre el primer algoritmo resultando con una precisión y capacidad de discriminación mejorando casi en un 10% los anteriores. La fiabilidad de los resultados al entrenar el modelo con estos clasificadores concluye en predicciones más exactas sobre el problema.

Estos hechos se han conseguido con los hiperparámetros estándar al instanciar cada modelo. Cuanto más adecuados sean los hiperparámetros definidos al conjunto de datos que se utiliza en el problema más se ajustará el análisis. Con el objetivo de conocer los mejores se han analizado con GridSearchCV y se ha obtenido los nuevos resultados de los clasificadores utilizando estos. Además, para corrobar que los resultados eran mejores que con los hiperparámetros definidos por defecto se vuelto a hacer el test estadístico de la validación cruzada utilizando 10-KFold cross validation.

El último de los puntos en este apartado ha sido el estudio de los clasificadores sensibles al coste. Tras su análisis hemos podido comprobar que hay características que son mucho más relevantes para los predictores que otras, esta idea se desarrollará con más profundidad en el siguiente apartado.

Nivel avanzado

Gradient Boosting

Una vez que se ha visto que el predictor Gradient Boosting es el que mejor resultados nos aporta se ha intentado mejorar aun más estos resultados, aplicando para ello diferentes técnicas.

En primer lugar, se han unido el modelo Gradient Boosting con el modelo adaboost para después hacer una predicción con un metamodelo a partir de las nuevas características. El código entrena dos modelos de clasificación, AdaBoost y Gradient Boosting, usando datos de entrenamiento y realiza predicciones mediante validación cruzada. Luego, construye un conjunto de entrenamiento combinando las predicciones de ambos modelos. Un clasificador Random Forest se entrena con este conjunto. Se promedian las predicciones de los modelos base para generar predicciones de ensemble sobre datos de prueba. Calcula el área bajo la curva (AUC) de la curva ROC para cada clase y luego promedia estos valores para evaluar el rendimiento general del ensemble.

Gracias a este algoritmo hemos conseguido una AUC de 0.6485 mejor que los modelos por separado.

Por otro lado se ha utilizado smote, una técnica de sobre-muestreo que se utiliza comúnmente en problemas de desequilibrio de clases en conjuntos de datos de aprendizaje automático. En los conjuntos de datos desequilibrados, donde una clase es significativamente menos frecuente que otra, los modelos de aprendizaje automático pueden sesgarse hacia la clase mayoritaria y mostrar un rendimiento deficiente en la predicción de la clase minoritaria.En definitiva se podría decir que es muy similar al oversampling y no hemos conseguido mejorar el AUC.

La última técnica ha sido implementar técnicas de regularización en Gradient Boosting, podemos utilizar parámetros como max_depth para limitar la profundidad de los árboles base y learning_rate para reducir la contribución de cada árbol. Los parámetros utilizados han sido max_depth=6 y learning_rate=0.1

SHAP y LIME

Aplicar razonamientos o técnicas relacionadas con la práctica no estudiadas directamente en prácticas pero útiles para el estudio del conjunto de datos.

Por ejemplo, existen unas librerías en Python que ayudan a explicar los resultados propuestos por el sistema predictivo (clasificador):

- **LIME (Local Interpretable Model-Agnostic Explanations) y**
- **SHAP (SHapley Additive exPlanations).**

En primer lugar, se ha trabajado con la librería SHAP. Para ello Se ha creado un objeto explainer con el modelo y los datos de entrenamiento, luego se calculan los valores SHAP para el conjunto de prueba. Finalmente, se visualiza un resumen de los valores SHAP con shap.summary_plot(), que muestra la importancia relativa de cada característica en las predicciones del modelo. Gracias a esta librería podemos ver las características más representativas y sus relaciones entre ellas.

```
# Crear un objeto explainer de SHAP con el modelo entrenado
explainer = shap.Explainer(logistic_model, data_train)

# Calcular los valores SHAP para el conjunto de prueba
shap_values = explainer(data_test)

shap.summary_plot(shap_values, data_test, feature_names=data_test.columns)
```

Ilustración 41. Código Fuente SHAP

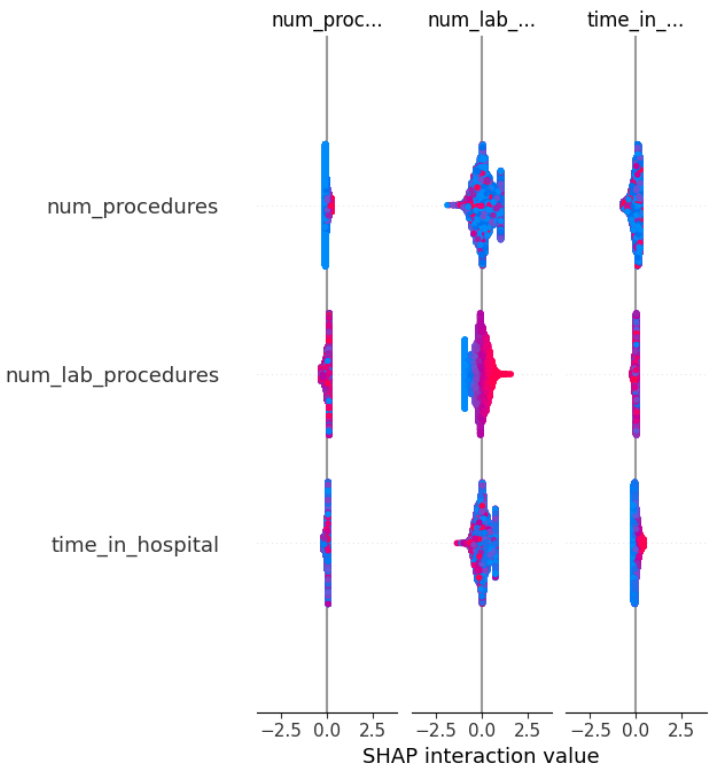


Ilustración 42. Resultado SHAP 1

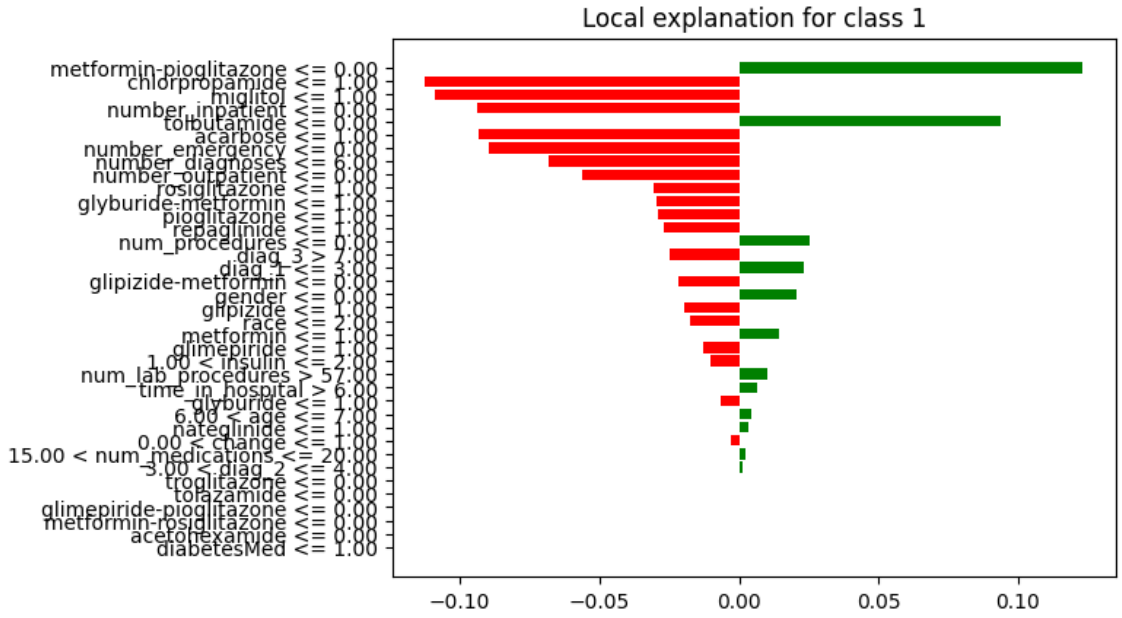


Ilustración 43. Resultado SHAP 2

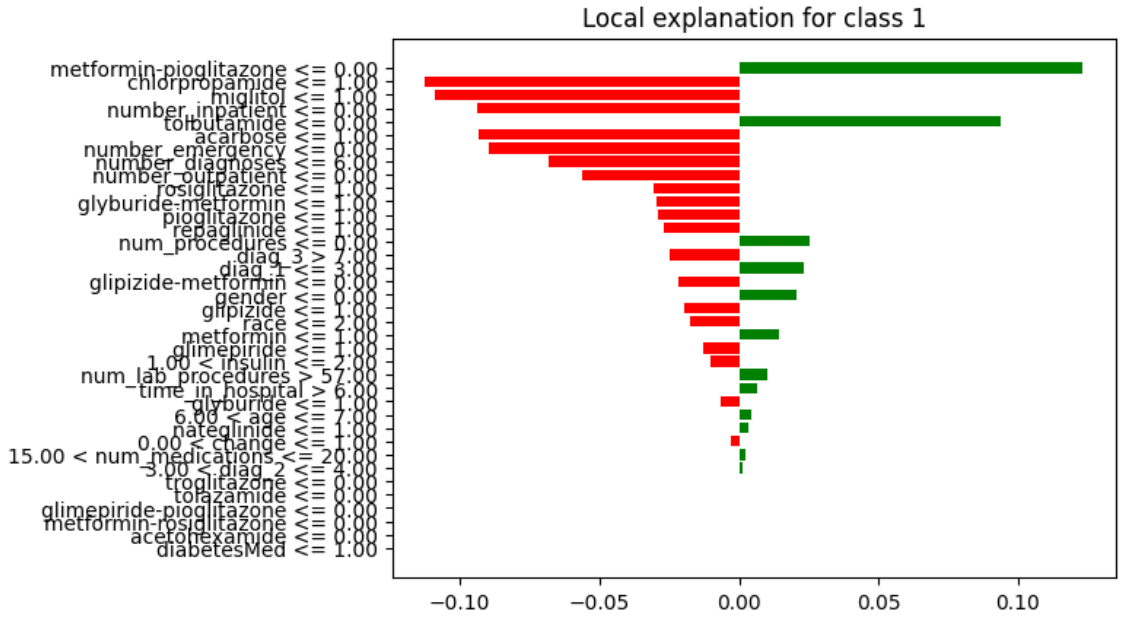


Ilustración 44. Resultado SHAP 3

A continuación, se ha utilizado la librería lime con la que se proporciona una descripción de cómo las características específicas contribuyeron a la predicción de que la instancia dada. Mostrará qué características influyeron más en esa predicción y cómo lo hicieron. Esto nos ayuda a entender cómo funcionan los modelos.

Para ello primero, se crea un objeto LimeTabularExplainer, que se utiliza para explicar las predicciones del modelo. Luego, se selecciona una instancia específica de los datos de prueba y se utiliza el método explain_instance para generar una

explicación de cómo el modelo ha llegado a su predicción para esa instancia en particular.

```
# Crear un objeto explainer de Lime para datos tabulares
explainer_lime = lime.lime_tabular.LimeTabularExplainer(data_train.values,
                                                        feature_names=data_train.columns)

# Explicar una instancia específica utilizando el modelo de AdaBoost
lime_explanation = explainer_lime.explain_instance(data_test.iloc[15].values,
                                                   adaboost_classifier.predict_proba,
                                                   num_features=len(data_train.columns))

# Mostrar la explicación en un cuaderno (si estás utilizando Jupyter Notebook)
lime_explanation.show_in_notebook()
```

Ilustración 45. Código Fuente LIME

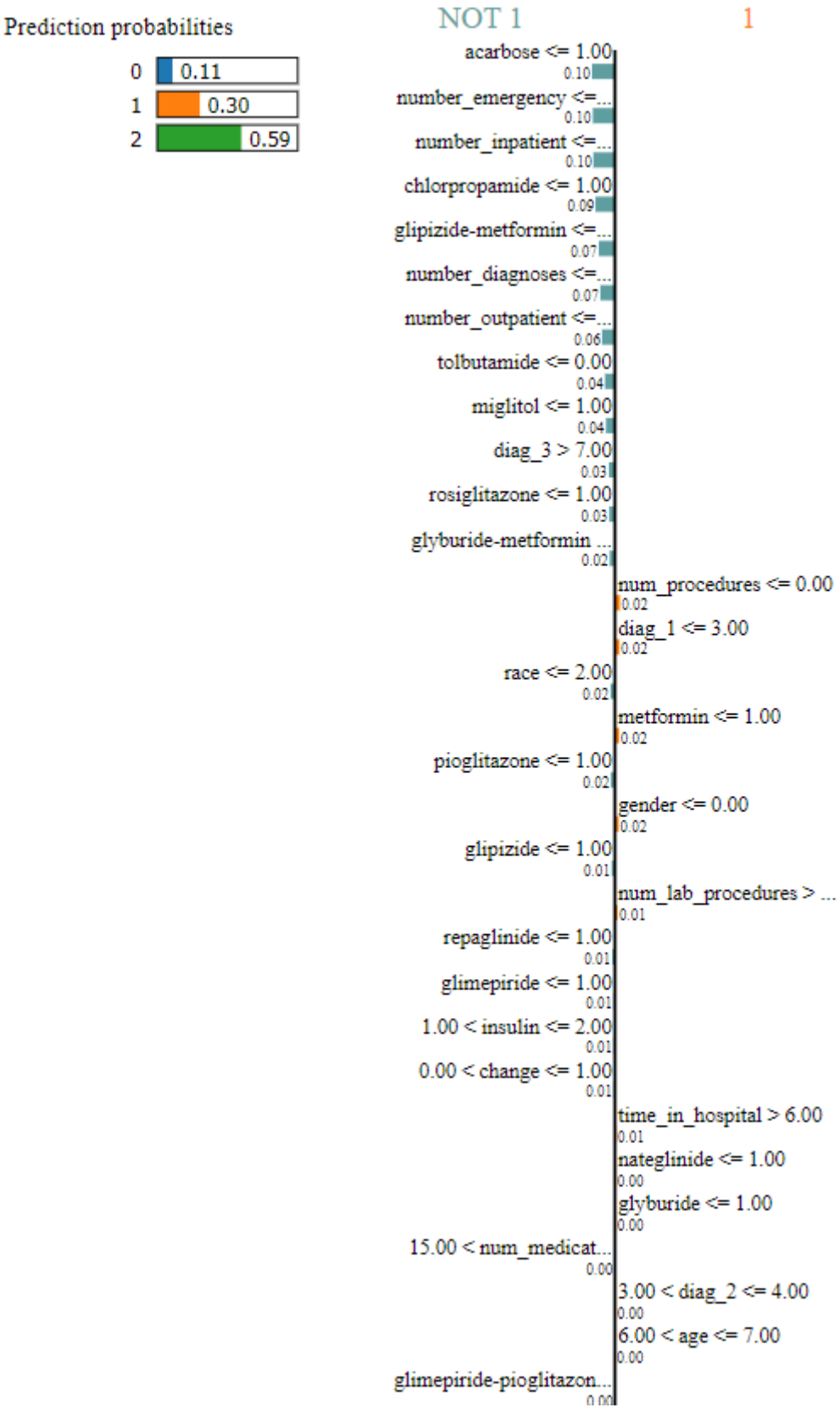


Ilustración 46. Resultado LIME para instancia 1

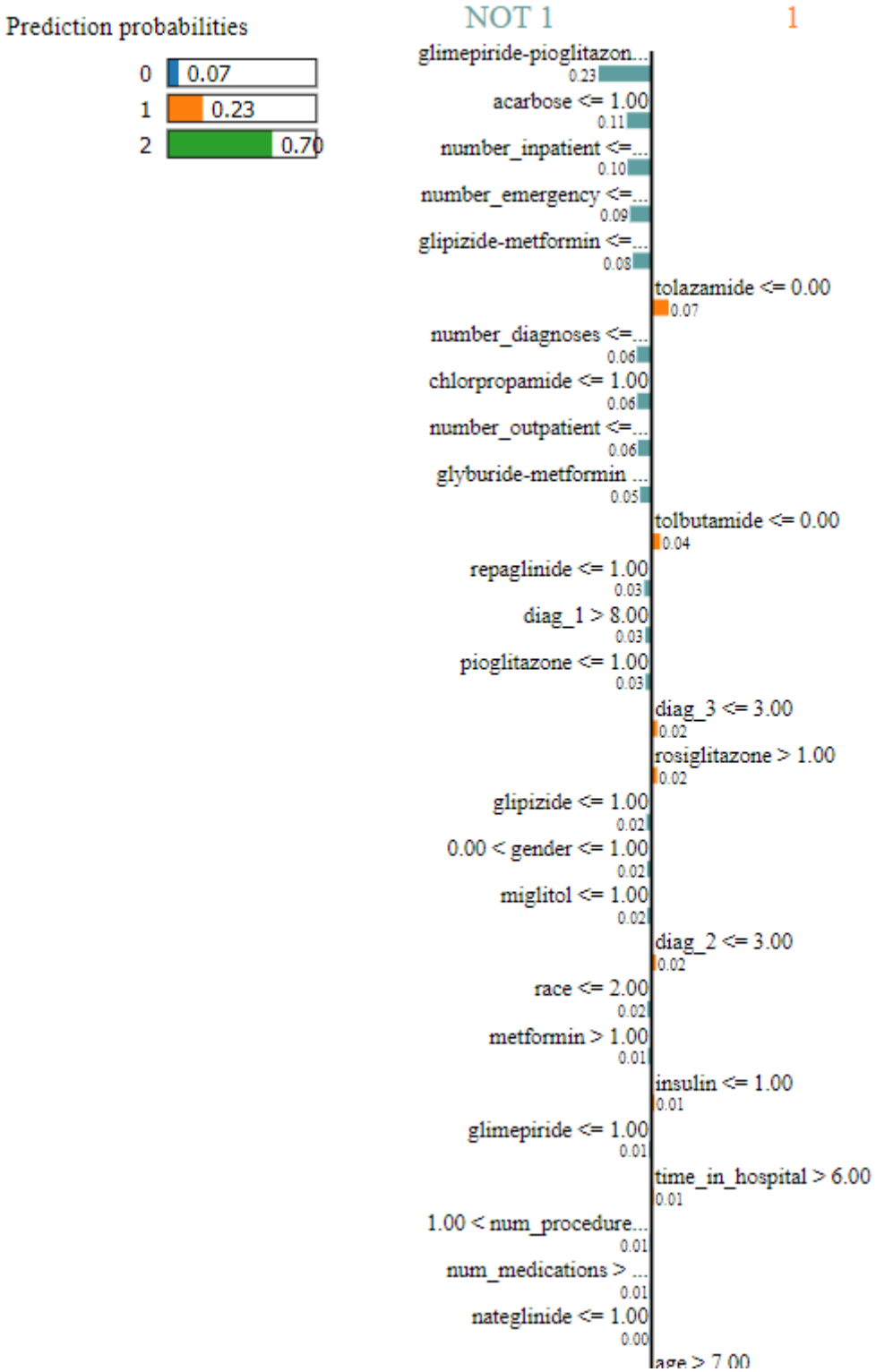


Ilustración 47. Resultado LIME para instancia 6

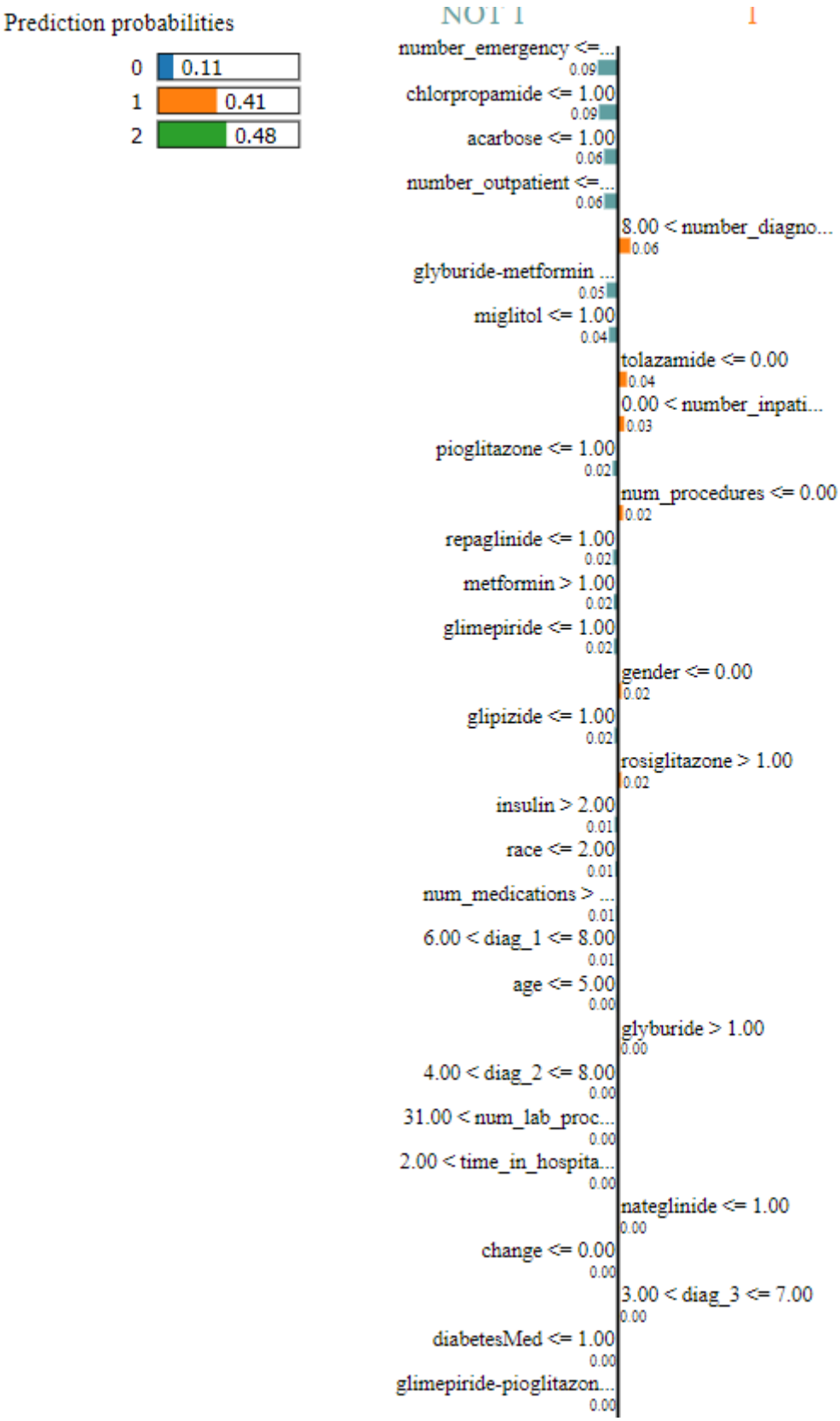


Ilustración 48. Resultado LIME para instancia 16

En la siguiente tabla se muestran los pesos de las características calculados con lime y que corroboran la hipótesis previa sobre que hay variables que influyen mucho más en la predicción y que otras. Además, se ve claramente como hay variables que no aportan nada y que pueden ser eliminadas.

Tabla 3. Valores de las características calculados con LIME

Feature	Value
metformin-pioglitazone	0.00
number_inpatient	0.00
chlorpropamide	1.00
number_emergency	0.00
acarbose	1.00
glyburide-metformin	1.00
number_diagnoses	2.00
number_outpatient	0.00
miglitol	1.00
diag_1	14.00
pioglitazone	1.00
rosiglitazone	2.00
race	2.00
tolazamide	0.00
diag_3	0.00
repaglinide	1.00
diag_2	2.00
gender	1.00
insulin	1.00
glipizide	1.00
metformin	2.00
glimepiride	1.00
num_procedures	2.00
time_in_hospital	7.00
num_medications	27.00
age	8.00
nateglinide	1.00
glyburide	1.00
change	0.00
num_lab_procedures	42.00
diabetesMed	1.00
tolbutamide	0.00
troglitazone	0.00
acetohexamide	0.00
glipizide-metformin	0.00
glimepiride-pioglitazone	0.00
metformin-rosiglitazone	0.00

Filtrado de datos y reducción de la dimensionalidad

Usar clasificadores avanzados, filtrado de datos o algoritmos de reducción de la dimensionalidad existentes scikit-learn o en otras librerías que no se hayan estudiado en prácticas y cuyos resultados pudieran ser relevantes.

Para filtrar características o reducir la dimensionalidad de los datos, puedes considerar el uso de algoritmos de selección de características y reducción de dimensionalidad disponibles en scikit-learn y otras bibliotecas.

Por último, se utiliza el clasificador Gradient Boosting. Este modelo destaca por su alto rendimiento con grandes conjuntos de datos. El conjunto de datos que utilizamos tiene 100.000 instancias, junto a un alto riesgo de sobre ajuste se requiere un modelo que pueda mitigarlo y sea robusto. Además, el Gradient Boosting capaz de capturar relaciones complejas entre las características y las etiquetas de los datos, lo que puede llevar a modelos altamente predictivos.

```
# Inicializar el clasificador Gradient Boosting
gb_classifier = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, random_state=42)

# Entrenar el clasificador
gb_classifier.fit(data_train, target_train)

# Realizar predicciones en el conjunto de prueba
y_pred = gb_classifier.predict(data_test)

y_proba = gb_classifier.predict_proba(data_test)
print("Resultados del clasificador con n_estimators = 100 y learning_rate = 0.1")
# Calcular la precisión
accuracy = accuracy_score(target_test, y_pred)
print("Precisión del clasificador Gradient Boosting: {:.4f}".format(accuracy))

auc_scores = roc_auc_score(target_test, y_proba, multi_class='ovr')
print("Roc AUC score: {:.4f}".format(auc_scores))
```

```
Resultados del clasificador con n_estimators = 100 y learning_rate = 0.1
Precisión del clasificador Gradient Boosting: 0.5773
Roc AUC score: 0.6493
```

Ilustración 49. Gradient Boosting

Se han llevado acabado dos experimentos seleccionando características relevantes, y reduciendo la dimensionalidad de los datos. El primero se ha realizado con las técnicas SelectKBest y PCA respectivamente, mientras que el segundo se ha realizado con SelectFromModel y Mainfold Approximation and Projection.

SelectKBest y PCA son técnicas utilizadas para seleccionar características relevantes y reducir la dimensionalidad de los datos en el aprendizaje automático. SelectKBest identifica las características más informativas utilizando pruebas estadísticas como la prueba chi-cuadrado, lo que mejora la eficiencia y la

interpretabilidad del modelo al centrarse en las características más influyentes. Por otro lado, PCA transforma el conjunto de datos en un conjunto más compacto de componentes principales, capturando la mayor variabilidad posible en menos dimensiones y facilitando la interpretación de los datos, este en concreto crea un conjunto con dos componentes principales. Ambas técnicas son herramientas valiosas para mejorar el rendimiento y la comprensión de los modelos de aprendizaje automático al abordar problemas de dimensionalidad y selección de características.

```
# Escalar las características para que estén en el rango [0, 1]
scaler = MinMaxScaler()
data_train_scaled = scaler.fit_transform(data_train)
data_test_scaled = scaler.transform(data_test)

# Seleccionar las mejores características utilizando chi-cuadrado (puede variar según el tipo de problema
selector = SelectKBest(score_func=chi2, k=2)
data_train_selected = selector.fit_transform(data_train_scaled, target_train)
data_test_selected = selector.transform(data_test_scaled)

# Reducir la dimensionalidad utilizando PCA
pca = PCA(n_components=2)
data_train_pca = pca.fit_transform(data_train_scaled)
data_test_pca = pca.transform(data_test_scaled)

# Entrenar un clasificador Gradient Boosting utilizando las características seleccionadas por SelectKBest
clf_kbest = GradientBoostingClassifier(random_state=42)
clf_kbest.fit(data_train_selected, target_train)
y_pred_kbest = clf_kbest.predict(data_test_selected)
accuracy_kbest = accuracy_score(target_test, y_pred_kbest)
auc_kbest = roc_auc_score(target_test, clf_kbest.predict_proba(data_test_selected), multi_class='ovr')
print("Accuracy using SelectKBest with Gradient Boosting:", accuracy_kbest)
print("AUC using SelectKBest with Gradient Boosting:", auc_kbest)

# Entrenar un clasificador Gradient Boosting utilizando las características reducidas por PCA
clf_pca = GradientBoostingClassifier(random_state=42)
clf_pca.fit(data_train_pca, target_train)
y_pred_pca = clf_pca.predict(data_test_pca)
accuracy_pca = accuracy_score(target_test, y_pred_pca)
auc_pca = roc_auc_score(target_test, clf_pca.predict_proba(data_test_pca), multi_class='ovr')
print("Accuracy using PCA with Gradient Boosting:", accuracy_pca)
print("AUC using PCA with Gradient Boosting:", auc_pca)
```

Ilustración 50. Selección de características y reducción de dimensionalidad con SelectKBest y PAC

A continuación, se muestran los resultados del primer experimento. Esto se puede deber a varios motivos, el modelo puede haber seleccionado de forma inadecuada las características, la reducción de la dimensionalidad puede haber llevado a una pérdida de capacidad de generalización del modelo y haber resultado en una disminución del valor de AUC.

```
Accuracy using SelectKBest with Gradient Boosting: 0.5641459004795221
AUC using SelectKBest with Gradient Boosting: 0.6074654094745545
Accuracy using PCA with Gradient Boosting: 0.5370253910856064
AUC using PCA with Gradient Boosting: 0.5492628113899763
```

Ilustración 51. Resultados Gradient Boosting tras la aplicación de técnicas de selección de características

```
Precisión del clasificador Gradient Boosting: 0.5758
Roc AUC score: 0.6519
```

Ilustración 52. Resultados clasificadores con data_train estándar

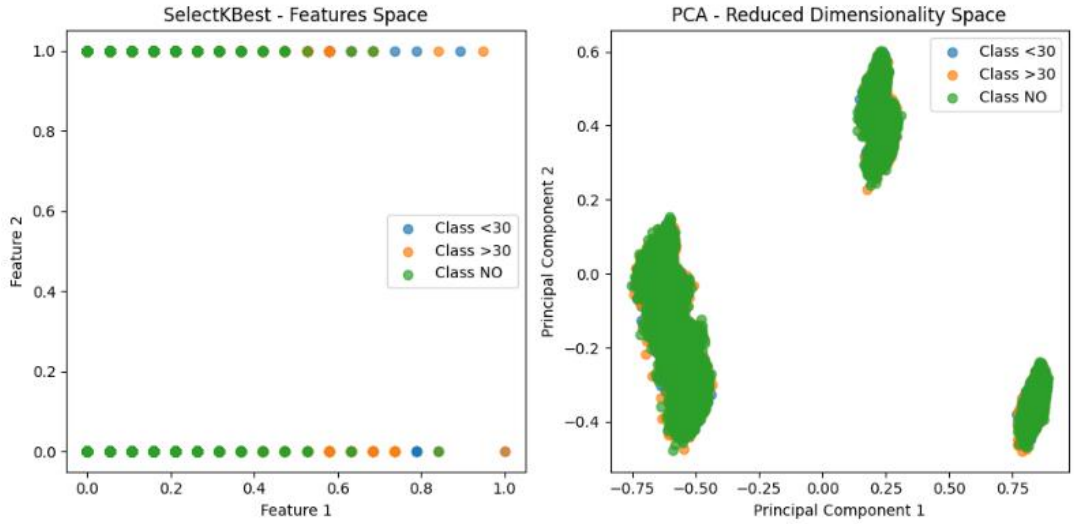


Ilustración 53. Resultados SelectKBest y PCA

A la vista de estos resultados se hizo un estudio para conocer cuáles serían los mejores parámetros para utilizar SelectKBest y PCA. Con la técnica de la validación cruzada utilizando cinco particiones mediante GridSearchCV se calculan los mejores parámetros para 'n_components' y 'k'.

```python
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import f1_score

warnings.filterwarnings("ignore")
param_grid_KBest = {'k': [2, 4, 6, 8]}
selector = SelectKBest(score_func=chi2)

# el mejor valor de k utilizando F1 como métrica de puntuación
grid_search = GridSearchCV(estimator=selector, param_grid=param_grid_KBest, cv=5, scoring='f1_micro')
grid_search.fit(data_train_scaled, target_train)

best_k = grid_search.best_params_['k']
print("El mejor valor de k es:", best_k)
```

```
El mejor valor de k es: 2
```

Ilustración 54. Mejor hiperparámetros para SelectKBest

```
warnings.filterwarnings("ignore")
#Utilizamos la técnica del cross validation para encontrar el mejor parámetro para reducir
#la dimensionalidad
param_grid_pca = {'n_components': [20,30,31,35]}  # Definir los valores de n_components a probar
pca = PCA()
grid_search = GridSearchCV(estimator=pca, param_grid=param_grid_pca, cv=5)
grid_search.fit(data_train_scaled)
best_n_components_pca = grid_search.best_params_['n_components']
print("El n_componentes mejor es:", best_n_components_pca)
```

```
El n_componentes mejor es: 31
```

Ilustración 55. Mejor hiperparámetros para k

Los resultados de la segunda ejecución, esta vez utilizando los hiperparámetros que se han establecido como los mejores, mejoran los resultados anteriores.

```
Accuracy using SelectKBest with Gradient Boosting: 0.5641459004795221
AUC using SelectKBest with Gradient Boosting: 0.6074654094745545
Accuracy using PCA with Gradient Boosting: 0.5728323245027906
AUC using PCA with Gradient Boosting: 0.642116529920123
```

Ilustración 56. Resultados Gradient Boosting

La precisión del modelo ha alcanzado el mismo nivel que el clasificador Gradient Boosting con el conjunto general de las características, mientras que la capacidad de discriminación ha aumentado 5 puntos, aun así, permanece un punto por detrás del resultado del clasificador con el conjunto general de las características.

Se utilizan a continuación otra pareja de técnicas de selección de características y modificación de las dimensiones para intentar mejorar la precisión y capacidad de discriminación. En este segundo experimento se utilizan las técnicas de Select From Model y Mainfold Approximation and Projection (UMAP).

SelectFromModel es una técnica que utiliza un estimador de aprendizaje automático para seleccionar automáticamente las características más importantes del conjunto de datos. El estimador evalúa la importancia de cada característica y selecciona aquellas que contribuyen significativamente a la predicción del objetivo. Esto ayuda a mejorar la eficiencia computacional y la interpretabilidad del modelo al utilizar solo las características más relevantes.

Por otro lado, la reducción de dimensionalidad utilizando Mainfold Approximation and Projection (UMAP) es una técnica no lineal que mapea los datos de alta dimensionalidad a un espacio de dimensionalidad menor mientras preserva las estructuras locales y globales de los datos. UMAP es especialmente útil para visualizar y explorar conjuntos de datos complejos, ya que permite representar datos de alta dimensión en un espacio bidimensional o tridimensional de manera más efectiva. Esto facilita la interpretación y comprensión de la estructura subyacente de los datos, así como la identificación de patrones y relaciones ocultas.

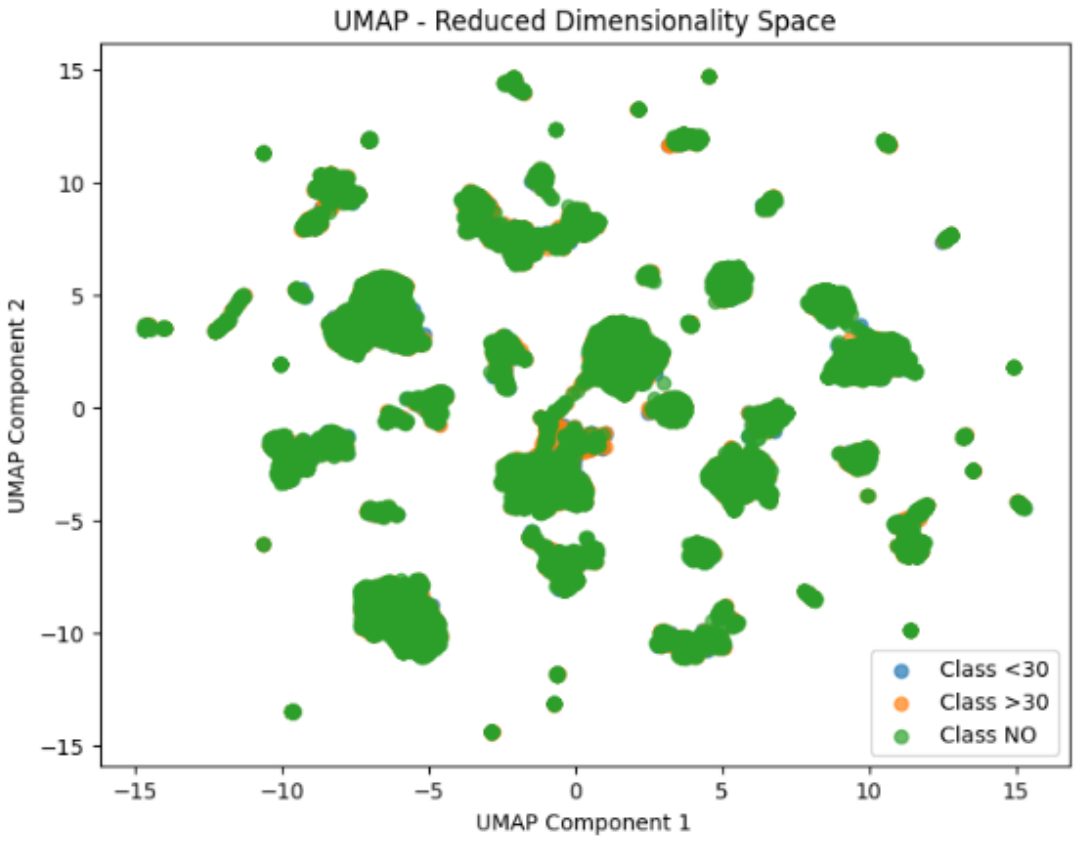
Esta gráfica proporciona una representación visual de cómo UMAP ha reducido la dimensionalidad de los datos y cómo se distribuyen las diferentes clases en el nuevo espacio de menor dimensión. Esto puede ayudar a entender mejor la estructura y las relaciones entre las instancias de datos en un espacio más manejable y fácil de interpretar.

```
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from umap import UMAP
warnings.filterwarnings('ignore')

# Escalar los datos
scaler = StandardScaler()
data_train_scaled = scaler.fit_transform(data_train)
data_test_scaled = scaler.transform(data_test)

# Seleccionar características utilizando SelectFromModel con Random Forest
clf = GradientBoostingClassifier(n_estimators=100, random_state=42)
selector = SelectFromModel(clf)
data_train_selected = selector.fit_transform(data_train_scaled, target_train)
data_test_selected = selector.transform(data_test_scaled)

# Entrenar clasificador con características seleccionadas
clf.fit(data_train_selected, target_train)
y_pred = clf.predict(data_test_selected)
accuracy = accuracy_score(target_test, y_pred)
print("Accuracy using SelectFromModel with GradientBoostingClassifier:", accuracy)

# Obtener las probabilidades de predicción en lugar de las clases predichas
y_prob = clf.predict_proba(data_test_selected)

# Calcular el AUC
auc = roc_auc_score(target_test, y_prob, multi_class='ovr')
print("AUC using SelectFromModel with GradientBoostingClassifier:", auc)
```

Ilustración 57. Selección de las características con Select From Model

```python
# Reducir la dimensionalidad utilizando UMAP
umap = UMAP(n_components=2)
data_umap = umap.fit_transform(data_train_scaled)

# Aplicar UMAP a los datos de prueba
data_test_umap = umap.transform(data_test_scaled)
clf_umap = GradientBoostingClassifier(random_state=42)
clf_umap.fit(data_umap, target_train)

#Predecir las etiquetas o las probabilidades de las muestras de prueba transformadas por UMAP
y_pred_umap = clf_umap.predict(data_test_umap)
y_prob_umap = clf_umap.predict_proba(data_test_umap)

#Calcular la precisión y el AUC utilizando las etiquetas verdaderas y las predicciones del clasificad
accuracy_umap = accuracy_score(target_test, y_pred_umap)
auc_umap = roc_auc_score(target_test, y_prob_umap, multi_class='ovr')

# Imprimir los resultados
print("Accuracy after applying UMAP:", accuracy_umap)
print("AUC Score after applying UMAP:", auc_umap)
```

Ilustración 58. Reducción de la dimensionalidad con UMAP



Los resultados de la selección de características y reducción de dimensionalidad con estas técnicas son los de la siguiente figura. Despues se grafican los datos reducidos por UMAP. Se puede apreciar que la precisión del modelo al seleccionar las características siguiendo el criterio de examinar como la importancia de estas afecta a la predicción, da un resultado bastante similar respecto a la precisión del modelo sin seleccionar las características. Esto implica que al seleccionar las

características no se han pasado por alto detalles importantes o considerado de más detalles irrelevantes. De la misma forma afecta a la capacidad de discriminación cuándo se han seleccionado las características mediante Select From Model, sin embargo, aplicando UMAP hay una disminución de la capacidad de discriminación de casi el 10%. Para reducir este margen y obtener unos resultados mejores se van a calcular hiperparámetros más adecuados al problema para utilizarlos en un segundo intento del estudio. Aun así, a la vista de los resultados hoy la capacidad de discriminación del modelo utilizando la técnica Select From Model ha aumentado un 4% respecto a utilizar la técnica SelectKBest. También se ha incrementado la precisión y la capacidad de discriminación al utilizar la reducción de dimensionalidad aplicando UMAP en comparación con PCA, en concreto 3 y 8 puntos respectivamente.

```
Accuracy using SelectFromModel with GradientBoostingClassifier: 0.5727930194167126
AUC using SelectFromModel with GradientBoostingClassifier: 0.6406008919469253
Accuracy after applying UMAP: 0.5404449335744045
AUC Score after applying UMAP: 0.5603109116525183
```

Ilustración 59. Resultados de SelectFromModel y UMAP

Sin embargo, antes de proseguir con el experimento, se ha representado el espacio reducido con la técnica UMAP, para dos clases: la primera >30 y <30 la segunda. Se ha mostrado también la precisión del modelo que es considerablemente superior cuando se trabaja con una variable binaria. Esto demuestra la correcta adaptación del modelo a los datos con los que trabaja, la todavía mejorable capacidad de discriminación y la precisión del modelo.

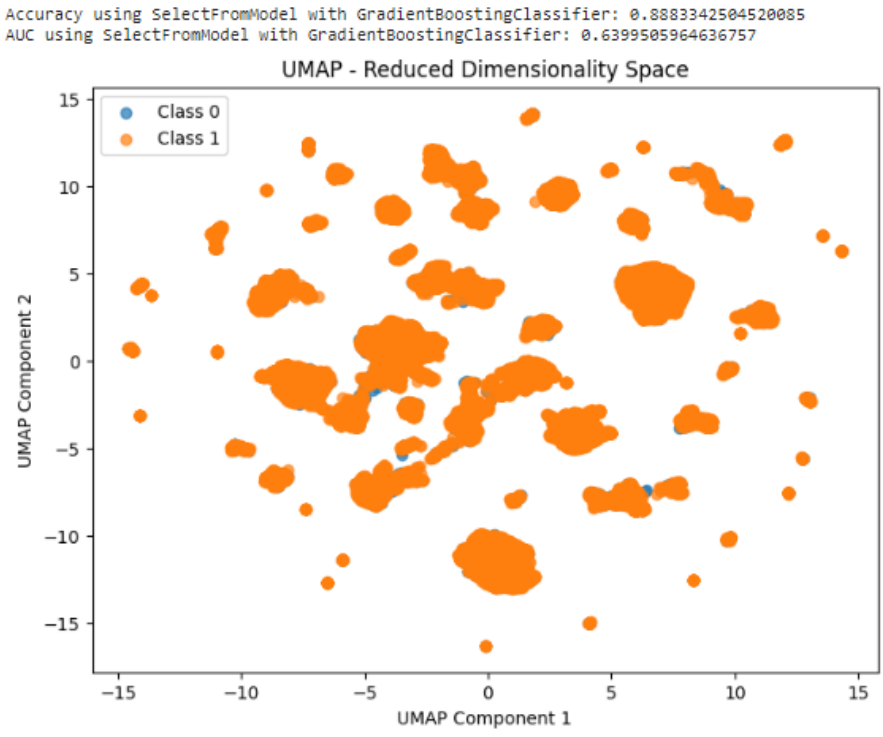


Ilustración 60. UMAP para dos características

Si se utiliza un 'n_components' = 30, y además para el clasificador Gradient Boosting se tiene en cuenta un 'n_estimators' = 150, se obtienen los siguientes resultados. Mejoran los resultados no significativamente pero sí que se puede considerar que son mejores que los anteriores. En la siguiente ilustración se puede observar de forma gráfica.

```
Accuracy using SelectFromModel with GradientBoostingClassifier: 0.574365222859838
AUC using SelectFromModel with GradientBoostingClassifier: 0.6423099365599019
```

Ilustración 61. Resultado SelectFromModel modificando hiperparámetros

```
Accuracy after applying UMAP: 0.5461441710557347
AUC Score after applying UMAP: 0.5820645550037554
```

Ilustración 62. Resultado UMAP modificando hiperparámetros.

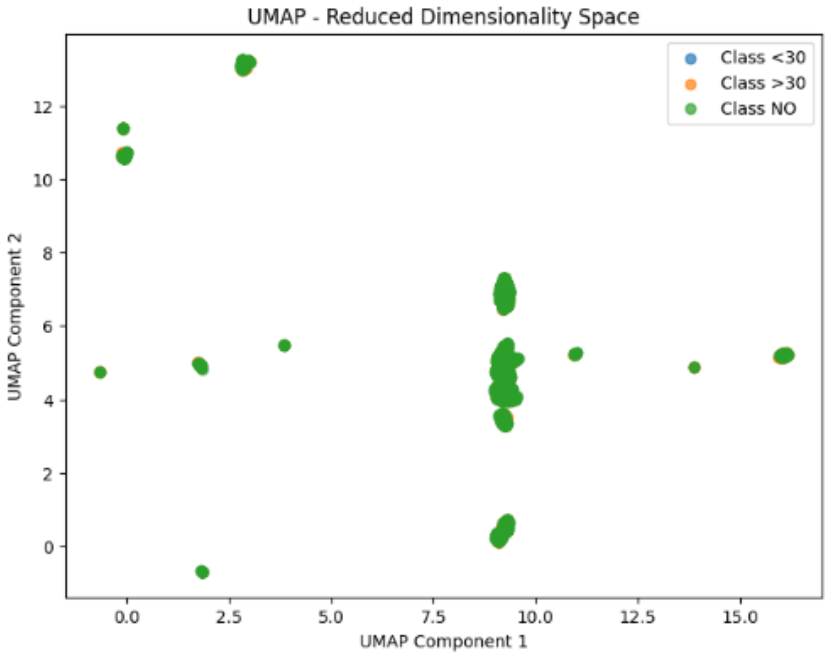


Ilustración 63. Representación UMAP Dimensionality Space

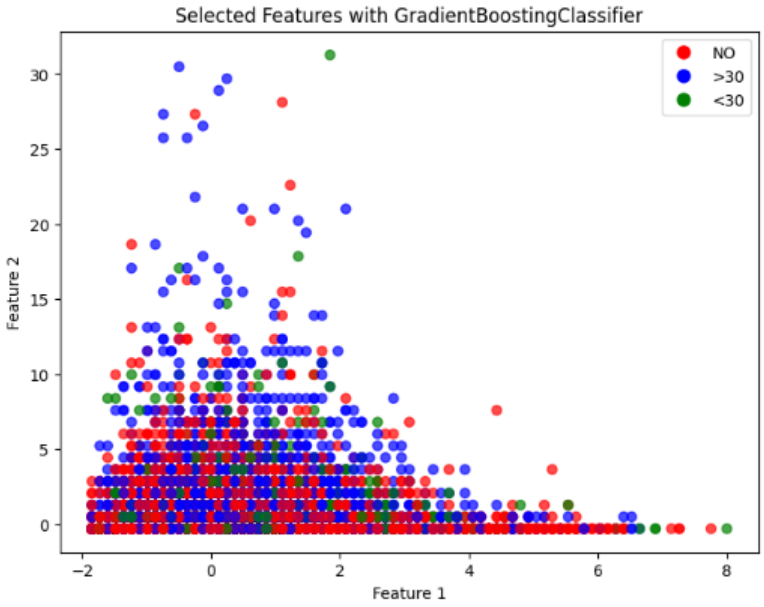


Ilustración 64. Representacion de selección de características

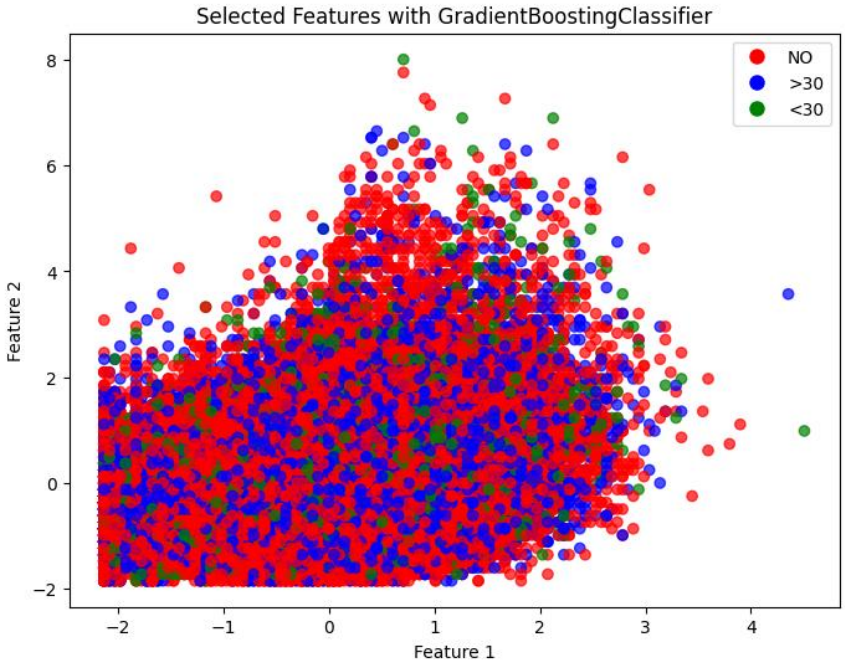


Ilustración 65. Representación de la selección de características tras mejorar los hiperparámetros

Comparación de las características de distribución

Por último, se busca comparar dos variables de la variable objetivo, en concreto >30 y <30. Se utiliza un histograma para visualizar la distribución de la primera característica en el conjunto de datos para las clases '<30' y '>30'.

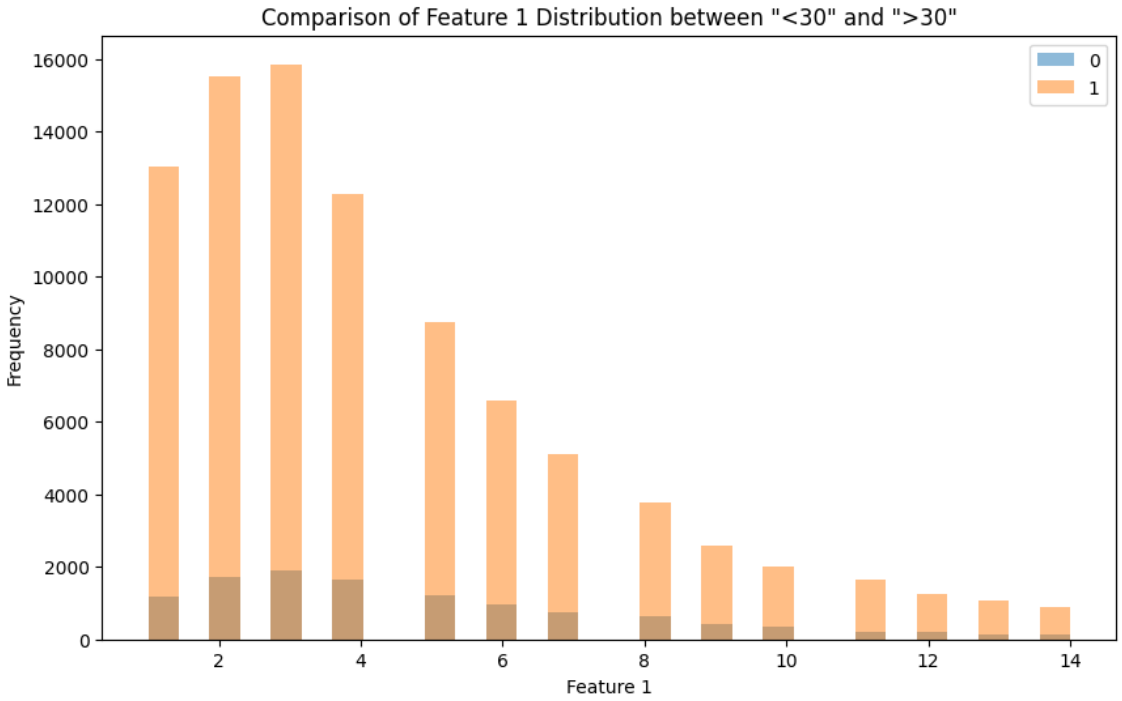


Ilustración 66. Histograma comparación '<30' y '>30'

Después se realiza una prueba de ANOVA (Análisis de Varianza) para comparar las medias de la primera característica entre las clases '<30' y '>30'. El F-value y el p-value se imprimen para evaluar la significancia de la diferencia.

El F-value (valor F) y el P-value (valor p) son resultados de una prueba de ANOVA (Análisis de Varianza) que se utiliza para comparar las medias de dos o más grupos en un conjunto de datos. Aquí está el significado de cada uno:

1. **F-value (valor F):** Es una medida de la variabilidad entre los grupos en comparación con la variabilidad dentro de los grupos. En el contexto de ANOVA, un valor F grande sugiere que la variabilidad entre los grupos es significativamente mayor que la variabilidad dentro de los grupos, lo que implica que al menos una de las medias de los grupos es significativamente diferente de las otras. Un valor F grande también puede indicar que el modelo es útil para predecir la variable dependiente.

2. **P-value (valor p):** Es la probabilidad de obtener un valor de prueba (en este caso, el valor F) al menos tan extremo como el observado en los datos, bajo la hipótesis nula de que no hay diferencia significativa entre los grupos. En otras palabras, el valor p indica la significancia estadística del resultado. Si el valor p es menor que un nivel de significancia predefinido (generalmente 0.05), se rechaza la hipótesis nula y se concluye que al menos uno de los grupos tiene una media significativamente diferente de los demás.

```
F-value: 147.8253894510799
P-value: 5.5688510585461525e-34
Feature Importances: [4.39789582e-02 2.36778938e-02 6.46625622e-03 3.37518040e-02
 6.63088390e-03 5.06906650e-02 6.53980991e-01 2.78006537e-02
 9.45600795e-04 7.49511313e-04 4.08039763e-02 2.53886513e-02
 2.53973781e-02 1.31845690e-02 5.25153114e-03 3.49036630e-03
 5.43934136e-04 0.00000000e+00 2.87112825e-03 0.00000000e+00
 2.82044594e-03 3.00150507e-03 4.86290963e-04 1.34071789e-03
 7.08910598e-04 1.61161817e-03 2.34112026e-03 0.00000000e+00
 0.00000000e+00 1.00584739e-02 1.18582751e-03 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 1.08403374e-02]
```

Ilustración 67. Resultados test estadístico ANOVA, F-value y P-value.

Los resultados que se muestran en la anterior ilustración indican una alta variabilidad entre los grupos en comparación con la variabilidad dentro de los grupos. Por otro lado, el valor p-value es extremadamente pequeño. Esto significa que la probabilidad de obtener un valor F al menos tan extremo como el que tenemos, tomando como hipótesis nula la no diferencia significativa entre los grupos, es prácticamente nula. Se rechaza por tanto la hipótesis nula y se concluye finalmente, que al menos una de las medias de los grupos es significativamente diferente de las otras.

Después, se hace un análisis de importancia de características utilizando un clasificador Gradient Boosting Classifier para calcular la importancia de las características. La importancia de cada característica se almacena en un arreglo **feature_importances**.

```
Mean Feature Values:
Class '0': [4.76824866e+00 4.42260280e+01 1.28088404e+00 1.69031434e+01
 4.36911156e-01 3.57312671e-01 1.22400282e+00 7.69278859e+00
 1.60931584e+00 4.58307652e-01 6.17601479e+00 6.17689531e+00
 6.07546007e+00 5.78814828e+00 1.16632914e+00 1.01928326e+00
 1.00695606e+00 1.00044026e+00 1.04552259e+00 0.00000000e+00
 1.12159901e+00 1.09800123e+00 8.80514220e-05 1.06753544e+00
 1.05934666e+00 1.00246544e+00 1.00000000e+00 0.00000000e+00
 2.64154266e-04 1.41164040e+00 1.00669191e+00 8.80514220e-05
 0.00000000e+00 0.00000000e+00 0.00000000e+00 5.10610196e-01
 8.02236506e-01]
Class '1': [4.34922408e+00 4.29536440e+01 1.34712252e+00 1.59111372e+01
 3.60871152e-01 1.77803095e-01 5.61647624e-01 7.38866706e+00
 1.59745158e+00 4.62962758e-01 6.08673915e+00 6.37849108e+00
 5.97651782e+00 5.66199162e+00 1.19927220e+00 1.01482153e+00
 1.00692409e+00 1.00094017e+00 1.05102368e+00 1.10608457e-05
 1.12117156e+00 1.10199206e+00 2.43338606e-04 1.07254809e+00
 1.06299152e+00 1.00314128e+00 1.00033183e+00 3.31825371e-05
 4.09251291e-04 1.40469422e+00 1.00692409e+00 1.32730149e-04
 1.10608457e-05 2.21216914e-05 1.10608457e-05 5.41494763e-01
 7.65985687e-01]
Selected Feature Indices: [0 3 5 6 7]
```

Ilustración 68. Resultados Mean Features Values.

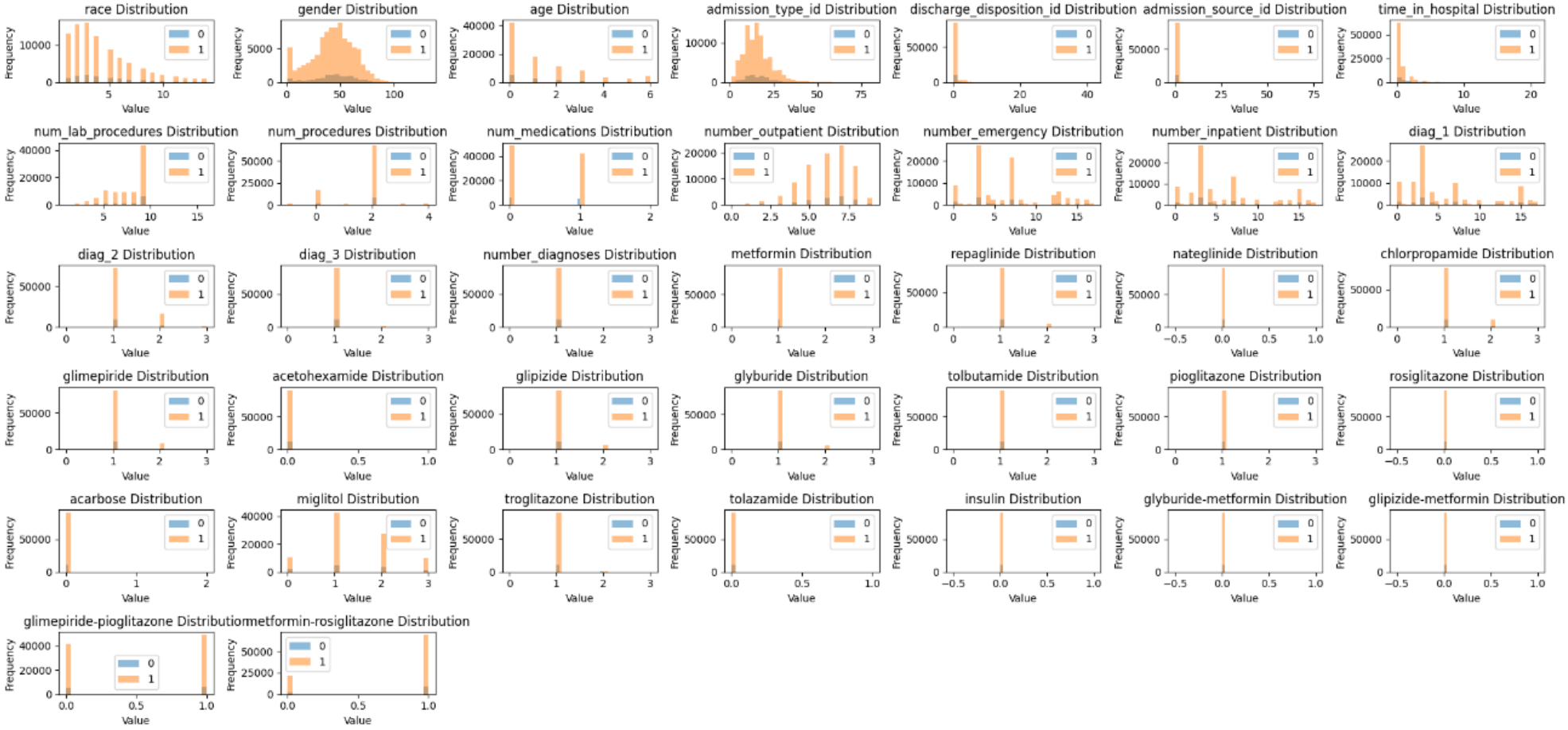


Ilustración 69. Distribución de Features

En base a las gráficas que se muestran y apoyándonos en los resultados que se han obtenido con lime anteriormente se ha llegado a la conclusión de que se pueden eliminar características. Estas características contienen más del 90% de instancias dentro del mismo valor único por lo que no aportan en principio valor para la predicción.

Para comprobarlo se ha realizado un bucle que evalúa la cantidad de instancias que tiene que cada valor único de cada columna.

```
# Bucle for para recorrer las columnas
for column in diabetic_data.columns:
    # Calcular los valores únicos en la columna actual
    unique_values = diabetic_data[column].unique()

    # Calcular el porcentaje de filas con el mismo valor único
    most_common_value_percentage = diabetic_data[column].value_counts(normalize=True).max() * 100

    # Agregar la columna a la lista si cumple el criterio
    if most_common_value_percentage > 85:
        columnas_a_eliminar.append(column)

# Mostrar las columnas a eliminar
print("Columnas a eliminar:")
print(columnas_a_eliminar)

# Eliminar las columnas utilizando el método drop()
diabetic_data = diabetic_data.drop(columnas_a_eliminar, axis=1)
```

A continuación, se ha comprobado volviendo a entrenar y a evaluar los diferentes modelos. Y estas han sido las graficcas que se han obtenido

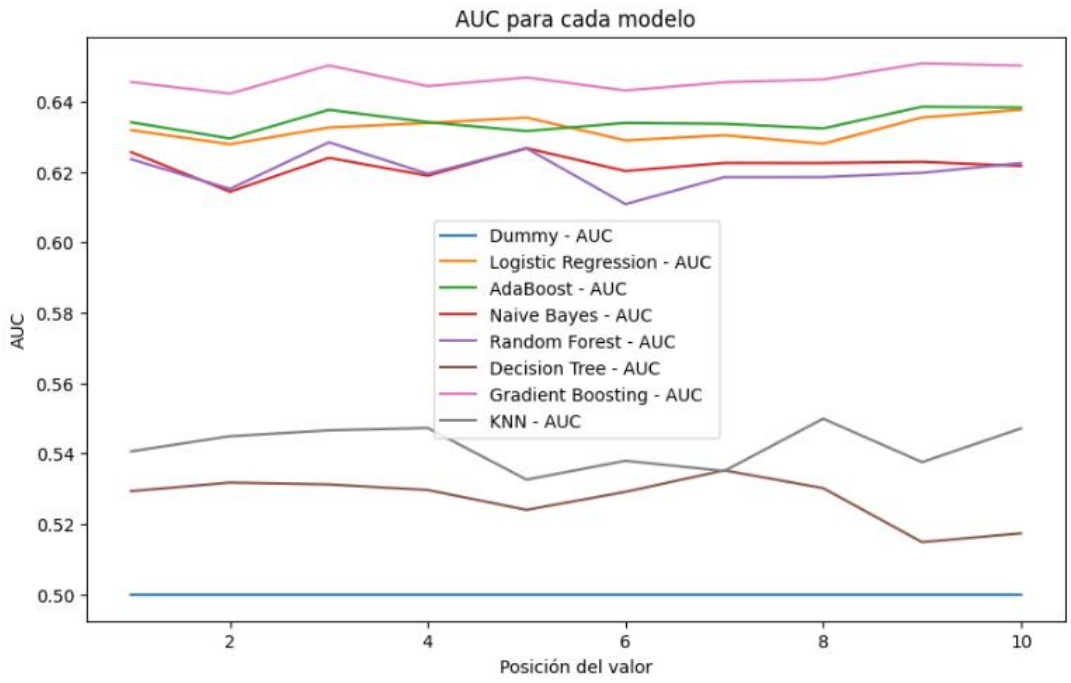


Ilustración 70. Valores AUC con características reducidas

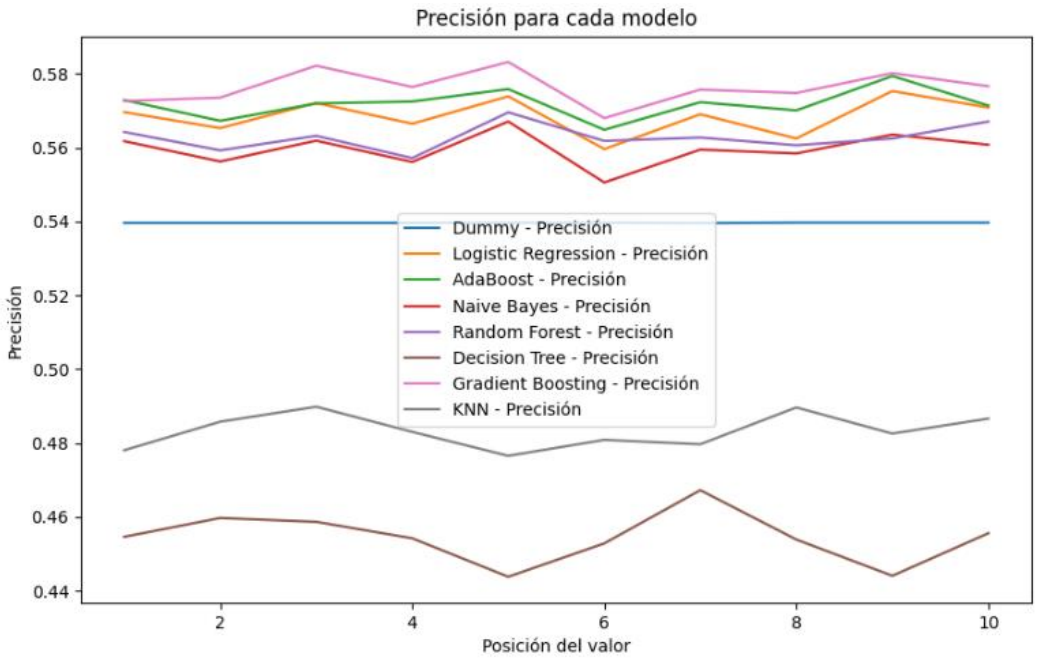


Ilustración 71. Valores de precisión con características reducidas

Comparándolas con las ilustraciones 15 y 16 podemos ver como el AUC aumenta considerablemente, aunque en algunos modelos se reduce la precisión.

Reducción de los datos.

Según la reducción de los datos a la vista de las distribuciones de las características se han obtenido los siguientes resultados. El modelo que más ha mejorado con la reducción de características ha sido Naive Bayes. Se observa un aumento de 0.44 en la precisión y de 0.04 en el AUC, lo que indica una mejora considerable en la capacidad del modelo para clasificar correctamente las instancias y para discriminar entre las clases. Esta optimización sugiere que la selección cuidadosa de características puede tener un impacto positivo en el rendimiento del clasificador Naive Bayes.

	Mean Accuracy	Mean AUC
Dummy	0.539660	0.500000
Logistic Regression	0.568484	0.632175
AdaBoost	0.571865	0.634327
Naive Bayes	0.559601	0.621917
Random Forest	0.562824	0.618809
Decision Tree	0.456160	0.528713
Gradient Boosting	0.576346	0.646459
KNN	0.483269	0.541960

Ilustración 72. AUC y accuracy con Stratified Kfold.

A continuación, con Grid Search CV se han vuelto a calcular los mejores hiperparámetros para el nuevo conjunto de datos obteniendo los siguientes.

```
Searching best parameters for Logistic Regression...
Best parameters found:
{'C': 10, 'penalty': 'l2'}

Searching best parameters for AdaBoost...
Best parameters found:
{'learning_rate': 1, 'n_estimators': 200}

Searching best parameters for Random Forest...
Best parameters found:
{'max_depth': 10, 'n_estimators': 100}

Searching best parameters for Decision Tree...
Best parameters found:
{'max_depth': 10, 'min_samples_split': 2}

Searching best parameters for Gradient Boosting...
Best parameters found:
{'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 200}

Searching best parameters for KNN...
Best parameters found:
{'n_neighbors': 9, 'weights': 'distance'}
```

Ilustración 73. Hiperparámetros según GridSearchCV

Al aplicarlos a la implementación se obtienen los resultados de las siguientes ilustraciones. El modelo de todos que más ve afectado su capacidad de discriminación por la mejora de hiperparámetros es el Gradient Boosting, se ve incrementada medio punto.

```
Precisión del modelo: 0.5666208116340768
AUC scores for each class (Logistic Regression): 0.6332430510316702
```

Ilustración 74. Linear Model

```
Precisión del modelo AdaBoost: 0.573793848874914
AUC scores for each class (AdaBoost): 0.6362415773568381
```

Ilustración 75. AdaBoost

```
Resultados del clasificador con n_estimators = 100 y learning_rate = 0.1
Precisión del clasificador Gradient Boosting: 0.5762
Roc AUC score: 0.6477
```

Ilustración 76. Gradient Boosting

Además, se ha vuelto a implementar las técnicas de oversampling y Undersampling con el nuevo conjunto de datos.

	Mean Accuracy (Oversampling)	Mean AUC (Oversampling)	\
Dummy	0.333318	0.499980	
Logistic Regression	0.432251	0.617215	
AdaBoost	0.440130	0.623209	
Naive Bayes	0.413926	0.609604	
Random Forest	0.826570	0.947121	
Decision Tree	0.764271	0.822493	
Gradient Boosting	0.452633	0.641741	
KNN	0.598121	0.784055	

	Mean Accuracy (Undersampling)	Mean AUC (Undersampling)
Dummy	0.333260	0.499901
Logistic Regression	0.426440	0.612543
AdaBoost	0.433495	0.616054
Naive Bayes	0.410420	0.607142
Random Forest	0.416373	0.600264
Decision Tree	0.368166	0.526719
Gradient Boosting	0.437169	0.623124
KNN	0.363169	0.535459

Ilustración 77. Resultados Oversampling y Undersampling con la reducción del conjunto de datos

Por último, se han realizado nuevos experimentos con SHAP y LIME. Y aplicado un hibrido de clasificador AdaBoost y Gradient Boosting. Todos los resultados se muestran a continuación.

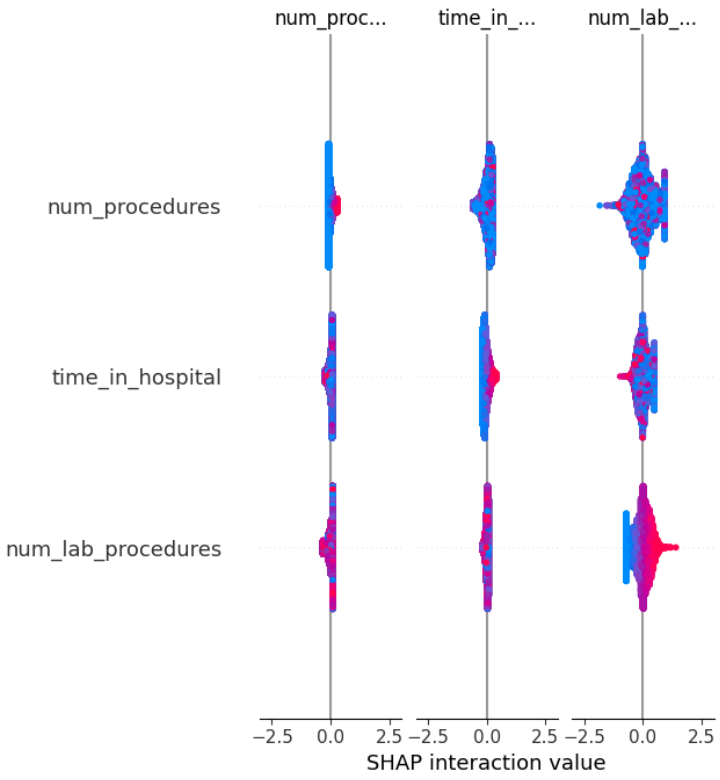


Ilustración 78. SHAP

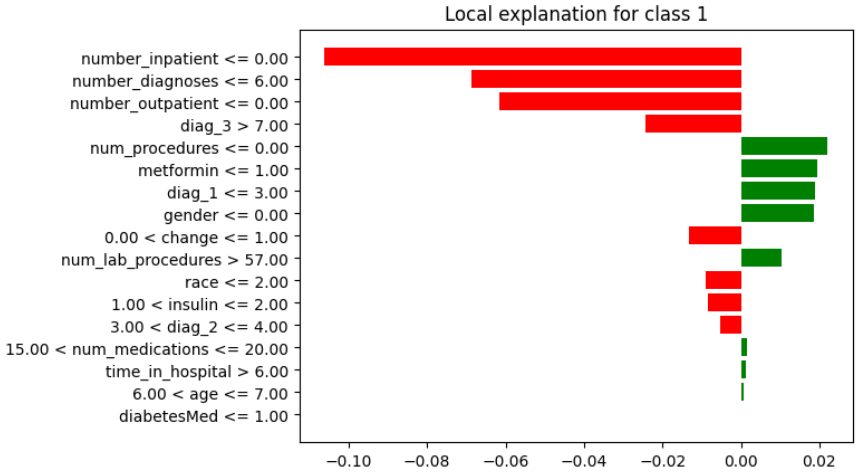


Ilustración 79. Resultado LIME 1

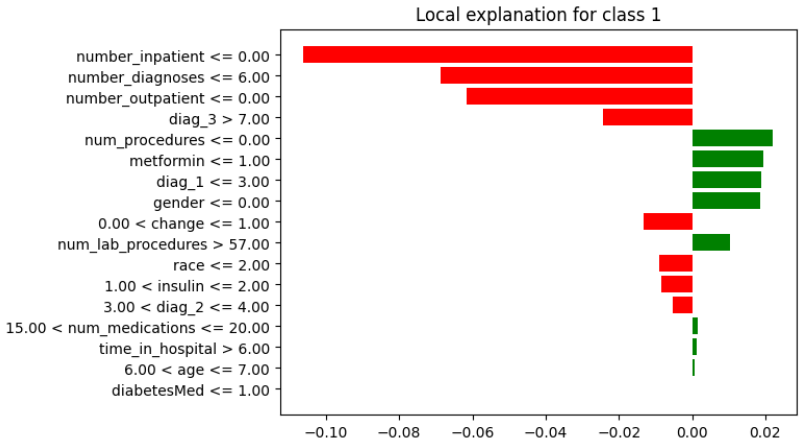


Ilustración 80. Resultado LIME 2

```
Resultados del clasificador con n_estimators = 100 y learning_rate = 0.1
Precisión del clasificador Gradient Boosting: 0.5747
Roc AUC score: 0.6460
```

Ilustración 81. AUC y accuray

```
Mean AUC for AdaBoost after SMOTE: 0.5917052249194091
Mean AUC for Gradient Boosting after SMOTE: 0.6066825206861977
```

Ilustración 82. Resultados de realizar SMOTE para AdaBoost y Gradient Boosting

El resultado de las nuevas distribuciones de las características despues de reducir el conjunto de datos es el de la siguiente ilustración.

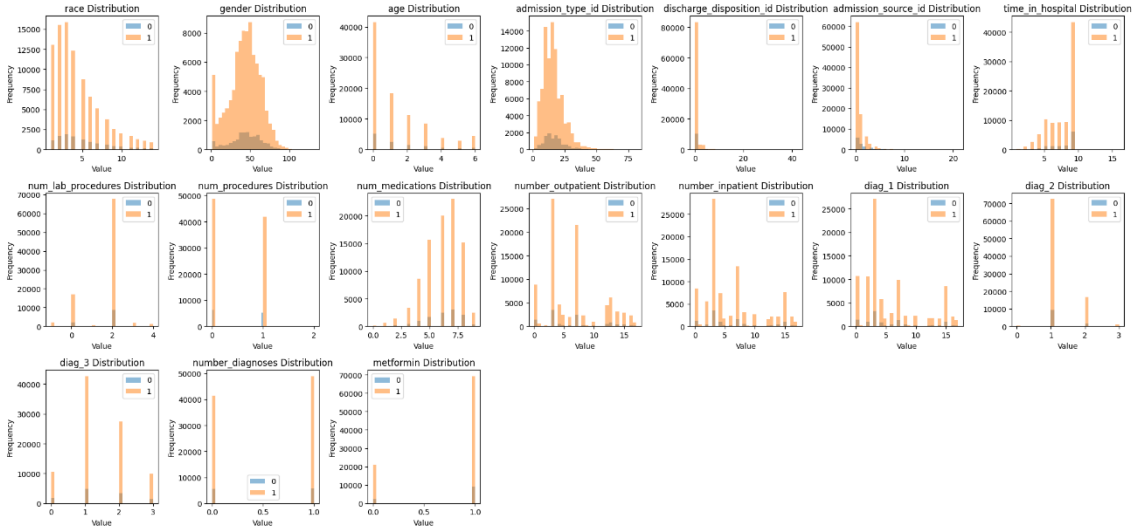


Ilustración 83. Distribución características.

Conclusión modelo avanzado

En este modelo mediante las técnicas LIME y SHAP se ha corroborado finalmente la hipótesis de que hay variables más importantes en los predictores y algunas que podrían ser eliminadas. Para comprobar esto último se ha llevado a cabo un análisis de la distribución de cada característica. El código se enfoca en analizar la distribución y realizar pruebas estadísticas entre las clases '<30' y '>30' en el conjunto de datos. Primero, se visualiza la comparación de la distribución de la primera característica mediante un histograma, lo que permite identificar posibles disparidades entre las clases. Luego, se realiza una prueba estadística ANOVA para comparar la media de esta característica entre las dos clases, brindando información sobre posibles diferencias significativas.

Posteriormente, se emplea el algoritmo Gradient Boosting para calcular la importancia de las características en la clasificación. Así se identifica qué características influyen más en la distinción entre las clases. Además, se calculan los perfiles de características promedio para cada clase, lo que proporciona una comprensión más profunda de las diferencias entre las clases en términos de valores característicos medios. Finalmente, se realiza la selección de características utilizando SelectKBest con ANOVA, lo que permite identificar las características más relevantes para la clasificación binaria. Los índices de las características seleccionadas se muestran, lo que facilita la identificación de las características más importantes para el modelo. Estas operaciones proporcionan una comprensión detallada de las características del conjunto de datos y su influencia en la clasificación binaria entre las clases '<30' y '>30'. Según este análisis se ha determinado cuáles son las características que se van a eliminar. Después de reducir aun mas el conjunto de datos pese a que alguno resultados mejoran la mayoría mantienen valores de precisión y de capacidad de

discriminación parecidos. Esto puede deberse a que las variables eliminadas no contribuían significativamente al modelo, es decir que no eran determinantes para conocer si un paciente debía ser o no readmitido.

Conclusión

En resumen, tras una evaluación en el nivel básico, se optimizó el conjunto de datos mediante la eliminación de datos irrelevantes e inconsistentes, y se estableció una línea base del problema con un Dummy Classifier. Sin embargo, la precisión inicial fue similar a la aleatoriedad, lo que indicó la necesidad de explorar otros clasificadores para mejorar el rendimiento del modelo. Se aplicaron Decision Tree y Linear Model, lo que resultó en una leve mejora en la precisión del modelo, aunque no significativa. No obstante, Linear Model demostró una mejor capacidad de discriminación según el valor de AUC.

Luego, se implementó la validación cruzada y el test de Wilcoxon para evaluar los modelos de manera más robusta y comparar su rendimiento de manera significativa. A pesar de los esfuerzos, la selección automática de atributos no produjo mejoras sustanciales, posiblemente debido al ruido en los datos.

Se aplicaron modelos más avanzados como Naive Bayes, AdaBoost, Random Forest, y Gradient Boosting. Tras esto se observaron mejoras notables en la precisión y capacidad de discriminación. Estas mejoras se lograron con los hiperparámetros estándar, pero se exploraron hiperparámetros optimizados utilizando GridSearchCV para obtener mejores resultados.

Despues de evaluar detenidamente cada modelo, de analizar cuáles son los mejores hiperparámetros que se adaptan a nuestro problema y al conjunto de datos se ha concluido a la vista de resultados que el modelo más preciso es el Gradient Boosting caracterizado por sus hiperparámetros, en concreto estos son: sido max_depth=6 y learning_rate=0.1.

El funcionamiento interno del modelo es el siguiente:

1. **Inicialización del modelo**: Se inicia con un modelo base, que puede ser un simple estimador, como un árbol de decisión poco profundo.
2. **Entrenamiento iterativo**: En cada iteración, se ajusta un nuevo modelo al residuo de los errores cometidos por el modelo anterior. El objetivo es reducir sistemáticamente los errores en la predicción del modelo en cada iteración.
3. **Construcción del árbol de decisión**: En cada iteración, se ajusta un árbol de decisión poco profundo al gradiente negativo del error de la función de pérdida (de ahí el nombre "Gradient Boosting"). Este árbol de decisión se ajusta de manera que minimize la función de pérdida.

4. **Paso de ajuste de tasa**: Se ajusta la tasa de aprendizaje, que controla la contribución de cada árbol al modelo final. Una tasa de aprendizaje más baja hace que el modelo sea más conservador, mientras que una tasa de aprendizaje más alta puede hacer que el modelo sea más propenso al sobreajuste.
5. **Suma de predicciones:** Las predicciones de cada nuevo modelo se agregan a las predicciones del modelo anterior, lo que permite que el modelo final sea la suma ponderada de todos los modelos en la secuencia.
6. **Finalización**: El proceso de entrenamiento continúa hasta que se alcanza un cierto criterio de detención, como un número máximo de iteraciones o hasta que se logra una mejora insuficiente en la función de pérdida.

Por otro lado, la importancia de ciertas características y la necesidad de eliminar otras menos relevantes se reveló con la evaluación de clasificadores sensibles al coste. Esto se confirmó mediante técnicas como LIME y SHAP, así como análisis estadísticos y selección de características, que se llevó a cabo con cuatro técnicas distintas descritas anteriormente. En última instancia, se volvió a optimizar el conjunto de datos permitió mejorar la precisión y la capacidad de discriminación del modelo, lo que demuestra la importancia de un proceso de evaluación profundo para desarrollar modelos de aprendizaje automático más precisos y fiables.

Referencias

Para abordar el problema planteado se han utilizado los notebooks proporcionados por el profesor de la asignatura. Además de páginas web recomendadas:

- https://scikit-learn.org/stable/index.html