



UA

Práctica Final

Daniel Asensi Roch

DNI: 48776120C

10 de diciembre de 2023

Índice

1. Introducción	2
1.1. Planteamiento del problema	2
1.2. Objetivos	2
2. Básico	3
2.1. Entender Dataset y Problema	3
2.2. Lectura de Dataset	3
2.3. Eliminación de todas las columnas de solo un valor	3
2.4. Eliminación de columnas con valores repetidos	4
2.5. Sacar porcentajes de nulos de columnas	4
2.6. Eliminación de columnas con alta proporción de nulos	5
2.7. Eliminación de filas con valores nulos	6
2.8. Eliminación de columnas IDs	6
2.9. Tratamiento individual de campos previamente estudiados	7
2.9.1. Estudio de valores y codificación de los mismos	7
2.10. Volvemos a sacar un el número de filas y columnas	8
2.11. Tipos de datos tras la transformación	9
2.12. Construcción de Datasets	9
2.13. Prueba de clasificadores	10
2.13.1. DummyClassifier	10
2.13.2. DecisionTreeClassifier	11
2.13.3. RandomForestClassifier	11
2.14. Obtención de mejores Hiperparámetros a mano	11
2.15. Comparativa de los cuatro clasificadores	13
3. Medio	14
3.1. Selección automática de atributos	14
3.2. Reducción de la dimensionalidad	16
3.3. Prueba de 3 Modelos nuevos y comparativa de los mismos con el mejor selector automatico	16
3.4. Prueba de Wilcoxon	17
4. Avanzado	19
4.1. Buscar parámetros óptimos de forma automática	19
4.1.1. GridSearchCV	19
4.1.2. BayesianOptimization	19
4.2. Prueba de Xgboost	20
4.2.1. BayesianOptimization a Xgboost	21
4.3. Comparativa de todos los clasificadores desarrollados	21
5. Conclusiones	23

1. Introducción

La diabetes es una enfermedad crónica que afecta a millones de personas en todo el mundo, representando un desafío significativo para los sistemas de salud. Entre las complicaciones asociadas con la diabetes, la readmisión hospitalaria de pacientes es un problema particularmente crítico. La readmisión no solo aumenta la carga sobre los sistemas de salud, sino que también afecta negativamente la calidad de vida de los pacientes. Este proyecto se centra en el análisis de los datos relacionados con pacientes diabéticos, con el objetivo de predecir y entender los factores que contribuyen a las readmisiones hospitalarias. Utilizando técnicas avanzadas de machine learning y análisis de datos, se busca desarrollar modelos predictivos que puedan servir de apoyo para tomar decisiones clínicas más informadas y mejorar la gestión del cuidado de la salud.

1.1. Planteamiento del problema

La readmisión hospitalaria de pacientes con diabetes es un desafío significativo que enfrenta el sistema de salud, con implicaciones en la calidad de la atención al paciente y costos sanitarios. Este problema es complejo debido a la naturaleza multifactorial de la diabetes y las múltiples vías de intervención posibles. La capacidad de predecir readmisiones puede permitir intervenciones más personalizadas y oportunas, reducir las estancias hospitalarias y mejorar los resultados clínicos. La identificación de factores clave que contribuyen a las readmisiones es, por tanto, de suma importancia para la gestión eficiente de la salud y el bienestar de los pacientes con diabetes.

1.2. Objetivos

Los objetivos de esta práctica son múltiples y se detallan a continuación:

- Explorar y preprocesar un conjunto de datos de pacientes con diabetes para facilitar el análisis posterior.
- Implementar y evaluar varios modelos de machine learning para predecir la readmisión hospitalaria de pacientes.
- Comparar la efectividad de diferentes métodos de selección de características y reducción de dimensionalidad.
- Optimizar los modelos de clasificación a través de la selección de hiperparámetros adecuados.
- Interpretar los resultados de los modelos para proporcionar insights clínicos útiles.

2. Básico

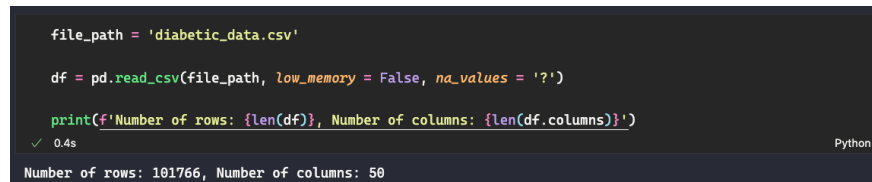
2.1. Entender Dataset y Problema

El dataset ***"diabetic_data.csv"*** se centra en los registros hospitalarios de pacientes con diabetes. Contiene múltiples variables clínicas y demográficas como edad, sexo, tipo de admisión, número de internaciones previas, resultados de laboratorio, diagnósticos y medicamentos recetados.

El problema a resolver es predecir la readmisión de pacientes con diabetes en hospitales. El objetivo es identificar patrones y factores que contribuyen a la readmisión, lo cual es crítico para mejorar la gestión del cuidado de la salud y reducir los costos hospitalarios. La tarea implica analizar y modelar estos datos para predecir eficazmente las readmisiones, lo que representa un reto importante debido a la complejidad y variedad de los factores involucrados.

2.2. Lectura de Dataset

Lo primero que debemos hacer es leer el dataset y ver qué contiene. Para ello, utilizamos la función `read_csv` de la librería `pandas`. Pero añadiremos un par de parámetros para que nos muestre más información. El parámetro ***"na_values='?'"*** indica que los valores marcados con un signo de interrogación en el archivo CSV deben ser tratados como valores ausentes (NA).



```
file_path = 'diabetic_data.csv'

df = pd.read_csv(file_path, low_memory = False, na_values = '?')

print(f'Number of rows: {len(df)}, Number of columns: {len(df.columns)}')
```

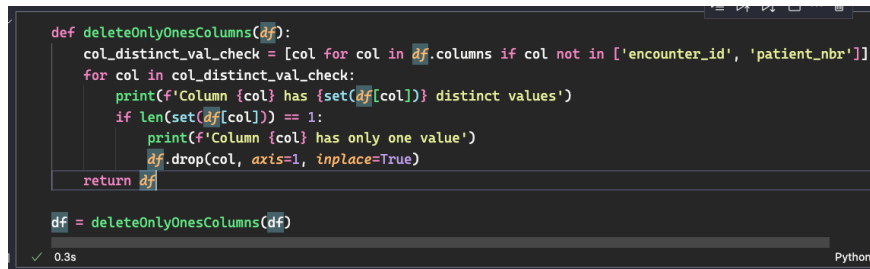
✓ 0.4s Python

Number of rows: 181766, Number of columns: 50

Figura 1

2.3. Eliminación de todas las columnas de solo un valor

La función `deleteOnlyOnesColumns` tiene como objetivo eliminar del DataFrame (`df`) las columnas que contienen un único valor distinto, excluyendo las columnas `'encounter_id'` y `'patient_nbr'` ya que son identificadores. Eliminar estas columnas es importante porque no aportan información variada o útil para el análisis de datos o la modelización. En el machine learning, 'las características que no varían no influyen en la predicción', y su presencia puede incluso aumentar innecesariamente la complejidad del modelo y el tiempo de procesamiento.



```
def deleteOnlyOnesColumns(df):  
    col_distinct_val_check = [col for col in df.columns if col not in ['encounter_id', 'patient_nbr']]  
    for col in col_distinct_val_check:  
        print(f'Column {col} has {set(df[col])} distinct values')  
        if len(set(df[col])) == 1:  
            print(f'Column {col} has only one value')  
            df.drop(col, axis=1, inplace=True)  
    return df  
  
df = deleteOnlyOnesColumns(df)
```

0.3s Python

Figura 2

2.4. Eliminación de columnas con valores repetidos

El objetivo de la función `deleteHighlyRepeatedColumns` es eliminar del DataFrame (`df`) las columnas que contienen valores repetidos. Eliminar estas columnas es importante porque no aportan información variada o útil para el análisis de datos o la modelización. En el machine learning, ‘las características que no varían no influyen en la predicción’, y su presencia puede incluso aumentar innecesariamente la complejidad del modelo y el tiempo de procesamiento.



```
def deleteHighlyRepeatedColumns(df, threshold):  
    total_rows = len(df)  
    columns_to_delete = []  
  
    for col in df.columns:  
        value_counts = df[col].value_counts()  
        most_common_value_count = value_counts.iloc[0]  
        most_common_value_percentage = most_common_value_count / total_rows  
  
        if most_common_value_percentage > threshold:  
            columns_to_delete.append(col)  
  
    df.drop(columns_to_delete, axis=1, inplace=True)  
  
    return df  
  
df = deleteHighlyRepeatedColumns(df, 0.75)
```

0.1s Python

Figura 3

2.5. Sacar porcentajes de nulos de columnas

A continuación, vamos a sacar los porcentajes de nulos de filas por cada Columna. Estos porcentajes nos ayudarán a detectar de manera visual las columnas que tienen un porcentaje de nulos muy alto y que por tanto, deberíamos eliminar o tratar de alguna manera.

```

# Itera sobre las columnas del dataframe y verifica si el tipo de dato es objeto.
# Calcula la proporción de valores nulos y la proporción de valores que son '?'
for col in df.columns:
    if df[col].dtype == object:
        proportion_null = df[col].isna().sum()/len(df)
        print(col, f'{proportion_null * 100:.4f}%')

```

✓ 0.0s Python

```

race 2.2336%
gender 0.0000%
age 0.0000%
weight 96.8585%
payer_code 39.5574%
medical_specialty 49.0822%
diag_1 0.0206%
diag_2 0.3518%
diag_3 1.3983%
insulin 0.0000%
change 0.0000%
readmitted 0.0000%

```

Figura 4

Con esta gráfica podemos ver que las columnas 'weight', 'payer_code' y 'medical_specialty' tienen un porcentaje de nulos muy alto, por lo que las eliminaremos.

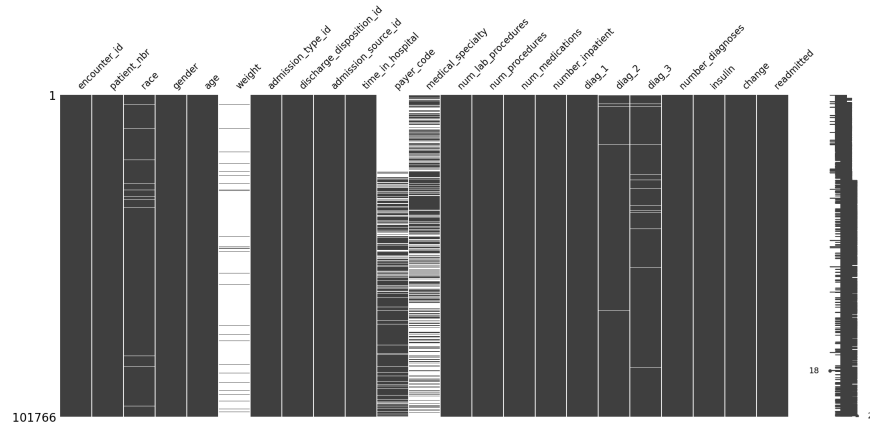


Figura 5

2.6. Eliminación de columnas con alta proporción de nulos

Esta función, *deleteNullColumns*, tiene como objetivo identificar y eliminar columnas de un DataFrame '(df)' de Pandas que tengan un alto porcentaje de valores nulos o faltantes. Eliminar columnas con una alta proporción de valores nulos es crucial en el análisis de datos y el modelado predictivo, porque un exceso de valores faltantes puede distorsionar el análisis y afectar la precisión de los modelos. Al eliminar estas columnas, se mejora la calidad de los datos, lo que conduce a análisis más precisos y fiables.

```
def deleteNullColumns(df, threshold):
    null_counts = df.isnull().sum()
    nan_counts = df.isna().sum()
    total_counts = len(df)

    column_counts = {
        'Null Counts': null_counts,
        'NaN Counts': nan_counts
    }

    df_counts = pd.DataFrame(column_counts)

    # Calculate percentage of nulls and nans
    df_counts['Null Percentage'] = round((null_counts / total_counts) * 100, 3)
    df_counts['NaN Percentage'] = round((nan_counts / total_counts) * 100, 3)

    # Eliminate rows with only zeros
    df_counts = df_counts[(df_counts == 0).all(axis=1)]

    # Delete columns with null percentage above the threshold
    columns_to_delete = df_counts[df_counts['Null Percentage'] >= threshold].index
    for column in columns_to_delete:
        print(f"Dropping column: {column}")
    df = df.drop(columns_to_delete, axis=1)

    return df

df = deleteNullColumns(df, 15)
✓ 0.0s Python
```

Dropping column: weight
Dropping column: payer_code
Dropping column: medical_specialty

Figura 6

2.7. Eliminación de filas con valores nulos

Elimina todas las filas del DataFrame `df` que contienen al menos un valor nulo o faltante. Esta acción es importante porque los valores nulos pueden distorsionar el análisis de datos y afectar negativamente el rendimiento de los modelos de machine learning. Al eliminar filas con valores nulos, se mejora la calidad y la fiabilidad de los datos, lo que conduce a análisis más precisos y fiables.

```
# Eliminando columnas que no aportan información
df.dropna(inplace=True)

# Volvemos a eliminar columnas que no aportan información después de eliminar los nulos
df = deleteOnlyOnesColumns(df)
✓ 0.1s Python
```

Figura 7

2.8. Eliminación de columnas IDs

1. Irrelevancia para el Análisis o Modelado: Estas columnas pueden contener identificadores únicos o códigos que no aportan información útil para el análisis o la modelización. Por ejemplo, identificadores de pacientes o encuentros no influyen en patrones o tendencias en los datos de salud.
2. Reducción de Dimensionalidad: Eliminar columnas no relevantes reduce la dimensionalidad del conjunto de datos, lo que puede mejorar la eficiencia del procesamiento y

la claridad del análisis.

3. Evitar Sesgo: Algunas de estas columnas pueden introducir sesgos o no ser representativas de la población o del fenómeno que se está estudiando.

```
df.drop(['encounter_id', 'patient_nbr', 'admission_type_id', 'discharge_disposition_id', 'admission_source_id'], axis=1, inplace=True)
```

Figura 8

2.9. Tratamiento individual de campos previamente estudiados

Como hemos visto en el apartado anterior, hay columnas que tienen un porcentaje de nulos. En este apartado, vamos a tratar de manera individual cada una de estas columnas para ver si podemos eliminarlas o tratarlas de alguna manera.

```
df['gender'].value_counts(dropna=False)
```

Female	52833
Male	45219
Unknown/Invalid	1

Name: gender, dtype: int64

Figura 9

Al tener un porcentaje muy bajo de registros con valores nulos, vamos a eliminar las filas que contengan estos valores.

```
exclude_indexes = df[df['gender'] == 'Unknown/Invalid'].index.tolist()
required_indexes = [index for index in df.index.tolist() if index not in list(set(exclude_indexes))]
df = df.loc[required_indexes]
```

Figura 10

2.9.1. Estudio de valores y codificación de los mismos

Como hemos visto en apartados anteriores hay columnas que tienen rangos de valores definidos ya sean tipo de 'string' u 'objeto'. Lo que realizaremos ahora será la codificación de estos valores para que sean numéricos y así poder trabajar con ellos. Estas columnas contienen datos categóricos que deben ser convertidos a un formato numérico para que puedan ser procesados por algoritmos de machine learning. LabelEncoder transforma cada

categoría única en un número entero. Esto es crucial porque muchos modelos de machine learning no pueden trabajar directamente con datos no numéricos y requieren que todos los inputs sean numéricos.

Esta transformación convierte las categorías en números enteros.

```
# Transformamos en valores categoricos
df['diag_1'] = df['diag_1'].astype('category').cat.codes
df['diag_2'] = df['diag_2'].astype('category').cat.codes
df['diag_3'] = df['diag_3'].astype('category').cat.codes

# Transformamos en valores categoricos
df['race'] = df['race'].astype('category').cat.codes
```

Figura 11

LabelEncoder, asignaría un valor entero único a cada categoría distinta sin considerar la lógica subyacente o el significado especial de las categorías. Por lo tanto, este enfoque de codificación manual es más apropiado para mantener la interpretación deseada de los datos en estas columnas.

```
# Esta es la variable que queremos predecir
df['readmitted'] = df['readmitted'].replace('<30', 1)
df['readmitted'] = df['readmitted'].replace('>30', 1)
df['readmitted'] = df['readmitted'].replace('NO', 0)

def transform_medications(df):
    medications = ['insulin']

    for col in medications:
        df[col] = df[col].replace('No', 0)
        df[col] = df[col].replace('Steady', 1)
        df[col] = df[col].replace('Up', 1)
        df[col] = df[col].replace('Down', 1)

    return df

df = transform_medications(df)
```

Figura 12

2.10. Volvemos a sacar un el número de filas y columnas

Al haber eliminado los registros nulos y las columnas que no nos aportaban información, volvemos a sacar el número de filas y columnas que tenemos en el dataset. Como podemos ver, hemos pasado de tener 101.766 filas y 47 columnas a tener 98052 filas y 15 columnas.

Para ver que el dataset no esta sesgado y que tenemos un número similar de registros para cada clase, vamos a sacar un gráfico de barras y otro de tarta. Como podemos ver, el dataset no esta sesgado y tenemos un número similar de registros para cada clase.

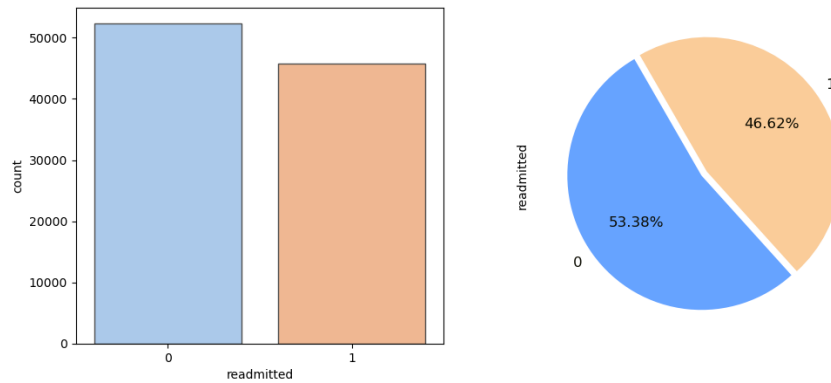


Figura 13

2.11. Tipos de datos tras la transformación

Tras la transformación y limpieza del dataset las variables seleccionadas para los entrenamientos son las que se describen en la siguiente imagen:

race	int8
gender	int64
age	int64
time_in_hospital	int64
num_lab_procedures	int64
num_procedures	int64
num_medications	int64
number_inpatient	int64
diag_1	int16
diag_2	int16
diag_3	int16
number_diagnoses	int64
insulin	int64
change	int64
readmitted	int64
dtype:	object

Figura 14

2.12. Construcción de Datasets

División en Conjuntos de Entrenamiento y Prueba: Se utiliza 'train_test_split' para dividir 'x (características)' y 'y (variable objetivo)' en conjuntos de entrenamiento y prueba. 'El test_size = 0.3' indica que el 30% de los datos se usarán para el conjunto de prueba.

ba, y ‘train_size = 0.7’ que el 70 % restante se utilizará para entrenamiento. El parámetro ‘random_state = 42’ asegura la reproducibilidad del split.

```
x = df.copy()
x.drop(['readmitted'], axis=1, inplace=True)
y = df['readmitted'].copy()
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state = 42, test_size = 0.3, train_size = 0.7)
✓ 0.0s
```

Figura 15

2.13. Prueba de clasificadores

La validación cruzada es importante porque proporciona una evaluación más robusta del modelo, reduciendo la variabilidad asociada con una única división de los datos en conjuntos de entrenamiento y prueba. Además, la validación cruzada permite utilizar todos los datos para entrenamiento y prueba, lo que conduce a una mejor precisión del modelo.

Validación cruzada de 10 particiones (10-CV). Esto implica dividir el conjunto de datos en 10 partes, utilizando cada parte como conjunto de prueba una vez y el resto como conjunto de entrenamiento. Se usa ROC AUC (Área Bajo la Curva de Característica de Operación del Receptor) como métrica de rendimiento.

2.13.1. DummyClassifier

Se crea una instancia de ‘DummyClassifier’ con la estrategia ‘most_frequent’, lo que significa que siempre predice la clase más frecuente en el conjunto de entrenamiento.

```
# Crear el DummyClassifier
dummy_clf = DummyClassifier(strategy='most_frequent')

# Usar cross_val_score para evaluar el modelo usando validación cruzada de 10 particiones
# y ROC AUC como métrica de rendimiento
cv_scores_dummy = cross_val_score(dummy_clf, x, y, cv=10, scoring='make_scorer(roc_auc_score)')

# Calcular el promedio de los puntajes de validación cruzada
mean_cv_score_dummy = cv_scores_dummy.mean()
print(f"Promedio de ROC AUC con validación cruzada (10-CV): {mean_cv_score_dummy}")

✓ 0.1s Python
Promedio de ROC AUC con validación cruzada (10-CV): 0.5
```

Figura 16

Un valor de ROC AUC de 0.5 es lo que se esperaría de un modelo que ‘hace predicciones aleatorias’, ya que esta métrica mide la habilidad de un modelo para distinguir entre clases. Por lo tanto, el promedio de ROC AUC de 0.5 confirma que el ‘DummyClassifier’ no tiene capacidad predictiva en este conjunto de datos. Esto establece una línea base muy básica contra la cual se pueden comparar modelos más avanzados.

2.13.2. DecisionTreeClassifier

Este código configura y evalúa un ‘DecisionTreeClassifier’ usando validación cruzada de 10 particiones y la métrica ROC AUC

```
# Crear el DecisionTreeClassifier
dtc = DecisionTreeClassifier()

# Usar cross_val_score para evaluar el modelo usando validación cruzada de 10 particiones
# y ROC AUC como métrica de rendimiento
cv_scores_dtc = cross_val_score(dtc, x, y, cv=10, scoring='roc_auc_score')

# Calcular el promedio de los puntajes de validación cruzada
mean_cv_score_dtc = cv_scores_dtc.mean()
print(f"Promedio de ROC AUC con validación cruzada (10-CV) para Decision Tree Classifier: {mean_cv_score_dtc}")

✓ 4.8s
Promedio de ROC AUC con validación cruzada (10-CV) para Decision Tree Classifier: 0.5379955174845381
```

Figura 17

En su configuración actual, tiene una capacidad limitada para distinguir efectivamente entre clases en este conjunto de datos. Esto puede indicar la necesidad de optimizar.

2.13.3. RandomForestClassifier

Este código configura y evalúa un ‘RandomForestClassifier’ usando validación cruzada de 10 particiones y la métrica ROC AUC

```
# Crear el RandomForestClassifier
rfc = RandomForestClassifier()

# Usar cross_val_score para evaluar el modelo usando validación cruzada de 10 particiones
# y ROC AUC como métrica de rendimiento
cv_scores_rfc = cross_val_score(rfc, x, y, cv=10, scoring='roc_auc_score')
print(cv_scores_rfc)

# Calcular el promedio de los puntajes de validación cruzada
mean_cv_score_rfc = cv_scores_rfc.mean()
print(f"Promedio de ROC AUC con validación cruzada (10-CV) para Random Forest Classifier: {mean_cv_score_rfc}")

✓ 1m 15.2s
[0.593298  0.61122933 0.60193253 0.59438682 0.57945821 0.59532066
 0.61274794 0.61587416 0.62188518 0.61482988]
Promedio de ROC AUC con validación cruzada (10-CV) para Random Forest Classifier: 0.6040881913191326
```

Figura 18

Los valores individuales de ROC AUC en cada partición varían entre aproximadamente 0.575 y 0.625, lo que sugiere una cierta consistencia en el rendimiento a través de diferentes divisiones del conjunto de datos. Estos resultados reflejan una capacidad más robusta del ‘RandomForestClassifier’ para distinguir entre clases en este conjunto de datos específico.

2.14. Obtención de mejores Hiperparámetros a mano

Los ‘hiperparámetros’ son configuraciones externas de los modelos de machine learning que deben establecerse antes de entrenar el modelo. No se aprenden a partir de los datos sino que se eligen manualmente y afectan significativamente el rendimiento del modelo.

Este código realiza una búsqueda exhaustiva para encontrar la mejor combinación de hiperparámetros para un 'DecisionTreeClassifier'. Prueba diferentes valores para 'max_depth', 'min_samples_split', 'min_samples_leaf' y 'max_features', evaluando cada combinación con validación cruzada y la métrica ROC AUC. Finalmente, registra y muestra la mejor combinación de hiperparámetros y el mejor puntaje AUC obtenido. Este proceso ayuda a optimizar el modelo para un rendimiento más eficiente y preciso.

```
# Lista de valores para probar para cada hiperparámetro
max_depth_values = [3, 5, 10, None]
min_samples_split_values = [2, 4, 10]
min_samples_leaf_values = [1, 2, 4]
max_features_values = [None, 'sqrt', 'log2']

best_auc = 0
best_params = {}

# Probando combinaciones de hiperparámetros
for max_depth in max_depth_values:
    for min_samples_split in min_samples_split_values:
        for min_samples_leaf in min_samples_leaf_values:
            for max_features in max_features_values:
                # Crear el modelo con un conjunto de hiperparámetros
                dtc = DecisionTreeClassifier(max_depth=max_depth, min_samples_split=min_samples_split,
                                           min_samples_leaf=min_samples_leaf, max_features=max_features)

                # Usar cross_val_score para evaluar el modelo usando validación cruzada
                cv_scores = cross_val_score(dtc, x, y, cv=10, scoring='roc_auc_score')

                # Calcular el promedio de los puntajes de validación cruzada
                mean_cv_score = cv_scores.mean()

                # Guardar si es el mejor modelo hasta ahora
                if mean_cv_score > best_auc:
                    best_auc = mean_cv_score
                    best_params = {'max_depth': max_depth, 'min_samples_split': min_samples_split,
                                'min_samples_leaf': min_samples_leaf, 'max_features': max_features}

print(f"Mejor AUC promedio con validación cruzada: {best_auc}")
print(f"Mejores parámetros: {best_params}")

Mejor AUC promedio con validación cruzada: 0.6062715733583585
Mejores parámetros: {'max_depth': 5, 'min_samples_split': 2, 'min_samples_leaf': 2, 'max_features': None}
```

Figura 19

Ahora cogemos los mejores hiperparametros y los utilizamos para entrenar el modelo.

```
# Crear el DecisionTreeClassifier
dtc = DecisionTreeClassifier(max_depth= 5, min_samples_split= 2, min_samples_leaf= 1, max_features= None)

# Usar cross_val_score para evaluar el modelo usando validación cruzada de 10 particiones
# y ROC AUC como métrica de rendimiento
cv_scores_dtc_mejoresParametros = cross_val_score(dtc, x, y, cv=10, scoring='roc_auc_score')

# Calcular el promedio de los puntajes de validación cruzada
mean_cv_score_dtc_mejoresParametros= cv_scores_dtc_mejoresParametros.mean()
print(f"Promedio de ROC AUC con validación cruzada (10-CV) para Decision Tree Classifier: {mean_cv_score_dtc_mejoresParametros}")

✓ 12s
```

Figura 20

El promedio de ROC AUC obtenido, 0.60627, indica una mejora significativa en comparación con la versión del DecisionTreeClassifier sin hiperparámetros optimizados. Esto demuestra cómo la selección cuidadosa de hiperparámetros puede aumentar la eficacia de un modelo de machine learning.

2.15. Comparativa de los cuatro clasificadores

Esta gráfica muestra los resultados de los 4 clasificadores obtenidos. Como podemos ver, el mejor clasificador es el RandomForestClassifier con los hiperparámetros optimizados, ya que tiene un valor de 0.6068 de ROC AUC. Pero también podemos ver que el DecisionTreeClassifier con los hiperparámetros optimizados tiene un valor de 0.6062 de ROC AUC, por lo que es muy similar al RandomForestClassifier.

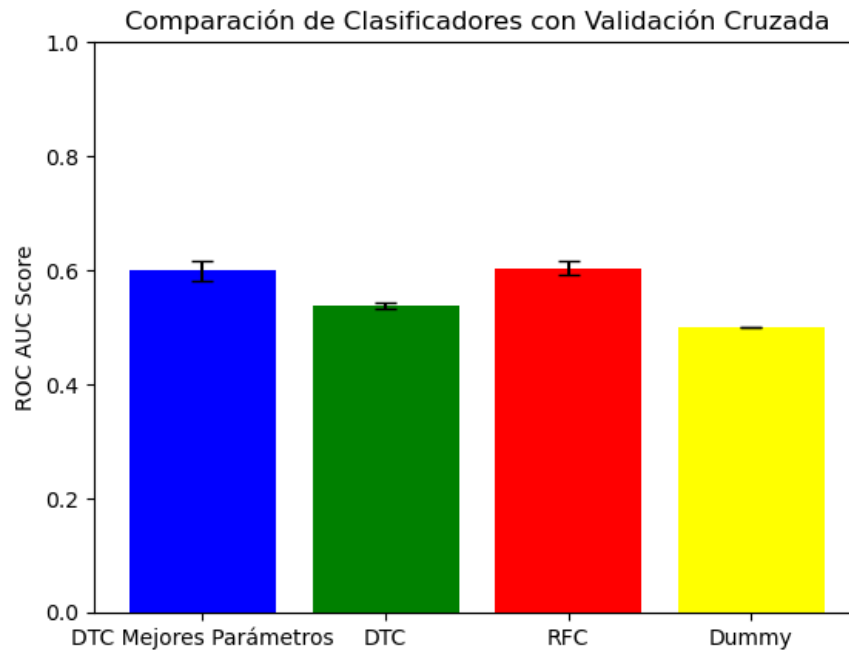


Figura 21

3. Medio

3.1. Selección automática de atributos

Los algoritmos de selección automática de atributos son técnicas de machine learning que eligen de manera automática las características más relevantes de los datos para su uso en modelos predictivos. Estos métodos ayudan a mejorar la eficiencia del modelo y pueden aumentar su precisión al eliminar el ruido o la redundancia en los datos.

El código define una función para evaluar diferentes métodos de selección de atributos aplicados a un conjunto de datos x con la variable objetivo y . La función 'evaluate_feature_selection' aplica un selector de atributos para reducir la dimensionalidad del conjunto de datos y luego evalúa el rendimiento de un modelo RandomForestClassifier con validación cruzada de 10 particiones, utilizando ROC AUC como métrica.

Se prueba un conjunto de selectores: 'umbral de varianza, SelectKBest, SelectFromModel y RFE'. Se almacenan los puntajes de cada método y se imprime el mejor puntaje. Finalmente, se grafican los resultados para comparar visualmente el rendimiento de cada método de selección de atributos. La variable best_x_reduced guarda la versión reducida del conjunto de datos con el mejor puntaje.

```
def evaluate_feature_selection(selector, x, y):  
    # Reducir el conjunto de datos  
    x_reduced = selector.fit_transform(x, y)  
  
    # Crear y entrenar el modelo  
    model = RandomForestClassifier()  
    cv_scores = cross_val_score(model, x_reduced, y, cv=10, scoring=make_scorer(roc_auc_score))  
  
    # Calcular el promedio de los puntajes de validación cruzada  
    return cv_scores.mean(), x_reduced  
  
# Métodos de selección de atributos  
selectors = [  
    ('Variance Threshold', VarianceThreshold(threshold=0.5)),  
    ('SelectKBest', SelectKBest(chi2, k=10)),  
    ('SelectFromModel', SelectFromModel(RandomForestClassifier(n_estimators=100))),  
    ('RFE', RFE(LogisticRegression(max_iter=1000), n_features_to_select=10))  
]
```

Figura 22

```

# Evaluar cada selector
scores = []
best_x_reduced = None
bestScore = 0
for name, selector in selectors:
    score, x_reduced = evaluate_feature_selection(selector, x, y)
    if score > bestScore:
        best_x_reduced = x_reduced
        scores.append(score)
    print(f'{name}: ROC AUC = {score}')
```

```

# Graficar los resultados
plt.bar(range(len(selectors)), scores, tick_label=[name for name, _ in selectors])
plt.ylabel('ROC AUC Score')
plt.title('Comparación de Métodos de Selección de Atributos')
plt.show()
```

✓ 6m 12.2s Python

Variance Threshold: ROC AUC = 0.5977531507015021
 SelectKBest: ROC AUC = 0.5863625702556542
 SelectFromModel: ROC AUC = 0.5492435562788656
[/Users/daniel/anaconda3/lib/python3.11/site-packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning](#)
 STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
<https://scikit-learn.org/stable/modules/preprocessing.html>
 Please also refer to the documentation for alternative solver options:
https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
 n_iter_i = _check_optimize_result(
 RFE: ROC AUC = 0.5697194764118383

Figura 23

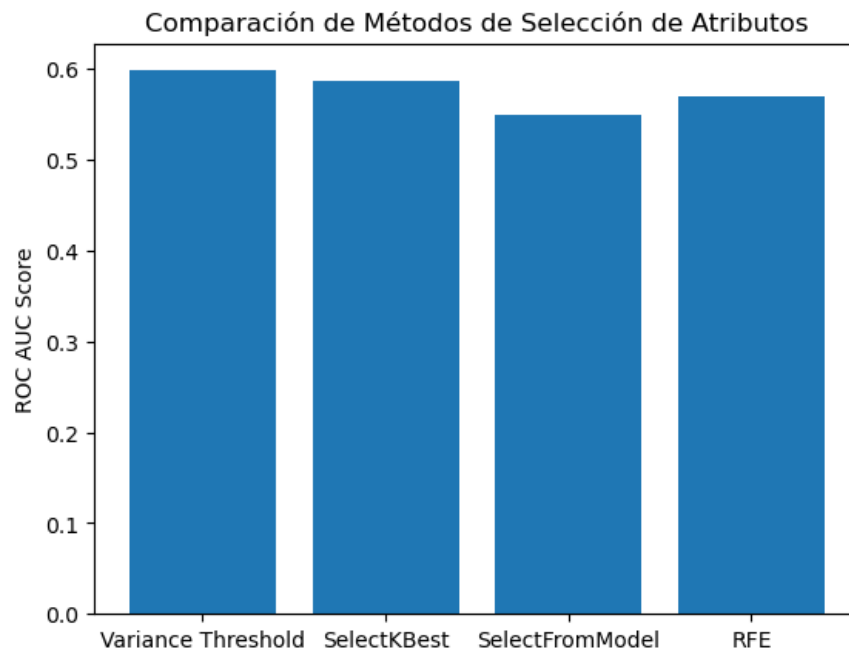


Figura 24

Los resultados indican que el método ‘RFE (Recursive Feature Elimination)’ obtuvo la mejor puntuación ROC AUC, seguido muy de cerca por Variance Threshold. Esto sugiere que el RFE fue ligeramente más efectivo para seleccionar características que contribuyen

positivamente a la capacidad predictiva del modelo. SelectKBest y SelectFromModel tuvieron un desempeño inferior en comparación con los otros dos métodos. Estos resultados pueden informar qué técnica de selección de atributos podría ser preferible en esta situación particular para mejorar la precisión del modelo.

3.2. Reducción de la dimensionalidad

El código realiza una reducción de dimensionalidad utilizando Análisis de Componentes Principales (PCA) después de estandarizar el conjunto de datos x . PCA se configura para mantener el 90 % de la varianza original del conjunto de datos, lo que significa que seleccionará el número de componentes principales que suman hasta el 90 % de la varianza total. Luego, divide los datos transformados por PCA en conjuntos de entrenamiento y prueba.

```
# Estandarizar los datos antes de aplicar PCA
scaler = StandardScaler()
X_scaled = scaler.fit_transform(x)

# Aplicar PCA
pca = PCA(n_components=0.90) # Conserva el 90% de la varianza
X_pca = pca.fit_transform(X_scaled)

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train_pca, X_test_pca, Y_train, Y_test = train_test_split(X_pca, y, test_size=0.3, random_state=42)

print(f"Original shape: {x.shape}")
print(f"Reduced shape: {X_pca.shape}")
print(f"Number of components: {pca.n_components_}")

✓ 0.0s Python
```

Original shape: (98852, 14)
Reduced shape: (98852, 12)
Number of components: 12

Figura 25

Como podemos observar el número de características ha pasado de las 14 originales a 12. Con estas nuevas características se van a probar nuevos modelos y los anteriores.

3.3. Prueba de 3 Modelos nuevos y comparativa de los mismos con el mejor selector automatico

En esta sección, se explorará la eficacia de tres modelos de machine learning adicionales: Regresión Logística, K-Nearest Neighbors y Gaussian Naive Bayes. Estos modelos se compararán con los ya evaluados (Random Forest, Decision Tree y DummyClassifier) utilizando el mejor método de selección automática de atributos identificado previamente. La Regresión Logística se optimizará aumentando el número máximo de iteraciones a 1000 para garantizar la convergencia. K-Nearest Neighbors y Gaussian Naive Bayes se añaden por sus características únicas en el manejo de relaciones no lineales y probabilidades, respectivamente. Esta comparativa nos permitirá entender mejor el rendimiento relativo de una variedad de enfoques en la tarea de predecir readmisiones en pacientes con diabetes.

```
# Crear los modelos
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "K-Nearest Neighbors": KNeighborsClassifier(),
    "GaussianNB": GaussianNB(),
    "Random Forest Classifier": RandomForestClassifier(),
    "Decision Tree Classifier": DecisionTreeClassifier(max_depth=None, min_samples_split=4, min_samples_leaf=1, max_features=None),
    "DummyClassifier": DummyClassifier(strategy='most_frequent')
}
✓ 0.0s
```

Figura 26

Los resultados de la evaluación de modelos muestran que la Regresión Logística tuvo el mejor rendimiento con un promedio de ROC AUC de aproximadamente 0.593, seguido por el Random Forest Classifier con 0.585. Gaussian Naive Bayes y K-Nearest Neighbors tuvieron un rendimiento inferior, con puntuaciones de 0.575 y 0.554 respectivamente. El Decision Tree Classifier, aunque optimizado, obtuvo un resultado más bajo de 0.533. El DummyClassifier, utilizado como línea base, se mantuvo en 0.5, como se esperaba. Estos resultados subrayan la eficacia relativa de diferentes enfoques algorítmicos en la tarea de predecir readmisiones en pacientes con diabetes.

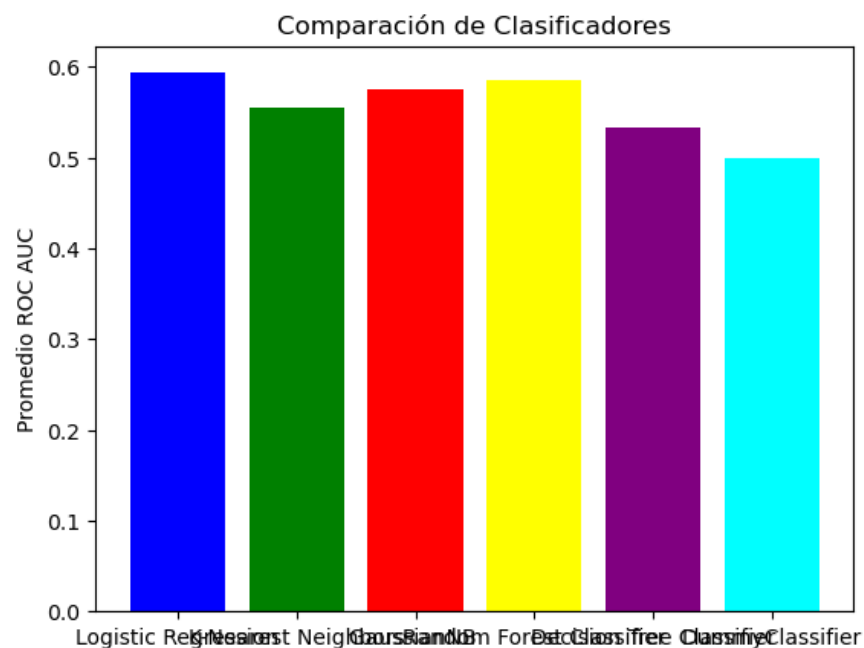


Figura 27

3.4. Prueba de Wilcoxon

Este tipo de prueba es útil para comparar dos conjuntos de resultados y determinar si hay una diferencia significativa entre ellos.

Explicación: Supongamos que tienes dos conjuntos de puntajes de validación cruzada (por ejemplo, ROC AUC scores) de dos modelos diferentes y quieres saber si hay una diferencia estadísticamente significativa en su rendimiento.



Figura 28

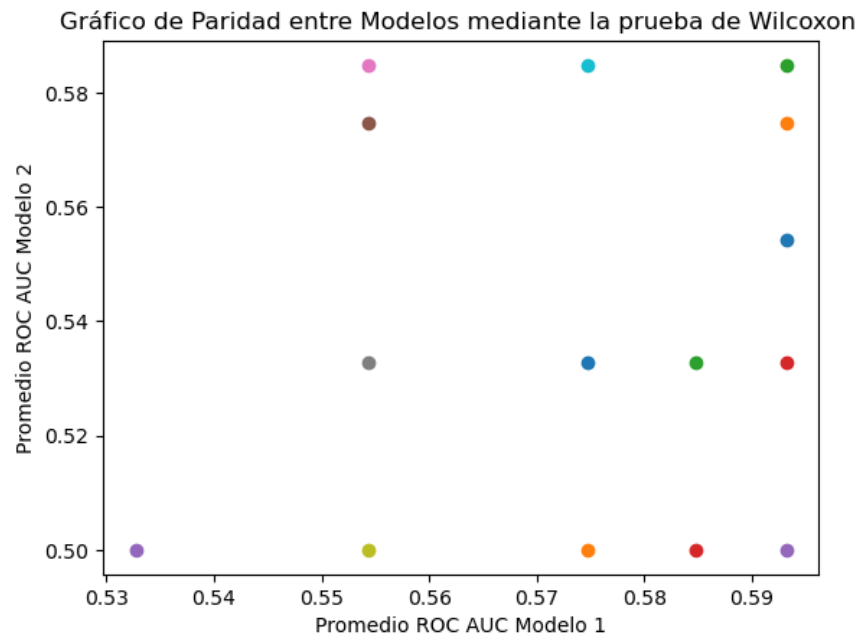


Figura 29

4. Avanzado

4.1. Buscar parámetros óptimos de forma automática

4.1.1. GridSearchCV

GridSearchCV es una técnica de optimización de hiperparámetros en machine learning. Funciona realizando una búsqueda exhaustiva sobre un espacio de hiperparámetros especificados. Para cada combinación de hiperparámetros, GridSearchCV entrena un modelo y evalúa su rendimiento mediante validación cruzada. Finalmente, selecciona la combinación que ofrece el mejor rendimiento según la métrica de evaluación elegida.

Aplicación de GridSearchCV a DTC: Como se puede apreciar en la imagen se han obtenido los mejores hiperparámetros entre los configurados.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, roc_auc_score

# Definir el espacio de hiperparámetros
param_grid = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 4, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [None, 'sqrt', 'log2']
}

# Crear el clasificador de árbol de decisión
dtc = DecisionTreeClassifier()

# Crear un objeto GridSearchCV
grid_search = GridSearchCV(estimator=dtc, param_grid=param_grid,
                           cv=10, scoring=make_scorer(roc_auc_score))

# Ajustar GridSearchCV a los datos
grid_search.fit(x_train, y_train)

# Imprimir los mejores parámetros y el mejor score
print(f"Mejores parámetros: {grid_search.best_params_}")
print(f"Mejor ROC AUC: {grid_search.best_score_}")

✓ 1m 28.2s Python
Mejores parámetros: {'max_depth': 3, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
Mejor ROC AUC: 0.6005830261372835
```

Figura 30

4.1.2. BayesianOptimization

BayesianOptimization es un enfoque para la optimización de hiperparámetros que utiliza modelos probabilísticos para predecir el rendimiento de un modelo con diferentes hiperparámetros. Este método se basa en el Teorema de Bayes para actualizar la probabilidad de cada conjunto de hiperparámetros según su rendimiento observado. A diferencia de las búsquedas exhaustivas o aleatorias, la optimización bayesiana es más eficiente, ya que utiliza información de intentos anteriores para mejorar la búsqueda. Se destaca por su habilidad para manejar espacios de búsqueda complejos y por encontrar buenos hiperparámetros en menos tiempo.

```

def dtc_cv(max_depth, min_samples_split, min_samples_leaf, max_features):
    dtc = DecisionTreeClassifier(
        max_depth=int(max_depth),
        min_samples_split=int(min_samples_split),
        min_samples_leaf=int(min_samples_leaf),
        max_features=max_features
    )
    cv_scores = cross_val_score(dtc, x, y, cv=10, scoring='roc_auc')
    return np.mean(cv_scores)

pbounds = {
    'max_depth': (1, 10),
    'min_samples_split': (2, 10),
    'min_samples_leaf': (1, 5),
    'max_features': (0.1, 1) # Porcentaje del número total de características
}

optimizer = BayesianOptimization(f=dtc_cv, pbounds=pbounds, random_state=1)
optimizer.maximize()

print(optimizer.max)

```

✓ 35.7s Python

Figura 31

Estas son las iteraciones realizadas por el algoritmo donde se pueden apreciar los hiperparámetros configurados

iter	target	max_depth	max_fe...	min_sa...	min_sa...
1	0.6386	0.753	0.7083	1.0	4.419
2	0.5592	2.321	0.1831	1.745	4.764
3	0.6117	4.571	0.5849	2.677	7.482
4	0.6264	2.84	0.8903	1.11	7.364
5	0.6337	4.756	0.6028	1.562	3.585
6	0.6394	6.368	0.9684	1.494	4.246
7	0.6399	6.447	1.0	1.0	6.691
8	0.6388	5.423	1.0	1.0	9.63
9	0.6408	7.991	1.0	2.763	8.828
10	0.6662	9.523	0.1	1.0	7.199
11	0.6399	6.01	1.0	3.72	10.0
12	0.6409	7.077	1.0	5.0	7.577
13	0.6396	8.765	1.0	5.0	10.0
14	0.6409	7.429	1.0	5.0	4.164
15	0.5932	7.826	0.1	1.358	2.0
16	0.6399	6.669	1.0	3.506	5.855
17	0.6371	9.242	0.6384	4.967	6.06
18	0.6281	2.466	1.0	2.548	10.0
19	0.6388	5.25	1.0	5.0	4.943
20	0.5594	3.973	0.1	5.0	10.0
21	0.6032	10.0	0.1	5.0	8.367
22	0.5831	6.928	0.1	2.185	10.0
23	0.6389	6.581	0.9987	4.511	4.973
...					
29	0.6388	5.218	1.0	1.0	7.948
30	0.6408	7.774	1.0	2.316	5.212

{'target': 0.6408554195217910, 'params': {'max_depth': 7.077133559954514, 'max_features': 1.0, 'min_samples_leaf': 5.0, 'min_samples_split': 7.576901195001283}}

Figura 32

4.2. Prueba de Xgboost

XGBoost (eXtreme Gradient Boosting) es un algoritmo de machine learning muy eficiente y popular que se utiliza para tareas de clasificación y regresión. Utiliza un enfoque de boosting de gradientes, donde construye modelos de manera secuencial corrigiendo los errores de los modelos anteriores. XGBoost es conocido por su rendimiento, velocidad y capacidad para manejar una gran cantidad de datos.

```
# Crear una instancia del clasificador XGBoost
xgb_clf = xgb.XGBClassifier()

# Ajustar el modelo a los datos de entrenamiento
xgb_clf.fit(x_train, y_train)

# Evaluar el modelo usando validación cruzada y ROC AUC como métrica
cv_scores = cross_val_score(xgb_clf, x, y, cv=10, scoring=make_scorer(roc_auc_score))

# Calcular el promedio de los puntajes de validación cruzada
mean_cv_score = cv_scores.mean()
print(f"Promedio de ROC AUC: {mean_cv_score}")
```

✓ 3.2s Python

Promedio de ROC AUC: 0.6078390487267271

Figura 33

4.2.1. BayesianOptimization a Xgboost

Aplicamos BayesianOptimization para encontrar los mejores hiperparámetros para un modelo XGBoost. Se define una función de puntuación que entrena un modelo XGBoost con un conjunto de hiperparámetros dados y devuelve el puntaje ROC AUC promedio de la validación cruzada de 10 particiones. Luego, se define un espacio de búsqueda para los hiperparámetros y se ejecuta la optimización bayesiana. Finalmente, se imprimen los mejores hiperparámetros y el mejor puntaje ROC AUC obtenido.

```
def xgb_cv(n_estimators, max_depth, learning_rate):
    model = xgb.XGBClassifier(
        n_estimators=int(n_estimators),
        max_depth=int(max_depth),
        learning_rate=learning_rate
    )
    cv_scores = cross_val_score(model, x_train, y_train, cv=5, scoring='roc_auc')
    return np.mean(cv_scores)

# Definir límites de hiperparámetros
pbounds = {
    'n_estimators': (100, 1000),
    'max_depth': (3, 10),
    'learning_rate': (0.01, 0.3)
}

optimizer = BayesianOptimization(f=xgb_cv, pbounds=pbounds, random_state=1)
optimizer.maximize()

for i, res in enumerate(optimizer.res):
    print("Iteración {}: \n\t{}".format(i, res))
```

✓ 2m 6.4s Python

Figura 34

4.3. Comparativa de todos los clasificadores desarrollados

Los resultados indican que XGBoost tuvo el mejor rendimiento con un promedio de ROC AUC de 0.613, seguido por RandomForestClassifier con 0.598 y Logistic Regression con 0.594. GaussianNB y Decision Tree Classifier tuvieron un rendimiento moderado, mientras que K-Nearest Neighbors obtuvo el resultado más bajo entre los modelos avanzados. El DummyClassifier, como era de esperar, se mantuvo en 0.5, sirviendo como un punto de referencia básico. Estos resultados sugieren que XGBoost es el modelo más efectivo en este

caso para la clasificación, mostrando una capacidad superior para predecir correctamente las clases.

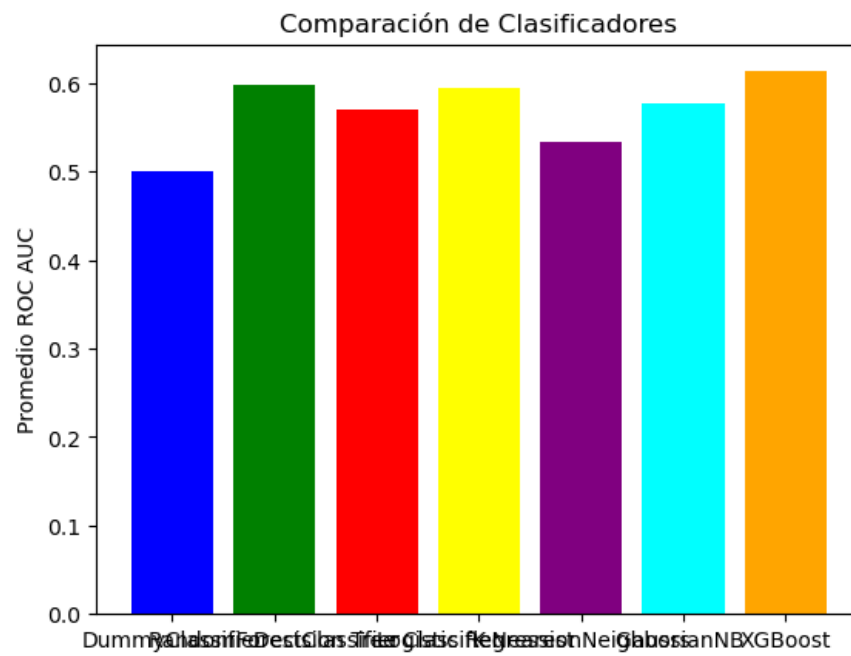


Figura 35

5. Conclusiones

La práctica demostró que los modelos avanzados de machine learning, especialmente XG-Boost, son eficaces en predecir readmisiones de pacientes con diabetes, superando a métodos más simples como Logistic Regression y K-Nearest Neighbors. La optimización bayesiana se destacó como una herramienta poderosa para la selección de hiperparámetros, mejorando significativamente el rendimiento de los modelos. Estos resultados enfatizan la importancia de la elección y afinación cuidadosa de modelos en aplicaciones clínicas. Sin embargo, el alcance de esta práctica se limitó a un conjunto de datos específico, sugiriendo la necesidad de pruebas adicionales en entornos más variados para validar la generalización de estos hallazgos.