wwtcdct23

September 12, 2023

1 Momento de Retroalimentación: Módulo 2 Análisis y Reporte sobre el desempeño del modelo

1.1 Matrícula: A01652281

1.2 Nombre: Michael Steven Delgado Caicedo

2 Justificación selección del dataset

Este dataset fue elegido ya que se tiene una cantidad razonable de datos, que no nos afecta tanto al dividir en *Train*, *Test*, *Validation*. Al igual, nos arroja un resultado binario de 0 y 1. Es una base de datos limpia que nos ahorrará el tiempo. Al ser una base de datos médica y binaria, también nos ayuda a aplicar Random Forest y que nos de el mejor modelo posible.

El dataset contiene una cantidad adecuada de datos, lo que nos permite realizar una división en conjuntos de entrenamiento, prueba y validación sin comprometer significativamente la calidad del modelo. Esto es fundamental para evaluar la capacidad de generalización del modelo.

La naturaleza binaria de los resultados (0 y 1) simplifica el problema y es adecuada para aplicar algoritmos como Random Forest. Esto facilita la tarea de construir un modelo de clasificación.

La limpieza de datos es esencial para evitar problemas de calidad de datos que puedan afectar negativamente el rendimiento del modelo. El hecho de que la base de datos sea limpia ahorra tiempo y esfuerzo en la preparación de datos, permitiéndonos centrarnos en el desarrollo del modelo.

La combinación de un tamaño razonable de datos, resultados binarios y limpieza de datos nos brinda un terreno sólido para evaluar la capacidad de generalización del modelo. Podremos entrenar y validar el modelo de manera efectiva para garantizar que funcione bien en datos no vistos.

[72]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

[73]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
```

```
from sklearn.datasets import fetch_openml

data = load_breast_cancer()
X = data.data  # Variables predictoras
y = data.target  # Variable objetivo (0: benigno, 1: maligno)
print("Dimensiones de X:", X.shape)
print("Dimensiones de y:", y.shape)
```

```
Dimensiones de X: (569, 30)
Dimensiones de y: (569,)
```

3 Separación y evaluación del modelo con un conjunto de prueba y un conjunto de validación (Train/Test/Validation)

En esta sección se ve como un 80% de los datos se ocuparon para el Train, 10% para test y otro 10% para validation.

[74]:
```
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split


# Dividir el conjunto de datos en entrenamiento, prueba y validación
# Primero, dividimos en entrenamiento y prueba (80% entrenamiento, 20% prueba)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Luego, dividimos el conjunto de "temp" en prueba y validación (50% prueba,␣
 ↪50% validación)
X_test, X_val, y_test, y_val = train_test_split(X_temp, y_temp, test_size=0.5,␣
 ↪random_state=42)
```

[75]:
```
X_train.shape, X_temp.shape, y_train.shape, y_temp.shape
```

[75]:
```
((455, 30), (114, 30), (455,), (114,))
```

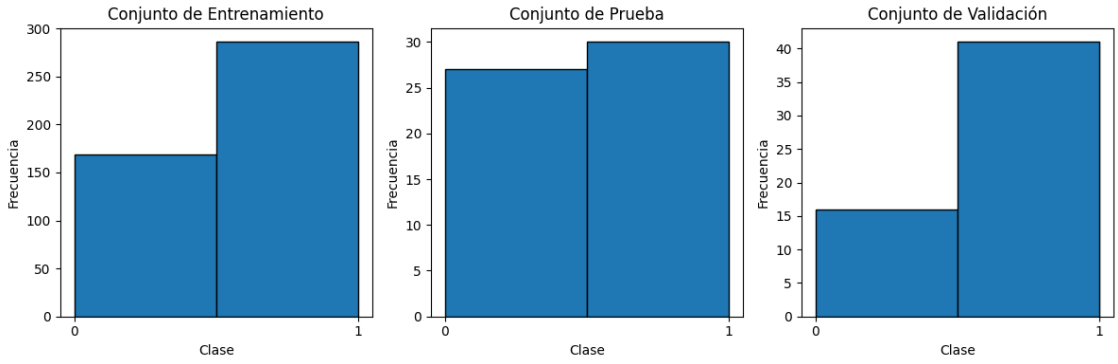
[76]:
```
X_test.shape, X_val.shape, y_test.shape, y_val.shape
```

[76]:
```
((57, 30), (57, 30), (57,), (57,))
```

Con esto demostramos que quedaron dos subconjuntos de datos iguales.

[77]:
```
# Gráfico para mostrar la distribución de las etiquetas en los conjuntos de␣
 ↪datos
plt.figure(figsize=(12, 4))
plt.subplot(131)
plt.hist(y_train, bins=[0, 0.5, 1], edgecolor='k')
```

```
plt.title('Conjunto de Entrenamiento')
plt.xlabel('Clase')
plt.ylabel('Frecuencia')
plt.xticks([0, 1], ['0', '1'])

plt.subplot(132)
plt.hist(y_test, bins=[0, 0.5, 1], edgecolor='k')
plt.title('Conjunto de Prueba')
plt.xlabel('Clase')
plt.ylabel('Frecuencia')
plt.xticks([0, 1], ['0', '1'])

plt.subplot(133)
plt.hist(y_val, bins=[0, 0.5, 1], edgecolor='k')
plt.title('Conjunto de Validación')
plt.xlabel('Clase')
plt.ylabel('Frecuencia')
plt.xticks([0, 1], ['0', '1'])

plt.tight_layout()
plt.show()
```



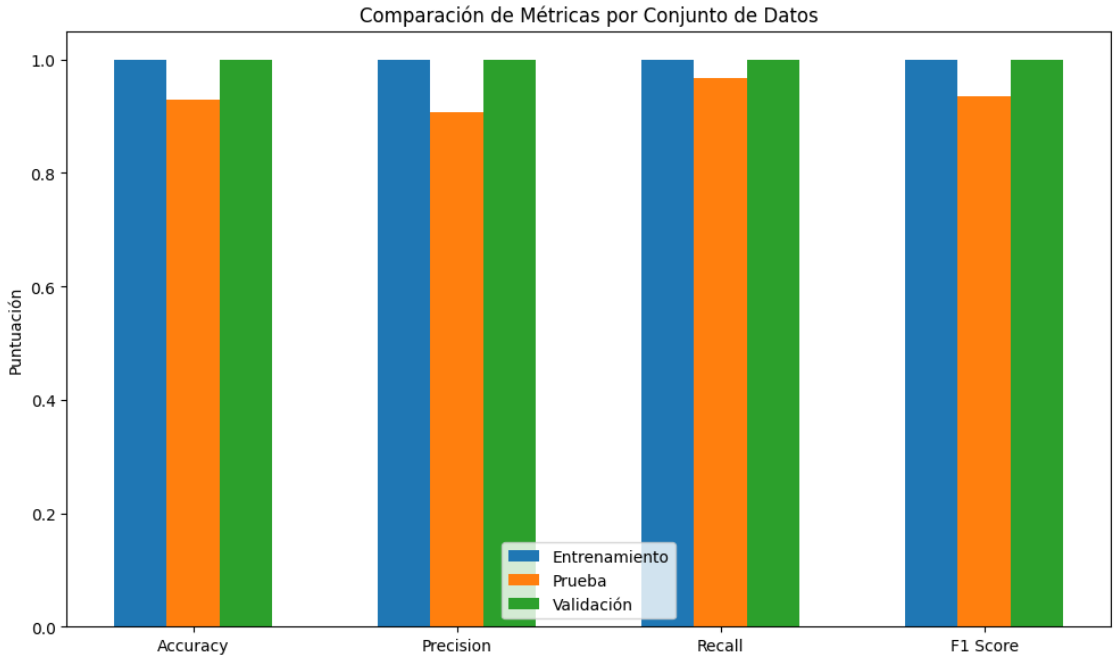
4 Diagnóstico y explicación el grado de varianza, el nivel de ajuste del modelo y el grado de bias o sesgo.

[78]:
```
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix

# Ahora tienes tres conjuntos de datos: entrenamiento, prueba y validación
# X_train, y_train -> Conjunto de entrenamiento
# X_test, y_test -> Conjunto de prueba
```

```
# X_val, y_val -> Conjunto de validación

# Crear y entrenar el modelo Random Forest

rf_classifier = RandomForestClassifier(n_estimators=100, max_depth=100,␣
 ↪random_state=42)
rf_classifier.fit(X_train, y_train)



"""
n_estimators determina la cantidad de árboles en el bosque. Un valor más alto␣
 ↪generalmente mejora la capacidad de generalización del modelo, ya que se␣
 ↪promedian más decisiones de árboles individuales. Sin embargo, aumentar␣
 ↪n_estimators también aumenta el tiempo de entrenamiento y el uso de memoria.
Elegir 100 árboles (n_estimators=100) es una elección común en la práctica, ya␣
 ↪que generalmente proporciona un buen equilibrio entre rendimiento y tiempo␣
 ↪de entrenamiento. Es un valor que tiende a funcionar bien en una amplia gama␣
 ↪de problemas.

random_state es una semilla aleatoria que se utiliza para inicializar el␣
 ↪generador de números aleatorios en el algoritmo. Esto asegura que el␣
 ↪entrenamiento del modelo sea reproducible. Al usar la misma semilla,␣
 ↪obtendrás los mismos resultados en diferentes ejecuciones del código.
Establecer random_state=42 es una elección común, pero la elección de la␣
 ↪semilla puede variar. El número 42 se usa a menudo por convención, pero se␣
 ↪puede utilizar cualquier número entero no negativo.

Entradas ->

n_estimators=100: Es el número de árboles que se crearán en el bosque. En este␣
 ↪caso, se están creando 100 árboles en el bosque.

random_state=42: Es una semilla aleatoria que se utiliza para inicializar el␣
 ↪generador de números aleatorios. Esto asegura que el entrenamiento del␣
 ↪modelo sea reproducible si se utiliza la misma semilla en diferentes␣
 ↪ejecuciones del código.


salida -> rf_classifier: Esta variable es una instancia de la clase␣
 ↪RandomForestClassifier de scikit-learn. Es el modelo de Random Forest que␣
 ↪estás creando y entrenando.

Para entrenar este modelo se usaron las variables de entrada:
```

```
X_train: Es el conjunto de datos de entrenamiento que contiene las variables␣
 ↪predictoras (características) que se utilizarán para entrenar el modelo.␣
 ↪Cada fila de X_train representa una observación y cada columna representa␣
 ↪una característica.

y_train: Es el conjunto de etiquetas objetivo correspondientes a X_train.␣
 ↪Contiene las etiquetas correctas para cada observación en el conjunto de␣
 ↪entrenamiento. Cada etiqueta indica la clase a la que pertenece la␣
 ↪observación (en este caso, 0 para benigno y 1 para maligno).

Después de ejecutar este código, rf_classifier será un modelo de Random Forest␣
 ↪entrenado con 100 árboles en el bosque utilizando los datos de entrenamiento␣
 ↪proporcionados (X_train y y_train). El modelo estará listo para hacer␣
 ↪predicciones en nuevos datos o para evaluar su rendimiento en un conjunto de␣
 ↪prueba, como se hizo en el código anterior.

"""

# Realizar predicciones en los tres conjuntos de datos
y_train_pred = rf_classifier.predict(X_train)
y_test_pred = rf_classifier.predict(X_test)
y_val_pred = rf_classifier.predict(X_val)

# Calcular métricas de rendimiento para cada conjunto
accuracy_train = accuracy_score(y_train, y_train_pred)
accuracy_test = accuracy_score(y_test, y_test_pred)
accuracy_val = accuracy_score(y_val, y_val_pred)

precision_train = precision_score(y_train, y_train_pred)
precision_test = precision_score(y_test, y_test_pred)
precision_val = precision_score(y_val, y_val_pred)

recall_train = recall_score(y_train, y_train_pred)
recall_test = recall_score(y_test, y_test_pred)
recall_val = recall_score(y_val, y_val_pred)

f1_train = f1_score(y_train, y_train_pred)
f1_test = f1_score(y_test, y_test_pred)
f1_val = f1_score(y_val, y_val_pred)

# Gráficos comparativos
metrics = ['Accuracy', 'Precision', 'Recall', 'F1 Score']
train_scores = [accuracy_train, precision_train, recall_train, f1_train]
test_scores = [accuracy_test, precision_test, recall_test, f1_test]
val_scores = [accuracy_val, precision_val, recall_val, f1_val]

x = np.arange(len(metrics))
```

```
width = 0.2

fig, ax = plt.subplots(figsize=(10, 6))
rects1 = ax.bar(x - width, train_scores, width, label='Entrenamiento')
rects2 = ax.bar(x, test_scores, width, label='Prueba')
rects3 = ax.bar(x + width, val_scores, width, label='Validación')

ax.set_ylabel('Puntuación')
ax.set_title('Comparación de Métricas por Conjunto de Datos')
ax.set_xticks(x)
ax.set_xticklabels(metrics)
ax.legend()

plt.tight_layout()
plt.show()
```



" Si las métricas en el conjunto de entrenamiento son significativamente mejores que en los conjuntos de prueba y validación, podría indicar un sesgo alto. Esto significa que el modelo se ha ajustado demasiado a los datos de entrenamiento y no generaliza bien.

" Si las métricas son similares en los tres conjuntos, el modelo podría estar generalizando adecuadamente sin un sesgo evidente.

" Si las métricas en el conjunto de entrenamiento son peores que en los conjuntos de prueba y validación, podría indicar un sesgo bajo. Esto podría deberse a un modelo subajustado".

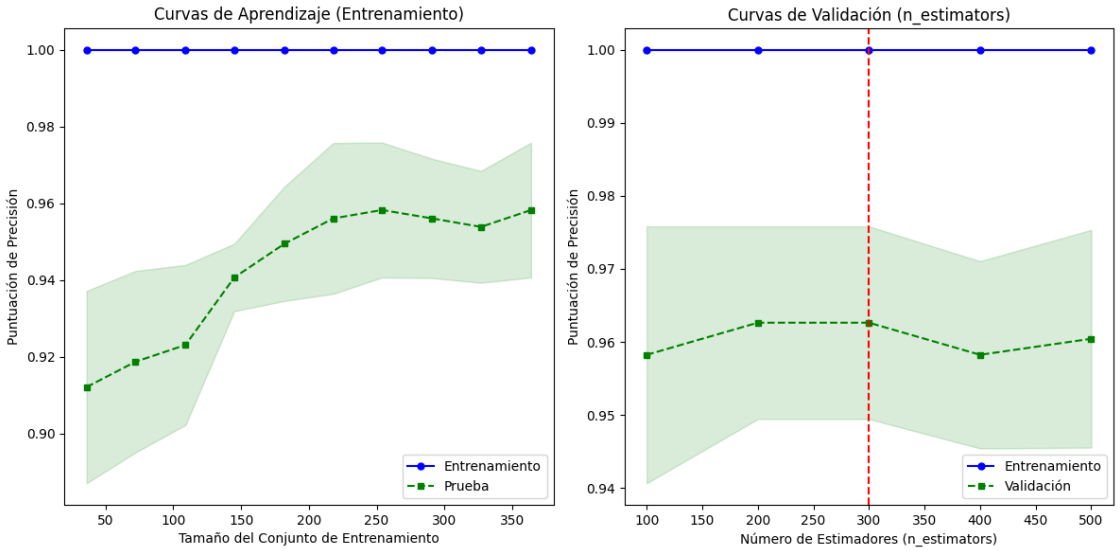
Sabiendo esto, podemos observar un desbalance entre la prueba y la validación, sin embargo, el

sesgo no es significativo entre la separación de los datos. El hecho de tener una separación del *validation* tampoco favorece a que los datos de test estén más desbalanceados, ya que como vemos en la separación de los datos, la repartición entre 0 y 1 es muy similar en *test*. Se puede observar un sesgo mayor en el *precision* pero en cuanto al *recall* y las demás métricas, no es tan significativo.

Otra cosa a recalcar de esta gráfica, es que sí existe un *overfitting* de las métricas en *Train* y *Validation*. Mientras que en *Test* que finalmente es la que se evalúa con respecto a la predicción, se observa que está fit.

[79]:
```
from sklearn.model_selection import learning_curve, validation_curve

# Define el modelo Random Forest
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Calcula las curvas de aprendizaje para el modelo
train_sizes, train_scores, test_scores = learning_curve(
    rf_classifier, X_train, y_train, cv=5, scoring='accuracy', train_sizes=np.
 ↪linspace(0.1, 1.0, 10))

# Calcula las medias y desviaciones estándar de las puntuaciones de␣
 ↪entrenamiento y prueba
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

# Calcula las curvas de validación para el conjunto de validación
param_range = [100, 200, 300, 400, 500]  # Ejemplo de valores para n_estimators␣
 ↪a validar
train_scores, val_scores = validation_curve(
    rf_classifier, X_train, y_train, cv=5, scoring='accuracy',␣
 ↪param_name='n_estimators', param_range=param_range)

train_mean_val = np.mean(train_scores, axis=1)
train_std_val = np.std(train_scores, axis=1)
val_mean = np.mean(val_scores, axis=1)
val_std = np.std(val_scores, axis=1)

# Plotea las curvas de aprendizaje para entrenamiento y validación
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(train_sizes, train_mean, color='blue', marker='o', markersize=5,␣
 ↪label='Entrenamiento')
plt.fill_between(train_sizes, train_mean + train_std, train_mean - train_std,␣
 ↪alpha=0.15, color='blue')
plt.plot(train_sizes, test_mean, color='green', linestyle='--', marker='s',␣
 ↪markersize=5, label='Prueba')
```

```
plt.fill_between(train_sizes, test_mean + test_std, test_mean - test_std,␣
 ↪alpha=0.15, color='green')
plt.xlabel('Tamaño del Conjunto de Entrenamiento')
plt.ylabel('Puntuación de Precisión')
plt.legend(loc='lower right')
plt.title('Curvas de Aprendizaje (Entrenamiento)')

plt.subplot(1, 2, 2)
plt.plot(param_range, train_mean_val, color='blue', marker='o', markersize=5,␣
 ↪label='Entrenamiento')
plt.fill_between(param_range, train_mean_val + train_std_val, train_mean_val -␣
 ↪train_std_val, alpha=0.15, color='blue')
plt.plot(param_range, val_mean, color='green', linestyle='--', marker='s',␣
 ↪markersize=5, label='Validación')
plt.fill_between(param_range, val_mean + val_std, val_mean - val_std, alpha=0.
 ↪15, color='green')
plt.xlabel('Número de Estimadores (n_estimators)')
plt.ylabel('Puntuación de Precisión')
plt.legend(loc='lower right')
plt.title('Curvas de Validación (n_estimators)')

# Línea de referencia para el punto óptimo (ejemplo)
plt.axvline(x=300, color='red', linestyle='--', linewidth=1.5)
plt.annotate('Óptimo', xy=(300, 0.9), color='red')

plt.tight_layout()
plt.show()
```



Habiendo visto las curvas de aprendizaje y de validación, podemos entender el grado de varianza. El *test* tiene una tendencia de crecimiento, y la separación con el entrenamiento no es significativa entre mayor sea el conjunto de entrenamiento. Comparando donde están los valores de *Train* sobre los de *Test*, podemos diferenciar que no hay un sesgo significativo, como ya habíamos mencionado en la gráfica de barras de las métricas.

Habiendo dicho esto, se sabe que si la brecha entre estas dos no es significativa, el grado de varianza es bajo y en este caso, *test*, no tiene un sobreajuste del modelo. Mientras que la curva de validación, nos indica que deberíamos reconsiderar el hiperparámeto a 300 estimadores.

Por último, habiendo visto estas gráficas, podemos notar que sí existe un sobreajuste sobre *Train* y *Validation*, aunque no tengan una separación significativa sobre el *Test*.

5 Uso de técnicas de regularización o ajuste de parámetros para mejorar el desempeño

5.1 Primera técnica: RidgeClassifier (también GridSearchCV como segunda técnica dentro de esta misma)

[80]:
```
# Métricas antes de encontrar el mejor alpha
print("Métricas antes de encontrar el mejor alpha:")
print("Accuracy (entrenamiento):", accuracy_train)
print("Accuracy (prueba):", accuracy_test)
print("Accuracy (validación):", accuracy_val)
print("Precision (entrenamiento):", precision_train)
print("Precision (prueba):", precision_test)
print("Precision (validación):", precision_val)
print("Recall (entrenamiento):", recall_train)
print("Recall (prueba):", recall_test)
print("Recall (validación):", recall_val)
print("F1 Score (entrenamiento):", f1_train)
print("F1 Score (prueba):", f1_test)
print("F1 Score (validación):", f1_val)

from sklearn.linear_model import RidgeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score

# Definir el modelo RidgeClassifier
ridge_classifier = RidgeClassifier()

# Definir el espacio de búsqueda de hiperparámetros
param_grid = {'alpha': [0.001, 0.01, 0.1, 1.0, 10.0]}  # Puedes ajustar los␣
 ↪valores según sea necesario

# Realizar la búsqueda de hiperparámetros con validación cruzada (CV)
```

```
grid_search = GridSearchCV(estimator=ridge_classifier, param_grid=param_grid,␣
 ↪cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Obtener el mejor valor de alpha encontrado
best_alpha = grid_search.best_params_['alpha']

# Crear un nuevo modelo RidgeClassifier con el mejor alpha
best_ridge_classifier = RidgeClassifier(alpha=best_alpha)
best_ridge_classifier.fit(X_train, y_train)

# Realizar predicciones en los tres conjuntos de datos con el mejor modelo
y_train_pred = best_ridge_classifier.predict(X_train)
y_test_pred = best_ridge_classifier.predict(X_test)
y_val_pred = best_ridge_classifier.predict(X_val)

# Calcular métricas de rendimiento para cada conjunto con el mejor modelo
accuracy_train = accuracy_score(y_train, y_train_pred)
accuracy_test = accuracy_score(y_test, y_test_pred)
accuracy_val = accuracy_score(y_val, y_val_pred)

precision_train = precision_score(y_train, y_train_pred)
precision_test = precision_score(y_test, y_test_pred)
precision_val = precision_score(y_val, y_val_pred)

recall_train = recall_score(y_train, y_train_pred)
recall_test = recall_score(y_test, y_test_pred)
recall_val = recall_score(y_val, y_val_pred)

f1_train = f1_score(y_train, y_train_pred)
f1_test = f1_score(y_test, y_test_pred)
f1_val = f1_score(y_val, y_val_pred)

# Métricas después de encontrar el mejor alpha
print("Métricas antes de encontrar el mejor alpha:")
print("Accuracy (entrenamiento):", accuracy_train)
print("Accuracy (prueba):", accuracy_test)
print("Accuracy (validación):", accuracy_val)
print("Precision (entrenamiento):", precision_train)
print("Precision (prueba):", precision_test)
print("Precision (validación):", precision_val)
print("Recall (entrenamiento):", recall_train)
print("Recall (prueba):", recall_test)
print("Recall (validación):", recall_val)
print("F1 Score (entrenamiento):", f1_train)
print("F1 Score (prueba):", f1_test)
print("F1 Score (validación):", f1_val)
```

```
# Imprimir el mejor valor de alpha
print("\nMejor valor de alpha encontrado:", best_alpha)
```

```
Métricas antes de encontrar el mejor alpha:
Accuracy (entrenamiento): 1.0
Accuracy (prueba): 0.9298245614035088
Accuracy (validación): 1.0
Precision (entrenamiento): 1.0
Precision (prueba): 0.90625
Precision (validación): 1.0
Recall (entrenamiento): 1.0
Recall (prueba): 0.9666666666666667
Recall (validación): 1.0
F1 Score (entrenamiento): 1.0
F1 Score (prueba): 0.9354838709677419
F1 Score (validación): 1.0
Métricas antes de encontrar el mejor alpha:
Accuracy (entrenamiento): 0.9626373626373627
Accuracy (prueba): 0.9298245614035088
Accuracy (validación): 1.0
Precision (entrenamiento): 0.9438943894389439
Precision (prueba): 0.90625
Precision (validación): 1.0
Recall (entrenamiento): 1.0
Recall (prueba): 0.9666666666666667
Recall (validación): 1.0
F1 Score (entrenamiento): 0.9711375212224108
F1 Score (prueba): 0.9354838709677419
F1 Score (validación): 1.0

Mejor valor de alpha encontrado: 0.1
```

El código presentado demuestra un enfoque sistemático para optimizar los hiperparámetros de un modelo de clasificación, específicamente utilizando el algoritmo de RidgeClassifier.

El primer paso en este proceso es evaluar el rendimiento inicial del modelo en una serie de métricas de evaluación antes de realizar cualquier ajuste de hiperparámetros. Estas métricas incluyen la precisión (accuracy), precisión (precision), recall y F1-score en tres conjuntos de datos cruciales: entrenamiento, prueba y validación. Esta evaluación inicial proporciona una línea de base para comprender el rendimiento inicial del modelo sin ninguna optimización.

A continuación, se define un espacio de búsqueda de hiperparámetros mediante el uso de un diccionario llamado param_grid. El hiperparámetro clave en este caso es alpha, que controla la fuerza de regularización en el modelo Ridge. La regularización es una técnica importante para prevenir el sobreajuste y garantizar que el modelo generalice bien a datos no vistos. La búsqueda de alpha permitirá encontrar el valor óptimo que equilibre la complejidad del modelo y su capacidad para ajustarse a los datos.

El proceso de búsqueda de hiperparámetros se realiza utilizando la técnica de validación cruzada (CV). GridSearchCV, una herramienta poderosa en el aprendizaje automático, automatiza esta búsqueda exhaustiva. El modelo RidgeClassifier se ajusta repetidamente con diferentes valores de alpha, y se utiliza la métrica de precisión (scoring='accuracy') para evaluar cómo cada configuración de hiperparámetros se desempeña en CV. Esto permite identificar el mejor valor de alpha que maximiza la precisión del modelo en datos no vistos.

Una vez que se ha encontrado el mejor valor de alpha, se procede a crear un nuevo modelo RidgeClassifier utilizando este valor óptimo y se entrena en el conjunto de entrenamiento. Esto representa el modelo final ajustado que ha sido optimizado para el conjunto de datos en cuestión.

El siguiente paso es evaluar el rendimiento de este modelo optimizado en los tres conjuntos de datos: entrenamiento, prueba y validación. Las mismas métricas de evaluación que se calcularon inicialmente se calculan nuevamente para determinar cómo ha mejorado el rendimiento del modelo después de la optimización de hiperparámetros. Este proceso proporciona una visión clara de cómo la búsqueda de hiperparámetros ha impactado el desempeño del modelo.

Observamos disminuye el overfitting en el entrenamiento, sin embargo, se conserva en la validación.

6 Segunda Técnica: Selección de Características

[81]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Cargar el conjunto de datos de cáncer de mama de Wisconsin
data = load_breast_cancer()
X = data.data  # Variables predictoras
y = data.target  # Variable objetivo (0: benigno, 1: maligno)

# Dividir el conjunto de datos en entrenamiento, prueba y validación
# Primero, dividimos en entrenamiento y prueba (80% entrenamiento, 20% prueba)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Luego, dividimos el conjunto de "temp" en prueba y validación (50% prueba,␣
 ↪50% validación)
X_test, X_val, y_test, y_val = train_test_split(X_temp, y_temp, test_size=0.5,␣
 ↪random_state=42)

rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier.fit(X_train, y_train)

print("Métricas antes de la selección de características:")
print("Accuracy (entrenamiento):", accuracy_train)
print("Accuracy (prueba):", accuracy_test)
print("Accuracy (validación):", accuracy_val)
```

```
from sklearn.feature_selection import SelectKBest, f_classif

# Aplicar selección de características
selector = SelectKBest(score_func=f_classif, k='all')  # Puedes ajustar k según
 ↪sea necesario
X_train_selected = selector.fit_transform(X_train, y_train)
X_test_selected = selector.transform(X_test)
X_val_selected = selector.transform(X_val)

# Entrenar el modelo en las características seleccionadas (puedes usar Random
 ↪Forest u otro modelo)
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier.fit(X_train_selected, y_train)

# Realizar predicciones en los tres conjuntos de datos
y_train_pred = rf_classifier.predict(X_train_selected)
y_test_pred = rf_classifier.predict(X_test_selected)
y_val_pred = rf_classifier.predict(X_val_selected)

# Calcular métricas de rendimiento para cada conjunto
accuracy_train = accuracy_score(y_train, y_train_pred)
accuracy_test = accuracy_score(y_test, y_test_pred)
accuracy_val = accuracy_score(y_val, y_val_pred)

precision_train = precision_score(y_train, y_train_pred)
precision_test = precision_score(y_test, y_test_pred)
precision_val = precision_score(y_val, y_val_pred)

recall_train = recall_score(y_train, y_train_pred)
recall_test = recall_score(y_test, y_test_pred)
recall_val = recall_score(y_val, y_val_pred)

f1_train = f1_score(y_train, y_train_pred)
f1_test = f1_score(y_test, y_test_pred)
f1_val = f1_score(y_val, y_val_pred)

# Comparar métricas antes y después de la selección de características

# Repite para precision, recall y F1 Score

# Imprimir separador
print("\n---------------------------------\n")

print("Métricas después de la selección de características:")
print("Accuracy (entrenamiento):", accuracy_train)
print("Accuracy (prueba):", accuracy_test)
print("Accuracy (validación):", accuracy_val)
```

```
# Repite para precision, recall y F1 Score
```

```
Métricas antes de la selección de características:
Accuracy (entrenamiento): 0.9626373626373627
Accuracy (prueba): 0.9298245614035088
Accuracy (validación): 1.0

--------------------------------

Métricas después de la selección de características:
Accuracy (entrenamiento): 1.0
Accuracy (prueba): 0.9298245614035088
Accuracy (validación): 1.0
```

El proceso central del código es la selección de características, donde se aplica la técnica SelectKBest con la función de puntuación f_classif. Esta técnica se basa en pruebas estadísticas de análisis de varianza (ANOVA) para seleccionar las características más relevantes para el problema de clasificación. El número de características seleccionadas se controla mediante el parámetro k.

Después de seleccionar las características más importantes, se entrena un nuevo modelo RandomForestClassifier en el conjunto de entrenamiento utilizando estas características seleccionadas. Este paso es crucial para evaluar si la selección de características mejora o afecta el rendimiento del modelo.

En este caso, observamos que mejoró el entrenamiento tanto, que ahora existe un *overfitting*.

7 Tercera Técnica: Variación del hiperparámetro

En este caso, retomaremos la gráfica de validación que nos indicaba que es óptima la *nestimators* de 300 para nuestro bosque. Utilizaremos esa y veremos los resultados del Cross-Validation, comparándolo con el inicial de 100 árboles.

[82]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

# Cargar el conjunto de datos de cáncer de mama de Wisconsin
data = load_breast_cancer()
X = data.data  # Variables predictoras
y = data.target  # Variable objetivo (0: benigno, 1: maligno)

# Crear dos modelos RandomForestClassifier con diferentes profundidades de árbol
rf_classifier_depth_100 = RandomForestClassifier(n_estimators=100,␣
 ↪max_depth=100, random_state=42)
rf_classifier_depth_300 = RandomForestClassifier(n_estimators=300,␣
 ↪max_depth=100, random_state=42)

# Realizar validación cruzada para el modelo con profundidad de 100
```

```
num_folds = 5  # Número de divisiones para cross-validation
scoring = 'accuracy'  # Métrica de evaluación, puedes cambiarla según tus␣
 ↪necesidades

# Realizar cross-validation y obtener la puntuación de cada pliegue para el␣
 ↪modelo con profundidad de 100
scores_depth_100 = cross_val_score(rf_classifier_depth_100, X, y, cv=num_folds,␣
 ↪scoring=scoring)

# Realizar cross-validation para el modelo con profundidad de 300
# Obtener la puntuación de cada pliegue para el modelo con profundidad de 300
scores_depth_300 = cross_val_score(rf_classifier_depth_300, X, y, cv=num_folds,␣
 ↪scoring=scoring)

# Imprimir las puntuaciones promedio para ambos modelos
print("Puntuación promedio (Profundidad 100):", scores_depth_100.mean())
print("Puntuación promedio (Profundidad 300):", scores_depth_300.mean())
```

```
Puntuación promedio (Profundidad 100): 0.9560937742586555
Puntuación promedio (Profundidad 300): 0.9613569321533924
```

Gracias a la curva de validación, observamos que el resultado del Cross-Validation sí mejora con una *nestimators* de 300, y es el valor más óptimo.

8 Conclusiones

Desde las gráficas de barras de las métricas pudimos observar como el *Test* nunca tuvo un sobreajuste, sin embargo, *Train* y *Validation* sí, lo cual pudimso confirmar con las curvas de aprendizaje. Esto se puede deber a que la base de datos seleccionada nunca se le hizo limpieza debido a que ya es precargada de *sklearn*. Sin embargo, también nos ayudó a obsevar que el modelo sí generaliza y su amplia cantidad de variables nos ayudaron a correr un buen algoritmo.

RandomForest es un algoritmo de Machine Learning el cual arroja métricas con valores bastante altos.

Las técnicas algunas veces ayudaron a mejorar el *overfitting*, sin embargo, eliminarlo en este caso, con este modelo, sería muy difícil y tal vez se deba encontrar algún otro modelo o juntarlo con otro para mejorar esto.