Practica_calificada2_Describa

May 18, 2025

Estudiante :

- Escriba Flores, Daniel Agustin

1 Primer Caso de Estudio: Predicción Temprana de Diabetes mediante Regresión Logística

---

1.1 Contexto

Se desea construir un modelo predictivo que permita estimar el riesgo de padecer diabetes en función de características relacionadas con la salud de una persona (por ejemplo: presión alta, colesterol alto, actividad física, entre otras). Para ello, se ha utilizado el conjunto de datos CDC Diabetes Health Indicators, disponible en el UCI Machine Learning Repository. Este dataset contiene información de más de 250.000 registros, con variables numéricas y categóricas relacionadas con hábitos y condiciones de salud.

La variable objetivo del modelo es binaria: - `0` = No presenta diabetes - `1` = Presenta diabetes

Debido a que el número de personas diagnosticadas con diabetes es considerablemente menor al de personas sin diagnóstico, este caso representa un escenario con clases desbalanceadas, frecuente en contextos de medicina preventiva.

1.2 Preguntas de Análisis e Interpretación

1. **¿El dataset presenta un problema de desbalance de clases? Justifique su respuesta con base en los porcentajes observados.**

Sí, el dataset presenta un claro desbalance de clases.La distribución muestra que el 86.07% de los registros corresponden a personas sin diabetes (Clase 0) , mientras que solo el 13.93% son casos positivos (Clase 1). Se hace mas visible con el grafico de Distribucion encontrado

> Esta gran diferencia indica que el modelo puede tener dificultades para aprender patrones asociados a la clase minoritaria, afectando su capacidad predictiva en esa categoría.

2. **Explique cómo el desbalance de clases afectó al modelo de regresión logística sin SMOTE, especialmente en su capacidad para detectar personas con diabetes.**

Sin SMOTE, el modelo priorizo la clase mayoritaria (0), que son precisamente las personas que no tienen diabetes. Esto se refleja claramente en el bajo recall de la Clase 1 (0.16) , lo que significa que solo identificó correctamente al 16% de los casos reales de diabetes .Aunque el accuracy fue alto (0.86) , este valor es engañoso, ya que solo se enfoca en la clase mayoriyaria. Por eso mismo, el F1-score para la Clase 1 fue muy bajo (0.24) , reflejando una mala combinación entre precisión y recall. > En resumen, el desbalance provocó que el modelo fallara en detectar la mayoría de los casos de diabetes.

3. **Luego de aplicar SMOTE, ¿qué cambios se observan en las métricas para la clase 1 (diabetes)? Comente los beneficios y las posibles consecuencias de este cambio.**

Al aplicar SMOTE, se observa una mejora significativa en el recall para la Clase 1 (de 0.16 a 0.76) , lo que implica que ahora se detecta casi el 76% de los casos reales de diabetes , reduciendo notablemente los falsos negativos .

Sin embargo, esta mejora viene con un costo: la precisión disminuye considerablemente (de 0.53 a 0.31) , lo que genera más falsos positivos (personas sin diabetes clasificadas erróneamente como diabéticas).

Sobre el F1-score este Aumentó ligeramente, pasando de 0.24 a 0.44 , aunque sigue siendo bajo debido al desequilibrio entre precisión y recall.

> Esto podría traducirse en diagnósticos preliminares incorrectos que generen falsas alarmas, lo que podría requerir revisiones adicionales o exámenes complementarios.

4. **¿Cuál de los dos modelos considera más apropiado para un contexto de salud pública, en el que es fundamental identificar la mayor cantidad posible de personas con diabetes, aunque se cometan algunos falsos positivos? Justifique su respuesta con base en las métricas.**

Aunque ninguno de los dos modelos alcanza un desempeño óptimo para un uso clínico directo, el modelo con SMOTE resulta más adecuado en un contexto de salud pública donde la detección temprana de diabetes es prioritaria.

Este modelo logra un recall del 0.76 para la clase 1 (diabetes) , lo cual implica que ahora se identifica correctamente al 76% de los casos reales , un gran salto desde el 16% del modelo sin balanceo. Este aumento en el recall es clave para minimizar los falsos negativos , es decir, dejar pasar por alto a personas que sí tienen la enfermedad.

> Aunque como se observa esto trae como consecuencia un aumento en los falsos positivos , estos pueden gestionarse mediante exámenes adicionales o revisiones médicas, lo cual es preferible a no detectar casos reales. En este tipo de contextos preventivos, detectar más casos potenciales, incluso con algunas alertas falsas, es generalmente más deseable que no detectar los verdaderos casos .

5. **Explique por qué la métrica de accuracy puede ser engañosa en problemas con clases desbalanceadas. ¿Qué métricas deben priorizarse en este tipo de problemas y por qué?**

Cuando hay desbalanceo en clases, La accuracy puede ser engañosa porque mide la proporción total de predicciones correctas sin distinguir entre tipos de error. Un modelo puede alcanzar un alto accuracy simplemente prediciendo siempre la clase mayoritaria, ignorando completamente la

clase minoritaria. Por ejemplo, en nuestro caso, el modelo sin SMOTE tuvo un accuracy de 0.86 , pero solo identificó al 16% de los casos reales de diabetes.

En escenarios desbalanceados, es preferible usar métricas que evalúen el desempeño en cada clase:

- Recall : Mide cuántos verdaderos positivos fueron identificados. Es clave cuando queremos minimizar los falsos negativos .
- Precision : Evalúa cuántas de las predicciones positivas son realmente correctas. Útil para controlar los falsos positivos .
- F1-score : Combina precisión y recall, ofreciendo un equilibrio útil en escenarios desbalanceados.
- AUC-ROC : Evalúa el desempeño global del modelo en distintos umbrales, permitiendo comparar modelos incluso en presencia de desbalance.

Estas métricas ofrecen una visión más realista del comportamiento del modelo, especialmente en la clase minoritaria, lo que es crucial en contextos médicos.

Codigos del Primer Caso

[ ]:
```
# Instalar las librerías necesarias

from ucimlrepo import fetch_ucirepo
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪ConfusionMatrixDisplay
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE
import seaborn as sns
import matplotlib.pyplot as plt

# ------------------------------
# 1. CARGA DE DATOS DESDE UCI
# ------------------------------

# Cargar dataset
cdc_diabetes = fetch_ucirepo(id=891)

# Extraer variables predictoras (X) y objetivo (y)
X = cdc_diabetes.data.features
y = cdc_diabetes.data.targets

# Fusionar en un solo DataFrame para exploración
df = pd.concat([X, y], axis=1)

# Mostrar primeras filas
```

```
print("Primeras filas del dataset:")
display(df.head())

# Renombrar variable objetivo si es necesario
target_col = y.columns[0]

# Visualizar distribución de clases
plt.figure(figsize=(6,4))
sns.countplot(x=target_col, data=df)
plt.title('Distribución de clases (Diabetes)')
plt.xlabel('Clase (0 = No Diabetes, 1 = Diabetes)')
plt.ylabel('Frecuencia')
plt.show()

# Porcentajes
class_percent = df[target_col].value_counts(normalize=True) * 100
print("\nDistribución porcentual de clases:")
print(class_percent)

if class_percent.min() < 40:
    print(" Dataset desbalanceado: se recomienda aplicar SMOTE.")
else:
    print(" Las clases están balanceadas.")
```

[ ]:
```
# ----------------------------
# 2. PREPROCESAMIENTO
# ----------------------------

# Escalamiento
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# División entrenamiento/prueba con estratificación
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.3, stratify=y, random_state=42
)

print(f"Tamaño entrenamiento: {X_train.shape}")
print(f"Tamaño prueba: {X_test.shape}")
```

[ ]:
```
# ----------------------------
# 3. REGRESIÓN LOGÍSTICA SIN BALANCEO
# ----------------------------

# Entrenar modelo
log_model = LogisticRegression(max_iter=1000)
log_model.fit(X_train, y_train)
```

```
# Predicción
y_pred = log_model.predict(X_test)

# Evaluación
print("Evaluación SIN SMOTE:")
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

# Visualizar matriz
ConfusionMatrixDisplay.from_estimator(log_model, X_test, y_test, cmap="Blues")
plt.title("Matriz de Confusión - Sin SMOTE")
plt.show()
```

[ ]:
```
# ----------------------------
# 4. APLICACIÓN DE SMOTE
# ----------------------------

# Aplicar SMOTE
sm = SMOTE(random_state=42)
X_res, y_res = sm.fit_resample(X_train, y_train)

# Entrenar nuevo modelo con datos balanceados
log_model_sm = LogisticRegression(max_iter=1000)
log_model_sm.fit(X_res, y_res)

# Predicción con modelo balanceado
y_pred_sm = log_model_sm.predict(X_test)

# Evaluación
print("Evaluación CON SMOTE:")
print(confusion_matrix(y_test, y_pred_sm))
print(classification_report(y_test, y_pred_sm))

# Visualizar matriz
ConfusionMatrixDisplay.from_estimator(log_model_sm, X_test, y_test,␣
 ↪cmap="Purples")
plt.title("Matriz de Confusión - Con SMOTE")
plt.show()
```

---

2 Segundo Caso de Estudio: Árboles de Decisión y Validación (10 puntos)

2.1 contexto

El dataset CDC Diabetes Health Indicators contiene información de encuestas realizadas por los Centros para el Control y la Prevención de Enfermedades (CDC) en Estados Unidos. Este conjunto de datos tiene como objetivo predecir si una persona tiene o no diabetes basándose en una serie de indicadores de salud, como IMC, edad, nivel de actividad física, presión arterial, entre otros.

Se te entrega un fragmento de código donde se entrena un modelo de árbol de decisión para clasificar si una persona padece diabetes o no. Tu tarea es analizar, explicar y comparar cómo cambia su desempeño cuando se le aplica un proceso de búsqueda de hiperparámetros con `GridSearchCV`.

La actividad está dividida en dos partes:

1. Código base del modelo de árbol de decisión sin ajuste de parámetros
2. Código con ajuste automático de hiperparámetros utilizando GridSearchCV

---

3 Instrucciones para desarrollar el caso

1. Analiza ambos bloques de código que se te entregan (modelo sin ajustes y modelo con ajuste de hiperparámetros).
2. Explica con tus propias palabras qué hace cada parte del código. Puedes escribir tus respuestas como un documento en Word, PDF o directamente en el cuaderno de Colab.
3. Agrega comentarios al código (en las celdas de código o al lado del texto) como si tú fueras el profesor que está explicando a otros.
4. Compara los resultados obtenidos antes y después del ajuste:

- ¿Qué métricas cambian?
  - ¿Qué diferencias notas en la estabilidad del modelo?

  `Respuesta en la parte final`

5. Justifica por qué es importante aplicar una búsqueda de hiperparámetros con GridSearchCV en modelos como el árbol de decisión. Si no entiendes alguna parte, puedes usar herramientas como ChatGPT, Bing o Copilot para ayudarte a comprenderla.

   `Respuesta en la parte final`

   Recuerda: - No se trata de copiar y pegar lo que diga la IA. - Tu tarea es entender el código y redactar tu propia explicación.

3.0.1 Interpretacion del bloque

El código entrena un modelo de árbol de decisión con profundidad máxima limitada en 4 para prevenir el sobreajuste. entramos el modelo con los datos, para luego hacer una visualizacion de la estructura completa con `plot_tree`.

Sobre el grafico, se puede ver cómo toma decisiones basado en las características del dataset y com solo llego a la profundidad maxima de 4 y como clasifica si una persona tiene dibetes o no.

[ ]:
```
# Importar librerías adicionales
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix, classification_report,␣
 ↪ConfusionMatrixDisplay

# ----------------------------------------
# 5. ENTRENAMIENTO DEL MODELO DE ÁRBOL
# ----------------------------------------

# Inicializar el modelo (se puede ajustar max_depth para evitar sobreajuste)

# Se fija la semilla (42) para reproducibilidad
# Limitamos la profundida maxima a 4 para evitar sobreajuste
tree_model = DecisionTreeClassifier(random_state=42, max_depth=4)
tree_model.fit(X_train, y_train) #Entrenamos el modelo

# Visualización del árbol
plt.figure(figsize=(20,10)) # Tamano del grafico
plot_tree(tree_model, filled=True, feature_names=X.columns, class_names=["No␣
 ↪Diabetes", "Diabetes"])
plt.title("Árbol de Decisión (profundidad=4)")
plt.show()
```

3.0.2 Interpretacion del bloque

Este bloque evalúa el desempeño del árbol de decisión en el conjunto de test mediante la matriz de confusión y el reporte de clasificación , que incluye métricas como `precisión`, `recall` y `f1-score` por clase.

Se muestra gráficamente cómo se distribuyen los verdaderos y falsos positivos/negativos, lo cual permite analizar el equilibrio entre aciertos y errores del modelo.

[ ]:
```
# ----------------------------------------
# 6. EVALUACIÓN DEL MODELO EN TEST
# ----------------------------------------

# Predicciones del modelo
y_pred_tree = tree_model.predict(X_test) # en el conjunto de prueba

# Matriz de confusión
print("Matriz de Confusión:")
cm = confusion_matrix(y_test, y_pred_tree)
print(cm)  # Compara las etiquetas reales vs la predichas
```

```
# Visualización de la matriz de confusion
ConfusionMatrixDisplay(cm, display_labels=["No Diabetes", "Diabetes"]).
 ↪plot(cmap="Greens")
plt.title("Matriz de Confusión - Árbol de Decisión")
plt.show()

# Reporte detallado
print("Reporte de Clasificación:")
print(classification_report(y_test, y_pred_tree))
```

3.0.3 Interpretacion del bloque

Este bloque realiza una validación cruzada de 4 pliegues para medir la estabilidad y generalización del modelo usando el `F1-score macro` , lo cual da relevancia equitativa a ambas clases. Los resultados muestran el desempeño promedio y su variabilidad entre pliegues, lo que permite identificar si el modelo se comporta consistentemente o necesita ajustes. Una baja desviación estándar indica confianza en el modelo; de lo contrario, se sugiere explorar optimización de parámetros o selección de características.

[ ]:
```
# ----------------------------------------
# 7. VALIDACIÓN CRUZADA
# ----------------------------------------

# Aplicar validación cruzada de 4 pliegues
scores = cross_val_score(tree_model, X_scaled, y, cv=4, scoring='f1_macro')  #␣
 ↪F1-score macro para evaluar el balance entre ambas clases

#  Impresion de los Resultados
print("Resultados de Validación Cruzada (F1-score macro):")
print(f"Scores individuales por pliegue: {scores}")
print(f"Promedio del F1-score: {np.mean(scores):.4f}")
print(f"Desviación estándar: {np.std(scores):.4f}")

# Interpretación sugerida
if np.std(scores) < 0.02: # Umbral de estabilidad
    print("El modelo muestra buena estabilidad entre los pliegues.")
else:
    print("El modelo podría ser inestable: considera ajustar hiperparámetros o␣
 ↪validar la selección de variables.")
```

3.0.4 Interpretacion del bloque

Explorando vemos que el código utiliza `GridSearchCV` para buscar automáticamente la mejor combinación de hiperparámetros del árbol de decisión, evaluando distintas profundidades, mínimos de

división y manejo de desbalance. La métrica usada es el `F1-score macro` y se aplica una validación cruzada de 5 pliegues para asegurar robustez. El resultado es un modelo optimizado que mejora su capacidad de generalización frente al modelo inicial sin ajuste, por tal motivo lo llamamos `best_tree_model`

[ ]:
```
from sklearn.model_selection import GridSearchCV

# Definir el modelo base sin ningun ajuste
base_tree = DecisionTreeClassifier(random_state=42)

# Definir el espacio de búsqueda de hiperparámetros
param_grid = {
    'max_depth': [3, 5, 7, 9, None], # Profundidad máxima del árbol
    'min_samples_split': [2, 5, 10], # Mínimo de muestras para dividir un nodo
    'class_weight': [None, 'balanced'] # Manejo de desbalanceo entre clases
}

# Buscar la mejor combinación de hiperparámetros
grid_search = GridSearchCV(
    estimator=base_tree,   # colocamos el modelo base
    param_grid=param_grid, # Los hiperparametos
    scoring='f1_macro',    # Métrica de evaluación
    cv=5,                  # Número de pliegues en validación cruzada
    n_jobs=-1              # Usar todos los núcleos disponibles
)

# Entrenar la búsqueda
grid_search.fit(X_train, y_train)

# Mejor combinación encontrada
print("Mejores hiperparámetros encontrados:")
print(grid_search.best_params_)

# Extraer el mejor modelo encontrado
best_tree_model = grid_search.best_estimator_
```

3.0.5 Interpretacion del bloque

Este bloque vuelve a aplicar la validación cruzada , pero ahora con el modelo optimizado tras el ajuste de hiperparámetros. Permite comparar si la búsqueda automática mejoró realmente el desempeño del modelo, evaluando el `F1-score macro promedio` y su estabilidad entre pliegues. De esta forma, se verifica si el nuevo modelo es más robusto y equilibrado que el original, especialmente para predecir ambas clases en un contexto médico.

[ ]:

```
# Validación cruzada con el mejor árbol
best_scores = cross_val_score(best_tree_model, X_scaled, y, cv=5,␣
 ↪scoring='f1_macro')

print("\nValidación cruzada con modelo ajustado:")
print(f"F1 macro (por pliegue): {best_scores}")
print(f"F1 promedio: {np.mean(best_scores):.4f}")
print(f"Desviación estándar: {np.std(best_scores):.4f}")

if np.std(best_scores) < 0.02:
    print("El modelo ajustado muestra buena estabilidad.")
else:
    print("El modelo sigue siendo inestable: considerar más técnicas como␣
 ↪selección de variables o ensambles.")
```

3.1 4.Compara los resultados obtenidos antes y después del ajuste:

- ¿Qué métricas cambian?
- ¿Qué diferencias notas en la estabilidad del modelo?

  Con el ajuste se puede observar que la métrica que muestra una mejora más clara es el F1-score macro , que pasó de un promedio de 0.5258 a 0.6141. Esto indica un mejor equilibrio entre precisión y recall, especialmente para la clase minoritaria (diabetes), gracias al uso de `'class_weight': 'balanced'`, que ayuda a dar mayor relevancia a esta clase durante el entrenamiento.

  Sobre la estabilidad, tambien hay diferencias, pues a diferencia del primer modelo el ajustado presenta buena estabilidad, ya que la desviación estándar bajó considerablemente de 0.0636 a 0.0076 , lo que refleja que el modelo ahora tiene un comportamiento mucho más consistente entre los distintos pliegues de validación. Este valor está por debajo de 0.02 , un umbral comúnmente usado como referencia para considerar que un modelo tiene buena estabilidad.

3.2 5. Justifica por qué es importante aplicar una búsqueda de hiperparámetros con GridSearchCV en modelos como el árbol de decisión.

Si no entiendes alguna parte, puedes usar herramientas como ChatGPT, Bing o Copilot para ayudarte a comprenderla.

La búsqueda de hiperparámetros mediante GridSearchCV es especialmente valiosa en modelos como el árbol de decisión, ya que permite encontrar automáticamente la combinación óptima de parámetros, como la profundidad máxima del árbol, el número mínimo de muestras para dividir un nodo o cómo se maneja el desbalance de clases puede mejorar significativamente el rendimiento final.

En nuestro cas, el proceso identificó como mejores parámetros los siguientes: - uso de balanceo de clases (class_weight igual a ‘balanced’), - profundidad máxima del árbol igual a 7 (max_depth=7) - un mínimo de 2 muestras para dividir un nodo (min_samples_split=2).

Estos valores permitieron que el modelo lograra un mejor equilibrio entre precisión y recall.

---

4 Conclusion Extra

Ajustar el árbol de decisión permitió mejorar su capacidad para detectar casos de diabetes sin descuidar la estabilidad del modelo. La búsqueda automatizada de hiperparámetros ayudó a encontrar una solución más equilibrada y generalizable. Validar con diferentes pliegues mostró que el modelo ahora responde de manera más consistente ante nuevos datos. En total, se logró un enfoque más sólido para un problema donde no se puede fallar en identificar riesgos reales. La interpretación activa de los resultados, apoyada por herramientas de IA, también resulta (de manera personal) ser valiosa para comprender el comportamiento del modelo y afianzar el aprendizaje práctico.

[9]:
```
import nbformat

# Cargar el notebook
notebook_path = 'Practica_calificada2_Describa.ipynb'
nb = nbformat.read(notebook_path, as_version=4)

# Actualizar metadatos
if 'latex_metadata' not in nb.metadata:
    nb.metadata['latex_metadata'] = {}

nb.metadata['latex_metadata']['title'] = 'Práctica Calificada 2'
nb.metadata['latex_metadata']['author'] = 'Tu Nombre'
nb.metadata['latex_metadata']['date'] = '\\today'  # Fecha actual en LaTeX

# Guardar el notebook
nbformat.write(nb, notebook_path)
print(f"Metadatos actualizados en {notebook_path}")
```

```
Metadatos actualizados en Practica_calificada2_Describa.ipynb
```