Predicción de Diabetes Usando Regresión Logística

Grant Nathaniel Keegan | A01700753

ABSTRACT

En este proyecto, voy a aplicar un algoritmo de regresión logística para predecir si un paciente tiene diabetes o no. Utilizando un conjunto de datos del repositorio de UC Irvine. El proceso incluye varias etapas, empezando por el procesamiento de datos usando ETL. Después, se desarrollaron tres algoritmos de aprendizaje de máquina. Uno base sin frameworks, uno mejorado que despliega mejores resultados basado en el cambio de hiperparámetros. Y el tercero utilizando frameworks como sklearn y XGBoost. En los algoritmos se implementaron técnicas como regularización para regresión logística. Los resultados fueron evaluados con métricas de exactitud, graficas, y juzgando el desempeño basado en indicadores como el grado de varianza, sesgo, accuracy, y juzgando si tiene overfitting o underfitting. El objetivo de este proyecto es desarrollar y promover el uso de aprendizaje de maquina para el campo de la medicina.

ÍNDICE

I. Introducción…….…...1

II. Análisis y procesamiento de datos……..…1

III. Desarrollo del modelo base (Regresión Logística)…2

IV. Resultados del modelo base……..3

V. Desarrollo del modelo mejorado……6

VI. Resultados del modelo mejorado……..6

VII. Desarrollo del modelo con frameworks…….….7

VIII. Resultados del modelo con frameworks……7

XI. Correcciones……8

X. Conclusiones……..9

Notas y Referencias……9

I. INTRODUCCIÓN

La diabetes es un problema global que ha crecido con el tiempo.[1] De acuerdo con la Organización Mundial de la Salud, los casos de diabetes han aumentado de 200 millones en 1990 a 830 millones en 2022. Los países más afectados son los de escasos recursos, y la prevención no es entendida del todo por gente que la padece.

Por esto es importante entender los síntomas de la diabetes, enfocado en prevención temprana, así como identificar quienes tienen más riesgo de sufrir de diabetes. El objetivo de este proyecto es generar un modelo de predicción efectivo para no solo prevenir diabetes, si no diagnosticar dentro de un rango de precisión (>90%) si una persona padece de ella.

Aquí voy a documentar cómo podemos predecir diabetes utilizando aprendizaje de máquina basado en procesamiento de datos. Analizando dos modelos efectivos utilizando un algoritmo de **regresíon logística**, basada en un cuestionario a personas que están en riesgo, o ya padecen de diabetes.

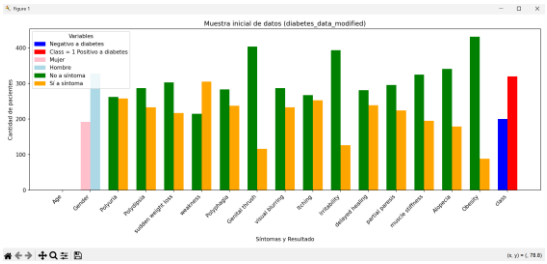
II. ANÁLISIS Y PROCESAMIENTO DE DATOS

Los datos que utilicé para este proyecto, fueron recolectados del artículo *Likelihood Prediction of Diabetes at Early Stage Using Data Mining Techniques.*[2] A través del recurso *UC Irvine Machine Learning Repository.* Los datos incluyen una muestra de 520 pacientes que tomaron una encuesta basada en síntomas. Los datos que incluyen son edad, sexo, una recolección de 14 síntomas, tales como alopecia, visión nublada, obesidad e irritabilidad. Finalmente, ‘class’, que indica si el paciente resultó positivo o negativo para la diabetes.

Para este proyecto, me voy a enfocar solamente en los datos binarios de los 14 síntomas, el género y la edad, a los que los pacientes respondieron "yes" o "no". Y "class" a la cual el resultado dio positive o negative. **Esto nos da un total de 2 clases.** En otras palabras, una clase binaria. Sera importante notar esto para el diseño de nuestro código.

Para poder trabajar con los datos, es importante modificarlos usando procesos de ETL que sean fáciles de trabajar en los algoritmos implementados. Es importante saber si la tabla está completa, así que se implementa el comando `print(df.isnull().any())` para saber si hay espacios en blanco. Como no hay pasamos al siguiente paso, que fue convertir los valores de "yes" o "no" a 0 y 1 para cada uno de los síntomas. También "positive" o "negative" en class para el resultado de si una persona padece diabetes o no. Esto lo logré usando el comando df.replace().

Los resultados los podemos ver en la siguiente tabla de figura 1, donde muestra el género, cada síntoma de quienes tuvieron 1 o 0 (naranja o verde), y al final, cuantas personas padecen de diabetes después de la prueba.



Aquí podemos ver cómo los síntomas más raros fueron los de "Genital thrush", "Irritability" y "Obesity". Mientras que los más comunes fueron "weakness", "polyuria" e "itching". Visualizar los datos de nuestra tabla antes de aplicar los algoritmos es importante, ya que nos da una representación gráfica de qué síntomas de diabetes son más comunes.

Es importante mencionar que en esta base de datos **no hay outliers** importantes que puedan afectar los resultados. Por lo que no aplique más procesamiento para eliminar o omitir datos o pacientes de esta base.

Uno de los pasos más importantes en nuestro proceso, es procesar los datos de la tabla transformada para que nuestro algoritmo de Machine Learning pueda trabajar adecuadamente con ellos.[5] Para esto, fue necesario

1) **Normalizar las variables de X** para que estén en la misma escala. Sin el uso de scikit-learn, con funciones como min-max scaling, tuve que usar la funcion `X = (X - X.mean()) / X.std()`. Esto divide entre la desviación estándar y mejora los datos para cuando trabaje con gradiente descendiente.
2) **Convertir los datos y clases a arreglos de NumPy** al usar la función `np.array()`, puedo utilizar operaciones vectorizadas con funciones matemáticas y utilizar NumPy de una forma más efectiva.
3) **Agregar Bias**. Con el bias, agrego un vector de 1s a los datos de X, concatenando el 1 con `np.c_` a cada columna de datos. Esto hace que el modelo ajuste el intercepto de la recta.

Técnicas de preprocesamiento implementadas:

1. Imputación básica: En mis datos, tuve que cambiar los valores de "Yes" y "No" a binario para que mis algoritmos puedan utilizarlos.

2. Normalización de Datos (X): Antes de que puedan ser utilizados los datos, tuve que normalizarlos y convertirlos a arrays con np.array().

Además de eso, no apliqué más técnicas de procesamiento de datos, ya que la tabla original está completa y con interpretación efectiva para trabajar con mis modelos de regresión logística.

III. DESARROLLO DEL MODELO BASE (REGRESIÓN LOGÍSTICA)

Archivo: logreg_base.py

Mi objetivo para este proyecto es utilizar un modelo de regresión logística, basada en los datos de la tabla. La edad del paciente en formato numérico, si es hombre o mujer (transformado a 1 o 0, respectivamente), y los 14 síntomas para predecir con alta exactitud si un paciente dará 0 o 1 en los resultados de su examen para detectar diabetes. Para este proyecto, me apoyé del recurso *How To Implement Logistic Regression From Scratch in Python*, de Jason Brownlee en Machine Learning Mastery.[3] Y de las notas de clase del módulo 2 y 3 del curso inteligencia artificial para la ciencia de datos I[4]. Además de recursos de GeeksforGeeks y YouTube. (Referencias Adicionales).

III (a): Dividir datos entre entrenamiento (Train) y prueba (test).

El primer paso para poder entrenar mi modelo, es dividir los datos entre datos de entrenamiento, y datos de prueba. Esto es algo que haré tanto para el algoritmo escrito a

mano, como para los algoritmos con framework. Para mantener el modelo simple. Para esto, utilice la función np.split.[6] Esta división es importante para el entrenamiento de modelos, ya que permite entrenar el modelo (train), ajustar hiperparámetros como pesos, y evaluar el desempeño en datos nuevos (test).

Para la implementación del modelo en todos los códigos, siempre se utilizó un Split de entrenamiento de 60% para entrenamiento, 20% para validación, y 20% para prueba. Lo que nos dio del total de 520 pacientes, 312 para el set de entrenamiento, 104 para el set de validación, y 104 para el set de prueba.

El valor de esta división de datos puede cambiarse, pero para la implementación, me pareció lo más adecuado. Aunque existen otros valores de partición, la división 60-20-20 me pareció un buen balance para este dataset, ya que obtiene suficiente información para el entrenamiento sin sacrificar la calidad de las evaluaciones en validación y prueba.

III (b): Escribir el algoritmo.

Para escribir el algoritmo, debemos de tener claro lo que queremos lograr. Necesitamos un **algoritmo de regresión logística**, que involucra varias funciones, sin usar sklearn (este lo utilizaremos en los algoritmos con framework). Para este desarrollo, me basé en las notas de clase[4], y un tutorial de Geeks4Geeks[7].

Mi objetivo es utilizar los datos limpios y procesados para…

Entradas comunes: **params:** un vector de parámetros, los pesos de nuestro modelo en otras palabras. **samples:** matriz de features. Nuestros valores de la tabla. **Labels:** etiquetas verdaderas de nuestros resultados. Las salidas esperadas basadas en y.

Funciones para el algoritmo de regresión logística, apoyado de un artículo de StackOverflow[8]:

$$Sigmoid\ Function: \quad g(z) = \frac{1}{1+e^{(-z)}}$$

$$Hypothesis: \quad h_\theta(x) = \frac{1}{1+e^{(-\theta^T x)}}$$

Funciones de gradiente descente y funcion de costo.

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x), y)$$

$$\mathrm{Cost}(h_\theta(\mathbf{x}), y) = -y log(h_\theta(\mathbf{x})) - (1-y) log(1-h_\theta(\mathbf{x}))$$

def sigmoid(z)	Toma como entrada una matriz de vectores. Np.exp(-z) calcula la exponencial negativa que queremos para nuestra función de hipótesis. Convierte la combinación en una probabilidad de 1 o 0.
def h(params, samples)	Nuestra función de hipótesis. Multiplica cada muestra por los parámetros. Aplica sigmoid para tener resultados en 1 o 0.
def cost_function(params, samples, y)	Nuestra función de costo o entropía cruzada. Y es la etiqueta verdadera (1 o 0) que queremos comparar con las predecidas. Calcula la perdida con las muestras de las clases 1 o 0.
def GD(params, samples, y, a)	Aplica gradiente descendiente. Calcula el gradiente de costo de la función con respecto a los parámetros. Actualiza los parámetros con el valor que definimos de Alpha (a en nuestro código). El modelo aprende con cada epoch.
def predict(params, samples)	Obtiene las predicciones finales y decide si es clasificado como 1 o 0, dependiendo del umbral de 0.5.
def accuracy(params, samples, labels)	Calcula la exactitud (accuracy) del modelo. Compara las predicciones generadas por predict con las etiquetas reales labels. Devuelve el porcentaje de aciertos (entre 0 y 1).

III (c): Tecnica de regularización usando L2 (Ridge regression).

En problemas de aprendizaje de maquina es muy común que un modelo se ajuste demasiado a los datos de entrenamiento. A esto se le llama overfitting, que discutiré más adelante en los resultados de mi primer modelo. Para reducir el problema, se usan las técnicas de regularización.Para este ajuste, me base en tutoriales enfocados en regularización en machine learning.[9]

Para este requerimiento, decidí utilizar la técnica de regularización **L2, tambien conocida como ridge regression.**

$$\mathrm{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{m} w_i^2$$

Lo que hace es agregar al término del costo original una penalización proporcional al cuadrado de los parámetros del modelo. El hiperparámetro *lambda* controla la fuerza de esta penalización.

En la implementación del código, lmbda_reg es aplicada a la función de costo y a la de gradiente descendiente. Añadiendo esta penalización para afectar los pesos. En esta implementación, vario valores de lambda fueron experimentados para obtener los resultados deseados.

IV. RESULTADOS DEL MODELO BASE (SIN FRAMEWORK)

Para correr el programa, es primero necesario generar la tabla adaptada para nuestro modelo de entrenamiento

ejecutando el código de limpiar_datos.py. Esto nos genera la tabla diabetes_data_modified.py

Una vez que la tabla está adaptada para que nuestro código pueda implementarla, se corre el código logreg_base.py. Este código crea las primeras predicciones (sin framework) y despliega los resultados en un arreglo de tablas utilizando **matplotlib**.

IV (a): Definición de Parámetros.

A continuación, voy a definir los parámetros e hiperparámetros que se utilizaron para el primer diseño del modelo. Estos son los valores "modificables" de mi modelo. Los cuales cambié para la implementación mejorada.

Epochs = 5000 | a (alpha) = 0.01 | lambda_reg = 0.01

Explicación de los parámetros:

Epochs: Representa cuantas veces el algoritmo recorre el conjunto de datos de entrenamiento. Un valor de 5000 fue seleccionado para asegurar que el modelo tuviera suficiente tiempo de aprendizaje. Un valor pequeño causaría underfitting, ya que el modelo no esta lo suficientemente entrenado. Mientras que demasiadas causarían overfitting.

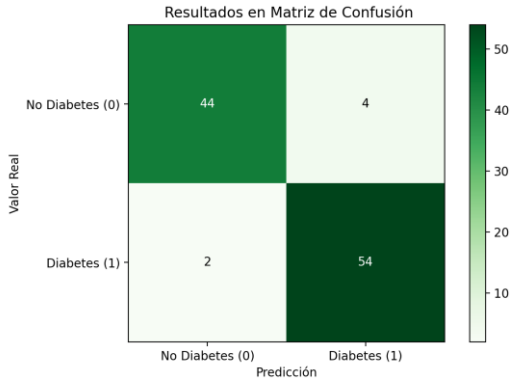
Alpha: la taza de aprendizaje. Controla la función de gradiente descendiente al actualizar los parámetros. Un valor muy pequeño haría el entrenamiento muy lento, mientras que un valor grande afectaría negativamente los resultados de la función de costo. Un valor de 0.01fue elegido al inicio.

Lambda_reg: El hiperparámetro de regularización, para la técnica L2 (ridge regression). Evita que el modelo se sobreajuste a los datos.

IV (b): Resultados de las Gráficas.

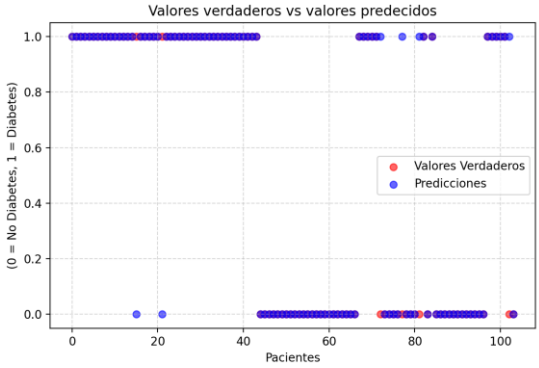
A continuación, voy a mostrar los resultados de las graficas y de los datos numéricos generados por mi primer código logreg_base.py.

Logreg_base.py | figura 01 | Matríz de Confusión



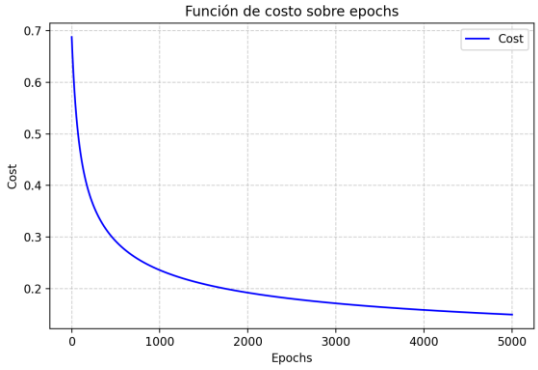
Predice con alto nivel de exactitud los pacientes que tienen y no tienen diabetes. Con un nivel bajo de falsos positivos y negativos (predicción de 2 sin diabetes que en realidad padecen de diabetes, y predicción de 4 con diabetes, que en realidad no padecen de diabetes).

logreg_base.py | figura 2 | Valores verdaderos vs valores predecidos



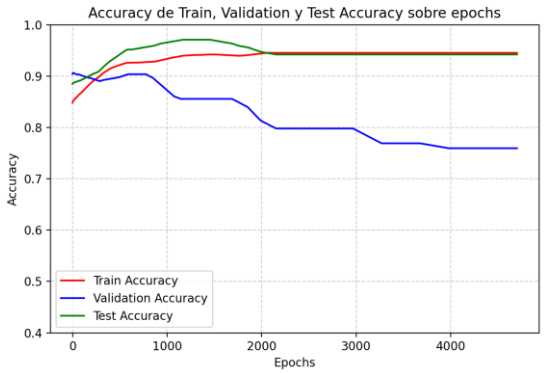
Esta gráfica muestra resultados similares a la matriz de confusión, solamente de una forma que podemos ver como se calificaron los 104 pacientes del set de predicciones mientras eran clasificados. Podemos ver pocos puntos rojos que representan los que no fueron clasificados correctamente.

logreg_base.py | figura 3 | Función de costo sobre epochs



Muestra la disminución del costo mientras avanzan los epochs de entrenamiento. Al inicio cae rápidamente, lo que nos hace entender que el modelo aprende rápido. Después, se suaviza, convergiendo en 0.15. Para futuros modelos seria buena idea reducir el modelo de epochs para evitar sobreentrenmaiento.

logreg_base.py | figura 4 | Accuracy de train, val y test sobre epochs.



Train (línea roja) alcanza valores altos, lo que indica que el modelo aprende bien de los patrones de entrenamiento. La línea azul (val) se mantiene alta, pero cae rápidamente, lo que nos indica que es una señal de overfitting como mencionaré después. Test (línea verde) se mantiene alta, lo que nos hace entender que el modelo aprende bien de datos nuevos.

Logreg_base.py | Resultados | R2, MSE, Bias, Varianza.

```
Epochs                Final Test Accuracy: 94.23%                Epochs
Train: R²=0.778, MSE=0.042 | Val: R²=0.262, MSE=0.154 | Test: R²=0.807, MSE=0.048
Bias (aprox): 0.051 | Varianza (aprox): 0.189
```

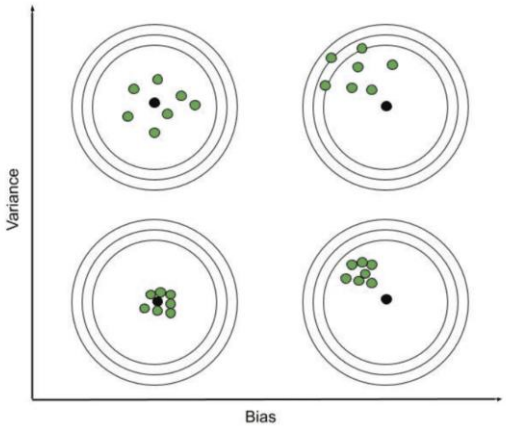
Al final de mis gráficas, se reporta el valor final de mis resultados. Estos incluyen el valor de accuracy final de test (la línea verde de mi grafica), el R2 y el MSE (mean squared error) de los sets de entrenamiento, y el el bias (sesgo) y varianza final de mis gráficas.

El accuracy final del set de test, fue un 94.23%, lo que lo hace muy bueno. Esto se puede ver en los resultados de mi matriz de confusión, Los valores de R^2 nos indican que la predicción es mejor mientras el valor se acerca a 1. Y el MSE mejor mientras más se acerca a 0.

Es aquí donde podemos ver que el accuracy de nuestro modelo es bueno en las fases de entrenamiento y prueba, pero falla en validación. Esto se debe a que el modelo se ajusto demasiado bien a los datos de entrenamiento, lo que muestra un ligero sobreajuste o **overfitting.** Para mejorar esto, fue útil **el método de regularización** que utilice y detalle previamente. Aunque no es perfecto, es un área de mejora que puedo ajustar para mi modelo mejorado.

IV (c): Cálculo de Sesgo y Varianza

Finalmente, tenemos que calcular el sesgo y la varianza de nuestro proyecto. Para esto, me apoyé de las notas de clase[4] y varios tutoriales. (notas y referencias). El objetivo de mi proyecto es lograr que la varianza y el bias sean los más bajos posibles para evitar problemas de ajuste del modelo. Estos valores pueden ir cambiando para obtener un punto intermedio que nos dé los mejores resultados.



El **Sesgo (bias)** se define como el error que comete un modelo definido por el valor precedido y el valor real. Un alto sesgo indica que el modelo se adapta demasiado a nuestro modelo con valores calculados de una forma demasiada preestablecida. Un sesgo más bajo indica más flexibilidad En mi modelo el bias final fue de 0.051, lo que nos indica un bias bajo y que el modelo se adaptó exitosamente a los datos.

La **varianza**, se define como la sensibilidad del modelo a dispersión o variabilidad de las predicciones. Una varianza muy alta da señal de overfitting, ya que el "modelo" se memoriza los datos en vez de aprender de los patrones. Una varianza demasiado baja indica que el modelo es menos sensible a las variaciones del entrenamiento. Mi modelo final nos mostró una varianza de 0.189, lo que nos indica que tiene varianza media a alta, dándonos entender que tenemos un ligero problema de overfitting.

IV (d): Nivel de Ajuste del Modelo (underfit u overfit?)

En aprendizaje de máquina, es importante que nuestro modelo no muestre casos de overfitting o underfitting para el entrenamiento. En casos como el nuestro, donde tenemos un sesgo bajo y una varianza media-alta, podemos observar un caso de overfitting ligero. Podemos verlo de la forma en la que aunque el accuracy es de 94.23% en nuestro set de prueba, el desempeño en

validación es peor. Esto lo podemos corregir parcialmente ajustando valores como lambda en la técnica de regularización. Sin embargo el desempeño de mi modelo es bueno considerando todo.

V. DESARROLLO DEL MODELO MEJORADO

Archivo: logreg_mejorado.py

Para el desarrollo del modelo mejorado, simplemente tomé como base logrreg_base.py y ajusté los hiperparámetros, modificándolos y probando diferentes versiones para poder obtener un mejor resultado. Mi objetivo fue llevar el accuracy lo más cercano a 100% posible, y la varianza y bias lo más cercano a 0%.

Para este nuevo modelo, la estrategia fue cambiar los valores de epochs, a (Alpha) y lambda (regularización L2) para obtener mis resultados deseados. Después muchos intentos sin mejoras (varios de los modelos empeoraron), la decisión fue cambiar el datasplit de 60-20-20 a 75-15-15. Esto fue una excelente decisión, ya que el resultado del accuracy subió de 94.23% a 98.08%. El bias bajo a 0.074 y la varianza a 0.036.

Todo con un menor numero de epochs. Los resultados de los hiperparámetros nuevos son los siguientes:

Hiperparametros nuevos (98.08% accuracy)

Epochs = 5000 | a (alpha) = 0.01 | lambda_reg = 0.01

Data Split: 70% Train, 15% val, 15% test.

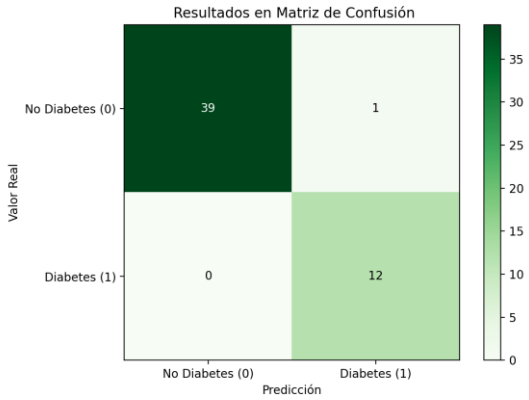
Hiperparametros viejos (94.23% accuracy)

Epochs = 2000 | a (alpha) = 0.05 | lambda_reg = 0.2

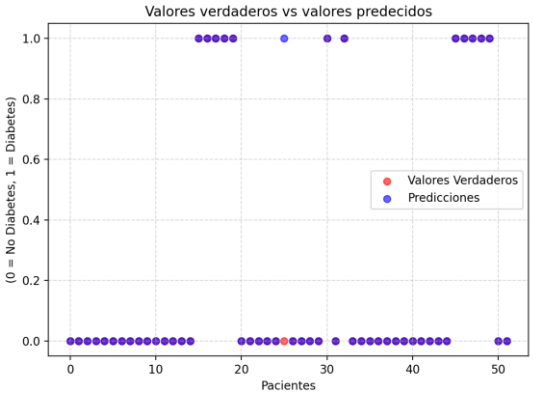
Data Split: 70% Train, 15% val, 15% test.

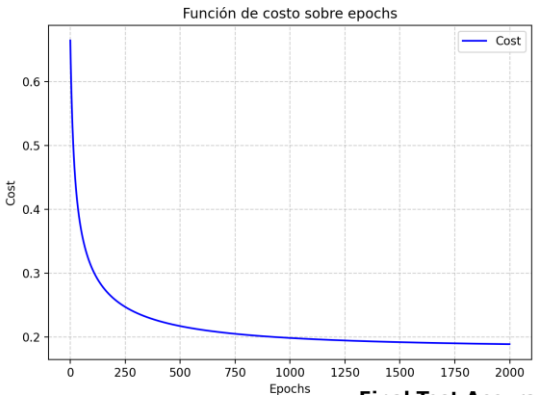
VI. RESULTADOS DEL MODELO MEJORADO

A continuación mostrare los resultados de las tablas nuevas y mejoradas. Con una explicación breve de cada una. Y como cambio el modelo con los nuevos ajustes.

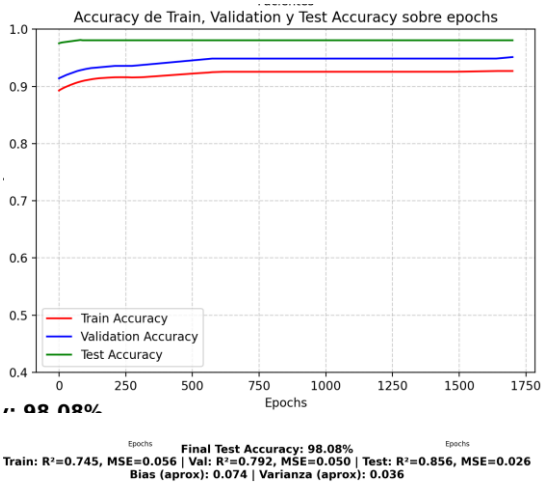


Aquí podemos ver como la cantidad de falsos positivos y negativos es reducida significativamente.





En nuestra tabla de función de costo, observamos como el modelo aprende mucho más rápido al inicio. Es por esto que decidí bajar el numero de epochs para evitar problemas de overfitting. Además de que una cantidad más grande de epochs es innecesaria.



En la tabla final, observamos que en nuestra validación, aprende mucho mejor, esto se debe a que durante la fase de entrenamiento, el modelo tuvo acceso a una mayor cantidad de datos, lo que le permitió aprender patrones más representativos de los pacientes. Reducir el modelo de epochs evitó que el modelo se sobreentrenara. El valor de Alpha fue reducido para que el aprendizaje fuera más estable. El valor de lambda fue ajustado para que la penalización del modelo de regresión fuera menor. Y evitar que el modelo se sobreajustara.

Al final, estoy satisfecho con los resultados de mi modelo mejorado, ya que permitió un grado de varianza y sesgo menor, y un accuracy más alto.

VII. DESARROLLO DEL MODELO CON FRAMEWORKS

Archivo: framework_xgb.py

Para mi último modelo, voy a utilizar dos frameworks requeridos. Estos son las librerías de scikit-learn (sklearn) y XGBoost.

Utilizar frameworks para nuestros modelos de aprendizaje de maquina ofrecen una alta variedad de ventajas, no solo para los resultados, si no para el desarrollo de futuros proyectos utilizando herramientas avanzadas para hacer más fácil el entrenamiento, evaluación y optimización de modelos de predicción.

Sklearn: Las librerías de scikit-learn, me proporcionaron múltiples ventajas. Entre ellas la división de de los datos en conjuntos de entrenamiento, validación y prueba, donde utilicé la función StandardScaler(), además de r2score() y mean_square_error() de sklearnmetrics para calcular los valores de R^2 y MSE que hice manualmente en mi primer código. En el futuro, me gustaría utilizar sklearn más seguido para simplificar mi proceso, utilizando librerías para agilizar la forma en la que programo algoritmos de aprendizaje de máquina.

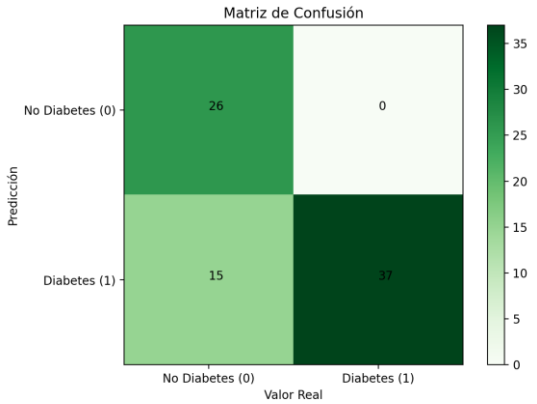
XGBoost: XGBoost es una herramienta poderosa que se basa en el algorimo de Gradient Boosting construyendo arboles de decisión.[12] Aquí, el uso de árboles de decisión es similar al algoritmo de **gradiente descendiente** utilizado en el primer modelo, donde cada árbol intenta corregir los errores de arboles anteriores, permitiéndole al modelo aprender. Fue impresionante ver lo complejo y flexible que es la librería de XGBoost, y ofrece mucho que aprender.

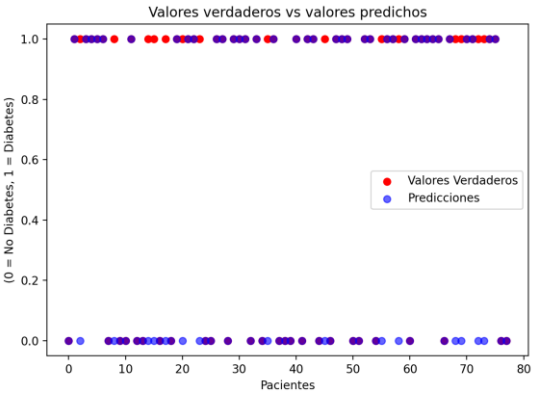
Parámetros de XGBoost: La forma en la que intenté mejorar mi modelo fue mediante la mejora de los parámetros de XGBoost.[13] Durante el desarrollo de este proyecto, intenté modificar y agregar parámetros a mi modelo para mejorar los resultados. Algunos llegaron a mejoras, mientras que otros empeoraron el modelo. Algo que me pareció interesante, es que dentro de los parámetros, se incluyen varios que utilice en mi modelo base, tales como “eta” que es el **learning rate**, y las **técnicas de regularización** como L1 y L2 que use en mi modelo anterior mediante parámetros como reg_lambda y reg_alpha.

Después de muchos intentos, pude obtener un grado de accuracy de 80.77%, lo cual no fue lo mejor, en especial utilizando XGBoost, pero fue satisfactorio para aprender y observar cómo funciona el modelo.

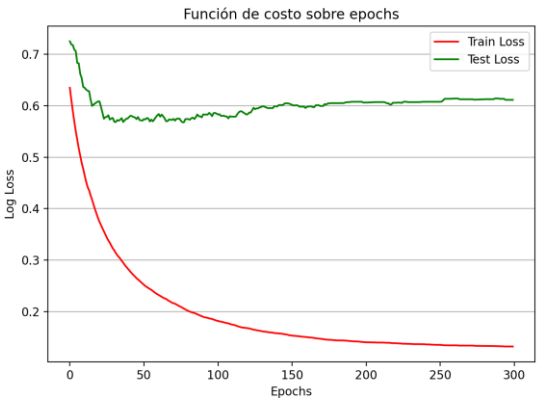
VIII. RESULTADOS DEL MODELO CON FRAMEWORKS

Finalmente, voy a mostrar los resultados del modelo utilizando los frameworks de sklearn y XGBoost. Los resultados son menores a los esperados. Ya que el modelo nuevo tuvo un ajuste complexo con los hiperparámetros dentro del alogritmo de XGBoost.

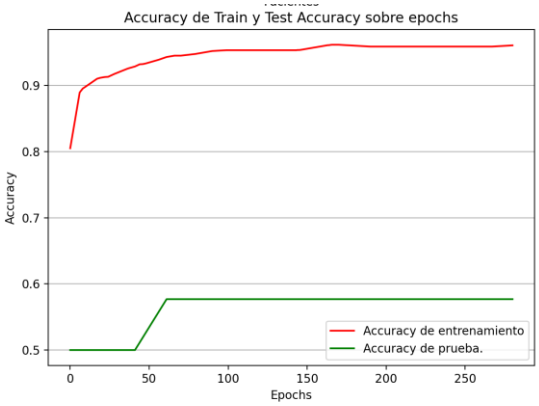


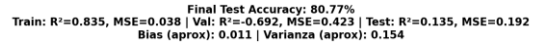


Un detalle que noté, es que el modelo implementado con XGBoost era excelente al evitar nombrar pacientes que en realidad tienen diabetes como si no tuvieran diabetes, lo cual es perfectamente conveniente para los pacientes y el uso del modelo. Pero lo mismo no se puede decir del caso inverso.



Aquí Podemos observar que aunque el modelo aprende exitosamente del set de daos de entrenamiento, tuvo problemas con el set de prueba.





Al final, el accuracy final de 80.77% no es lo mejor que pudo llegar en modelo, en especial con la capacidad de XGBoost. Para futuros trabajos, me gustaría seguir experimentando con esta herramienta para llegar a modelos mejores entrenados.

El bias fue bastante bajo, algo sorprendente. Pero la varianza llego a un nivel medio-alto.

XI. CORRECCIONES

Durante la retroalimentación de este proyecto, tuve que hacer correcciones para que el desempeño de los modelos y mi aprendizaje de la parte teoría pueda ser refinada. Aquí hay una lista de las correcciones que hice a lo largo del desarrollo de mis algoritmos de aprendizaje de maquina:

Mejorar el algoritmo con XGBoost: Antes de la revisión final de mis archivos. Mi algoritmo con XGBoost me presentaba problemas al clasificar erróneamente a los pacientes con diabetes. Tras analizar el error y ajustar la configuración de hiperparámetros, logré corregir este comportamiento y obtener resultados más consistentes. La versión final de este documento ya incluye estas mejoras.

Implementar una técnica de regularización: Una parte faltante de mi proyecto, es implementar una técnica de regularización para mejorar el desempeño de mi modelo. Esto lo hice aplicando L2, que penaliza los pesos excesivamente grandes, evitando así que el modelo cayera en overfitting.

Explicar la significancia del bias, varianza y nivel de ajuste: Algo que debía desarrollar más en mi proyecto, es la significancia, análisis y resultados de mi varianza, bias, y nivel de ajuste (si tenia overfitting o underfitting).

Esto lo desarrollé más en la versión final de este documento.

Pulir las gráficas: Finalmente, la corrección final de mi proyecto fue en la parte estética y de presentación. Hice las gráficas y los resultados de datos más presentables, además de la sintaxis de este documento.

X. CONCLUSIONES

Los modelos implementados fueron una forma muy efectiva de ver como diferentes técnicas de aprendizaje de maquina se pueden utilizar para llegar a objetivos similares. Podemos observar como un modelo puede ser diseñado sin el uso de frameworks con XGBoost, pero a la vez las formas en las que estas herramientas poderosas pueden ayudar a entrenar modelos de formas efectivas.

El uso de clasificación para pacientes de diabetes es solo un ejemplo de muchos en el que el aprendizaje de maquina esta transformando el mundo de la medicina, y como puede mejorar la calidad de vida de personas alrededor del mundo.

NOTAS Y REFERENCIAS

[1] World Health Organization. (2023, April 5). *Diabetes*. World Health Organization. https://www.who.int/news-room/fact-sheets/detail/diabetes

[2] Islam, M., & Ferdousi, R. (2019). *Likelihood prediction of diabetes at early stage using data mining techniques*. Semantic Scholar. https://www.semanticscholar.org/paper/Likelihood-Prediction-of-Diabetes-at-Early-Stage-Islam-Ferdousi/9329dec57c5f13f195220ffa7077fd0029983f07

[3] Brownlee, J. (2019, June 17). *How to implement logistic regression with stochastic gradient descent from scratch with Python*. Machine Learning Mastery. https://machinelearningmastery.com/implement-logistic-regression-stochastic-gradient-descent-scratch-python

[4] Notas de clase. Modulos 2 y 3. Inteligencia Artificial Para la Ciencia de Datos I. Valdés, Benjamín. **En especial el código linear_reg_gd.py**

[5] GeeksforGeeks. (s. f.). Data preprocessing in Machine Learning using Python. GeeksforGeeks. Recuperado el 14 de septiembre de 2025, de https://www.geeksforgeeks.org/machine-learning/data-preprocessing-machine-learning-python. **Para el preprocesamiento de datos.**

[6] W3Schools. (s. f.). NumPy Array Split. W3Schools. Recuperado el 14 de septiembre de 2025, de https://www.w3schools.com/python/numpy/numpy_array_split.asp **Para dividir los datos en train, val, test.**

[7] GeeksforGeeks. (2024, January 3). Implementation of logistic regression from scratch using Python. GeeksforGeeks. https://www.geeksforgeeks.org/machine-learning/implementation-of-logistic-regression-from-scratch-using-python. **Recurso para escribir el algoritmo de regression logística.**

[8] Stack Overflow. (2017, December 13). Logistic regression gradient descent. Stack Overflow. https://stackoverflow.com/questions/47795918/logistic-regression-gradient-descent. **Recurso adicional para las fórmulas de regresión logística.**

[9] GeeksforGeeks. (s. f.). Regularization in Machine Learning. GeeksforGeeks. Recuperado el 14 de septiembre de 2025, de https://www.geeksforgeeks.org/machine-learning/regularization-in-machine-learning/ **Recurso para la implementación de la técnica de regularización L2.**

[10] GeeksforGeeks. (s. f.). Bias-Variance in Machine Learning. GeeksforGeeks. Recuperado el 14 de septiembre de 2025, de https://www.geeksforgeeks.org/machine-learning/bias-vs-variance-in-machine-learning/ **Recurso de apoyo para calcular el bias y la varianza.**

[11] Hands-On Machine Learning: Logistic Regression with Python and Scikit-Learn. Ryan and Matt Data Science. https://www.youtube.com/watch?v=aL21Y-u0SRs **Recurso utilizado para el framework de scikit-learn.**

[12] Prashant111. (s. f.). XGBoost K-Fold CV & Feature Importance. Kaggle. Recuperado el 14 de septiembre de 2025, de https://www.kaggle.com/code/prashant111/xgboost-k-fold-cv-feature-importance **Recurso adicional para el desarrollo con el framework de XGBoost.**

[13] GeeksforGeeks. (s. f.). XGBoost Parameters. GeeksforGeeks. Recuperado el 14 de septiembre de 2025, de https://www.geeksforgeeks.org/machine-learning/xgboost-parameters/ **Recurso utilizado para modificar los parámetros de XGBoost.**

REFERENCIAS ADICIONALES

Lista de referencias utilizadas para la escritura del código.

[1] GeeksforGeeks. (2024, January 19). Matplotlib tutorial. GeeksforGeeks.https://www.geeksforgeeks.org/python/matplotlib-tutorial/ **Tutorial que utilicePara modificar las tablas de MatPlotLib.**