Instituto Tecnológico y de Estudios Superiores de Monterrey

Inteligencia artificial avanzada para la ciencia de datos I

Grupo 101

22 de Octubre 2023

Momento de Retroalimentación: Selección, configuración y entrenamiento del modelo

Autores:

Catherine Johanna Rojas Mendoza A01798149

Adrian Pineda Sánchez A00834710

Max A0

Rogelio Lizárraga Escobar A01742161

Rodolfo Jesús Cruz Rebollar A01368326

Profesor:

Edgar Gonzalez Fernandez

Índice

Identificación del tipo de modelo que se requiere para resolver el problema del reto.

El objetivo del reto es predecir si un pasajero sobrevivió (1) o no (0) al hundimiento del Titanic, lo que implica que la variable objetivo (Sobrevivio) es binaria. (Kaggle, n.d.) Esto significa que estamos tratando de clasificar cada pasajero en una de dos categorías: sobrevivió o no sobrevivió, por lo que el tipo de modelo que se requiere para resolver el problema del reto del Titanic es un **modelo de clasificación binaria.**

Modelos Adecuados para Clasificación Binaria

1. **Regresión Logística**
    - Es un modelo simple y efectivo para problemas de clasificación binaria.
    - Proporciona probabilidades que se pueden convertir fácilmente en predicciones binarias.
    - Es interpretable, permitiéndote entender el impacto de cada variable en la probabilidad de supervivencia. (IBM, n.d.-a)
2. **Árboles de Decisión**
    - Pueden capturar relaciones no lineales entre las características y la variable objetivo.
    - Pueden ser propensos al sobreajuste si no se controlan adecuadamente.
    - Son fáciles de interpretar y visualizar. (IBM, 2021)
3. **Random Forest**
    - Es un conjunto de árboles de decisión que mejora la precisión y reduce el sobreajuste.
    - Captura interacciones complejas entre las características.
    - Tiende a tener un buen rendimiento en una variedad de problemas de clasificación. (IBM, n.d.-c)
4. **Redes Neuronales**
    - Pueden ser útiles si se tiene una gran cantidad de datos y relaciones muy complejas entre las variables. (for Developers, n.d.)

Identificación del tipo de datos los modelos que son compatibles con ellos.

Tipo de Datos

1. **Variables Numéricas Continuas**
    - Edad: Representa la edad de los pasajeros. Es un valor continuo, aunque existen datos faltantes.
    - Tarifa: Representa el costo del boleto. También es un valor continuo.

2. **Variables Numéricas Discretas:**
    - Clase de Ticket: Un proxy para el estatus socioeconómico (1 = Alta, 2 = Media, 3 = Baja). Aunque es una variable numérica, es categórica ordinal.
    - Número de Hermanos/Esposos a bordo: Número de hermanos o cónyuges que un pasajero tenía a bordo.
    - Número de Padres/Hijos a bordo: Número de padres o hijos que un pasajero tenía a bordo.
3. **Variables Categóricas:**
    - Sexo: Género del pasajero (Masculino, Femenino).
    - Embarcación: El puerto donde el pasajero abordó el Titanic (C para Cherburgo, Q para Queenstown, S para Southampton).
    - Cabina: Número de cabina del pasajero, aunque tiene muchos valores faltantes. Es categórica y es difícil de manejar directamente.
4. **Variables de Identificación y Texto:**
    - Nombre: El nombre del pasajero es un campo de texto, y aunque no parece relevante para la predicción, se utiliza para extraer información adicional (por ejemplo, títulos como Mr., Mrs., Miss, que pueden dar información sobre el género o el estado civil).
    - Boleto: El número del boleto, es un campo de texto o mixto, y típicamente no se utiliza directamente en los modelos.
5. **Variable Objetivo:**
    - Sobrevivió: Indica si un pasajero sobrevivió o no (0 = No, 1 = Sí). Es una variable categórica binaria, lo que indica que el problema es de clasificación binaria.

Modelos compatibles

Dado el tipo de datos disponibles, los modelos compatibles y adecuados son aquellos que pueden manejar tanto variables numéricas como categóricas, y son capaces de realizar clasificación binaria:

1. **Regresión Logística**
2. **Árboles de Decisión**
3. **Random Forest**

4. **K-Nearest Neighbors (KNN)**

5. **Redes Neuronales**

Posibles modelos para resolver el problema y selección del más adecuado

En un primer tratamiento de datos (Colab, n.d.-a), nos enfocamos en limpiar, transformar, y preparar el conjunto de datos para aplicar varios modelos de machine learning.

Dentro de las principales acciones realizadas estan:

1. **Importación y Renombrado de Columnas:**
   - Se importaron los datos de entrenamiento y prueba desde un repositorio en GitHub.
   - Se renombraron las columnas al español para facilitar el manejo de los datos.
2. **Manejo de Valores Nulos:**
   - Se identificaron valores nulos en columnas clave como Edad y Tarifa.
   - Se imputaron los valores nulos en Edad utilizando la media y desviación estándar según el título (Mr, Mrs, etc.), extrayendo esta información de la columna Nombre.
   - Se imputaron valores faltantes en Tarifa utilizando la media y desviación estándar para la clase del ticket correspondiente.
3. **Transformación de Variables:**
   - Se aplicó One-Hot Encoding a la columna `Sexo`, mapeando `male` a 0 y `female` a 1.
   - Se creó una columna de clasificación de edad (`Clasificacion_Edad`) para agrupar a los pasajeros en categorías como Bebé, Niño, etc.
   - Se añadió una columna `Familiares`, que suma `Hermanos_Esposos` y `Padres_Hijos`.

Dado el hecho de que el problema principal a resolver en el reto radica en clasificar con la mayor precisión posible a los pasajeros que iban a bordo del barco Titanic, en el sentido de determinar mediante un modelo predictivo si cada uno de los pasajeros sobrevivió o no al desastre, se probaron diferentes modelos compatibles con los datos buscando clasificar de una forma binaria la sobrevivencia o no sobrevivencia de los pasajeros, mismos que se enuncian a continuación:

1. **Regresión logística:** Este modelo nos es adecuado para problemas de clasificación binaria y es especialmente útil cuando se espera una relación lineal entre las características (como Edad, Sexo,

Clase Ticket, Familiares) y la probabilidad de supervivencia. Además, la regresión logística es interpretable, lo que permite entender el impacto de cada característica en las predicciones (IBM, 2023).

2. **Árboles de Decisión:** Los árboles de decisión son útiles para capturar relaciones no lineales y no requieren escalar los datos o manejar las variables categóricas mediante encoding. Son adecuados para este conjunto de datos mixto (numérico y categórico). (de Datos, n.d.)

3. **Random Forest:** Como un conjunto de árboles de decisión, el Random Forest mejora la estabilidad y generalización del modelo. Es especialmente útil para manejar grandes conjuntos de datos con alta dimensionalidad y ruido. (IBM, n.d.-c)

4. **K-Nearest Neighbors (KNN):** Es un modelo simple que clasifica basándose en la similitud entre los datos. Es adecuado cuando las relaciones entre las variables no son lineales y cuando el conjunto de datos no es excesivamente grande. (IBM, n.d.-b)

5. **Red Neuronal Feedforward (FNN):** Se implementó una red neuronal con capas densas (fully connected) para predecir la supervivencia, pues las redes neuronales son potentes para capturar patrones complejos en los datos, aunque requieren más datos y poder de cómputo. Son útiles cuando se espera que las relaciones entre las variables sean no lineales y complicadas. (Education, 2023)

6. **LSTM (Long Short-Term Memory):** Se probó una arquitectura LSTM para ver si podría capturar patrones que otros modelos no logran, y aunque las LSTM son generalmente usadas para secuencias temporales, su capacidad de recordar y aprender patrones complejos en los datos las hace útiles en este caso.(GeeksforGeeks, 2024)

7. **XGBoost con Optuna:** Se optimizó el modelo XGBoost usando Optuna, probando varias configuraciones de hiperparámetros para encontrar el mejor rendimiento, pues este es una potente técnica de boosting que maneja bien datos numéricos y categóricos. Es especialmente útil en competencias como Kaggle debido a su capacidad para maximizar la precisión a través de una combinación de árboles de decisión. Optuna se utilizó para optimizar hiperparámetros de manera automática. (Para, 2021)

Cada modelo aplicado fue seleccionado por su capacidad para manejar el tipo de datos (mixto entre numérico y categórico) presente en el conjunto del Titanic y por su idoneidad para resolver un problema de clasificación binaria. Los modelos más simples, como la regresión logística y los árboles de decisión, ofrecen interpretabilidad, mientras que modelos más complejos, como Random Forest y XGBoost, ofrecen un

mayor poder predictivo a costa de una mayor complejidad. Las redes neuronales, aunque potentes, fueron exploradas como una posibilidad para capturar patrones complejos en los datos. Sin embargo, tomando en cuenta lo anteriormente mencionado, cabe mencionar que uno de los modelos más adecuados que se implementaron fue el de regresión logística, esto debido a que más allá de ofrecer una facilidad significativa al momento de interpretar los resultados derivados del modelo, también dicho modelo tiene la capacidad de generar predicciones altamente precisas con el entrenamiento adecuado, mismo que es necesario realizar por medio de datos que tengan clases balanceadas, es decir, que haya la misma cantidad de datos que correspondan a cada una de las clases al momento de entrenar el modelo, esto con el fin de evitar que las predicciones del modelo estén sesgadas.

En un segundo tratamiento de datos (Colab, n.d.-b), realizamos nuevas acciones para lograr mejores resultados en los modelos.

1. **Importación y Preparación de Datos:**
    - En este caso, solo se eliminó la columna Boleto.
    - El tratamiento de la columna Sexo varía. En el primer código, se realizó una simple transformación de Sexo a una variable binaria (0 y 1), mientras que en este se creó una columna separada para Femenino y Masculino.
2. **Manejo de Valores Nulos:**
    - El manejo de otros valores nulos varía. En el código anterior, los valores nulos en Edad y Tarifa se imputaron utilizando una técnica de imputación basada en la distribución normal, mientras que en este código se realizó una imputación similar pero con un enfoque ligeramente diferente al calcular los valores de manera más explícita.
3. **Transformación y Creación de Nuevas Características:**
    - Se crean más columnas binarias para variables categóricas (Primera Clase, Segunda Clase, S, Q, etc.) mediante One-Hot Encoding, lo que da más flexibilidad a los modelos para capturar relaciones.
    - También introduce una nueva variable llamada *Solo_Viaje*, que indica si un pasajero viajaba solo, lo que no estaba presente en el código anterior.
4. **Escalamiento y Reducción de Dimensionalidad:**
    - Se realizó una reducción de dimensionalidad utilizando PCA (Análisis de Componentes Principales), mientras que el código anterior no se aplicó ninguna técnica de reducción de

dimensionalidad. El PCA se utilizó aquí para reducir la dimensionalidad a seis componentes principales, explicando el 90 % de la variabilidad de los datos.

Para este caso, se volvieron a probar los modelos de Regresión Logística, Árboles de Decisión, Random Forest, KNN, y Naive Bayes.

Aplicamos el uso de PCA para la reducción de dimensionalidad buscando mejorar la eficiencia del modelo, lo cual puede ser más adecuado para manejar datos de alta dimensionalidad. Además, este código incluye más columnas categóricas convertidas a variables binarias mediante One-Hot Encoding, lo que potencialmente mejora la capacidad predictiva de los modelos.

El código anterior es más directo y simple, lo que puede ser preferible en situaciones donde la interpretabilidad y simplicidad son más importantes que la optimización extrema. Sin embargo, la falta de reducción de dimensionalidad y la menor cantidad de características pueden limitar su rendimiento en comparación con este código.

Configuración y entrenamiento de los modelos

Entrenamiento de Modelos con Scikit-learn

Se configuraron y entrenaron varios modelos de clasificación utilizando la biblioteca `scikit-learn`. Se utilizaron diferentes algoritmos como Regresión Logística, Árboles de Decisión, Bosques Aleatorios, K-Nearest Neighbors (KNN) y Naive Bayes. A continuación se describe la configuración y el proceso de entrenamiento de cada modelo. Los valores escogidos para cada una de las configuración primera,emte fue a manera de prueba.

Regresión Logística

Para la Regresión Logística, se probaron diferentes configuraciones de hiperparámetros como el valor de C, el algoritmo de optimización (`solver`) y el número máximo de iteraciones (`max_iter`). Los modelos se entrenaron utilizando los datos escalados del conjunto de entrenamiento (`X_train`, `y_train`).

- **Modelo 1:** $C = 0{,}01$, `solver='liblinear'`, `max_iter=1000`.
- **Modelo 2:** $C = 1$, `solver='sag'`, `max_iter=2000`.
- **Modelo 3:** $C = 10$, `solver='newton-cg'`, `max_iter=3000`.
- **Modelo 4:** $C = 0{,}05$, `solver='saga'`, `max_iter=5000`, `penalty='l2'`.
- **Modelo 5:** $C = 0{,}5$, `solver='lbfgs'`, `max_iter=1000`, `penalty='l2'`.

Árboles de Decisión

Se entrenaron varios árboles de decisión utilizando distintos criterios de división (`gini` y `entropy`) y profundidades máximas (`max_depth`).

- **Modelo 1:** `max_depth=30`, `min_samples_split=2`, `criterion='gini'`.
- **Modelo 2:** `max_depth=100`, `min_samples_split=10`, `criterion='entropy'`.
- **Modelo 3:** `max_depth=150`, `min_samples_split=5`, `criterion='gini'`.

Bosques Aleatorios

Para el modelo de Bosques Aleatorios, se exploraron diferentes números de estimadores (`n_estimators`), características máximas (`max_features`) y profundidades máximas (`max_depth`).

- **Modelo 1:** `n_estimators=50`, `max_features='sqrt'`, `max_depth=50`.
- **Modelo 2:** `n_estimators=100`, `max_features='log2'`, `max_depth=100`.
- **Modelo 3:** `n_estimators=200`, `max_features='sqrt'`, `max_depth=200`.

K-Nearest Neighbors (KNN)

El modelo KNN se configuró con diferentes números de vecinos (`n_neighbors`) y esquemas de ponderación (`weights`).

- **Modelo 1:** `n_neighbors=3`, `weights='uniform'`.
- **Modelo 2:** `n_neighbors=7`, `weights='distance'`.
- **Modelo 3:** `n_neighbors=15`, `weights='uniform'`.

Naive Bayes

Finalmente, para el modelo de Naive Bayes, se ajustó el suavizado de varianza (`var_smoothing`).

- **Modelo 1:** `var_smoothing=1e-9`.
- **Modelo 2:** `var_smoothing=1e-8`.
- **Modelo 3:** `var_smoothing=1e-10`.

Modelo	Accuracy	Precisión	Recall	F1-Score	ROC AUC
Regresión Logística (C=0.01, solver='liblinear')	0.775	0.456	0.900	0.605	0.799
Regresión Logística (C=1, solver='sag')	0.770	0.709	0.691	0.700	0.804
Regresión Logística (C=10, solver='newton-cg')	0.766	0.709	0.683	0.696	0.804
Regresión Logística (C=0.05, solver='saga')	0.768	0.671	0.702	0.686	0.801
Regresión Logística (C=0.5, solver='lbfgs')	0.775	0.690	0.708	0.699	0.803
Árbol de Decisión (max_depth=30, criterion='gini')	0.732	0.665	0.640	0.652	0.741
Árbol de Decisión (max_depth=100, criterion='entropy')	0.756	0.639	0.692	0.664	0.771
Árbol de Decisión (max_depth=150, criterion='gini')	0.718	0.627	0.627	0.627	0.765
Random Forest (n_estimators=50, max_depth=50)	0.739	0.677	0.648	0.663	0.794
Random Forest (n_estimators=100, max_depth=100)	0.732	0.684	0.635	0.659	0.796
Random Forest (n_estimators=200, max_depth=200)	0.730	0.677	0.633	0.654	0.797
Random Forest (n_estimators=150, max_depth=250)	0.730	0.671	0.635	0.652	0.797
Random Forest (n_estimators=300, max_depth=300)	0.749	0.684	0.663	0.673	0.802
KNN (n_neighbors=3, weights='uniform')	0.761	0.703	0.677	0.689	0.791
KNN (n_neighbors=7, weights='distance')	0.746	0.684	0.659	0.671	0.792
KNN (n_neighbors=15, weights='uniform')	0.758	0.696	0.675	0.685	0.810
Naive Bayes (var_smoothing=1e-9)	0.768	0.703	0.689	0.696	0.803
Naive Bayes (var_smoothing=1e-8)	0.768	0.703	0.689	0.696	0.803
Naive Bayes (var_smoothing=1e-10)	0.768	0.703	0.689	0.696	0.803

Cuadro 1

Desempeño de los Modelos con Scikit-learn con el primer tratamiento de datos (Colab, n.d.-a)

Modelo	Accuracy	Precisión	Recall	F1-Score	ROC AUC
Regresión Logística (C=0.01, solver='liblinear')	0.773	0.703	0.689	0.7	0.796
Regresión Logística (C=1, solver='sag')	0.768	0.715	0.685	0.7	0.795
Regresión Logística (C=10, solver='newton-cg')	0.768	0.715	0.685	0.7	0.795
Regresión Logística (C=0.05, solver='saga')	0.775	0.709	0.7	0.704	0.797
Regresión Logística (C=0.5, solver='lbfgs')	0.768	0.715	0.685	0.7	0.795
Árbol de Decisión (max_depth=30, criterion='gini')	0.691	0.741	0.571	0.645	0.702
Árbol de Decisión (max_depth=100, criterion='entropy')	0.725	0.722	0.616	0.665	0.739
Árbol de Decisión (max_depth=150, criterion='gini')	0.699	0.741	0.579	0.65	0.729
Random Forest (n_estimators=50, max_depth=50)	0.73	0.741	0.619	0.674	0.79
Random Forest (n_estimators=100, max_depth=100)	0.73	0.741	0.619	0.674	0.78
Random Forest (n_estimators=200, max_depth=200)	0.734	0.753	0.623	0.682	0.779
Random Forest (n_estimators=150, max_depth=250)	0.739	0.747	0.631	0.684	0.779
Random Forest (n_estimators=300, max_depth=300)	0.739	0.741	0.632	0.682	0.783
KNN (n_neighbors=3, weights='uniform')	0.749	0.728	0.65	0.687	0.764
KNN (n_neighbors=7, weights='distance')	0.734	0.734	0.627	0.676	0.778
KNN (n_neighbors=15, weights='uniform')	0.739	0.709	0.64	0.673	0.768
Naive Bayes (var_smoothing=1e-9)	0.737	0.728	0.632	0.676	0.779
Naive Bayes (var_smoothing=1e-9)	0.737	0.728	0.632	0.676	0.779
Naive Bayes (var_smoothing=1e-9)	0.737	0.728	0.632	0.676	0.779

Cuadro 2

Desempeño de los Modelos con Scikit-learn con el segundo tratamiento de datos (Colab, n.d.-b)

Búsqueda de hiperparámetros usando GridSearchCV

Anteriormente se fijaron algunas configuraciones de los hiperparámetros a cada modelo a manera de prueba. En búsqueda de mejorar los resultados se realizó una búsqueda de los mejores hiperparámetros usando GridSearchCV con las siguientes configuraciones:

- **Logistic Regression**
  - `penalty: ['l1', 'l2', 'elasticnet', 'none']`

- `C: [0.001, 0.01, 0.1, 1, 10, 100]`
- `solver: ['liblinear', 'saga', 'lbfgs']`
- `max_iter: [100, 200, 500, 1000]`

- **Decision Tree**
  - `criterion: ['gini', 'entropy']`
  - `max_depth: [None, 10, 20, 30, 50]`
  - `min_samples_split: [2, 5, 10]`
  - `min_samples_leaf: [1, 2, 4]`
  - `max_features: [None, 'auto', 'sqrt', 'log2']`
- **Random Forest**
  - `n_estimators: [50, 100, 200, 500]`
  - `criterion: ['gini', 'entropy']`
  - `max_depth: [None, 10, 20, 30, 50]`
  - `min_samples_split: [2, 5, 10]`
  - `min_samples_leaf: [1, 2, 4]`
  - `bootstrap: [True, False]`
- **Naive Bayes**
  - `var_smoothing: [1e-9, 1e-8, 1e-7, 1e-6]`
- **K-Nearest Neighbors (KNN)**
  - `n_neighbors: [3, 5, 7, 9, 11]`
  - `weights: ['uniform', 'distance']`
  - `algorithm: ['auto', 'ball_tree', 'kd_tree', 'brute']`
  - `leaf_size: [10, 20, 30, 40, 50]`
  - `p: [1, 2]`

GridSearchCV es una herramienta de scikit-learn que busca los mejores hiperparámetros para un modelo de Machine Learning. Realiza una búsqueda exhaustiva probando todas las combinaciones de

hiperparámetros predefinidas y utiliza validación cruzada para evaluar cada una. En este proceso, el conjunto de datos de entrenamiento se divide en varias partes (folds), donde el modelo se entrena en todas menos una y se evalúa en la restante, repitiendo esto para cada fold y combinando los resultados.

Al final, GridSearchCV selecciona la combinación de hiperparámetros que obtuvo el mejor rendimiento promedio en la validación cruzada, devolviendo el modelo optimizado y el mejor puntaje. Esto ayuda a encontrar automáticamente los mejores parámetros para maximizar la capacidad del modelo de generalizar a nuevos datos.

1. Buscando mejores parámetros para **LogisticRegression**...

- Mejor *accuracy*: **0.8024**
- Mejores parámetros: {'C': 0.01, 'l1_ratio': 0.0, 'max_iter': 100, 'penalty': 'l1', 'solver': 'liblinear'}
- **Accuracy** en prueba: **0.7512**

2. Buscando mejores parámetros para **DecisionTree**...

- Mejor *accuracy*: **0.8227**
- Mejores parámetros: {'class_weight': None, 'criterion': 'entropy', 'max_depth': 10, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 5, 'splitter': 'random'}
- **Accuracy** en prueba: **0.7225**

3. Buscando mejores parámetros para **RandomForest**...

- Mejor *accuracy*: **0.8092**
- Mejores parámetros: {'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 10, 'n_estimators': 100}
- **Accuracy** en prueba: **0.7512**

4. Buscando mejores parámetros para **NaiveBayes**...

- Mejor *accuracy*: **0.7946**
- Mejores parámetros: {'var_smoothing': 1e-09}
- **Accuracy** en prueba: **0.7488**

5. Buscando mejores parámetros para **KNN**...

- Mejor *accuracy*: **0.8070**
- Mejores parámetros: {'algorithm': 'auto', 'leaf_size': 10, 'metric': 'euclidean', 'n_neighbors': 9, 'p': 1, 'weights': 'uniform'}
- **Accuracy** en prueba: **0.7273**

Se encontraron buenos accuracys para cada uno de las configuraciones dentro de las validaciones cruzadas pero a la hora de predecir nuestra base de prueba se obtuvieron peores resultados que los de validazión cruzada dentro del GridSearchCV.

Análisis de Resultados

Regresión Logística (C=1, solver='sag')

- **Accuracy:** 0.770
- **Recall:** 0.691
- **Precision:** 0.709
- **F1-Score:** 0.700
- **ROC AUC:** 0.804

Este modelo mostró el mejor equilibrio general, logrando un alto puntaje de F1 y una excelente área bajo la curva ROC, lo que indica una sólida capacidad para distinguir entre clases.

K-Nearest Neighbors (KNN) con n_neighbors=15, weights='uniform'

- **Accuracy:** 0.758
- **Recall:** 0.675
- **Precision:** 0.696
- **F1-Score:** 0.685
- **ROC AUC:** 0.810

Este modelo tuvo la mayor área bajo la curva ROC, lo que sugiere una excelente capacidad para discriminar entre las clases. Sin embargo, su precisión es ligeramente inferior a la del modelo de regresión logística mencionado anteriormente.

Random Forest (n_estimators=300, max_depth=300)

- **Accuracy:** 0.749
- **Recall:** 0.663
- **Precision:** 0.684
- **F1-Score:** 0.673
- **ROC AUC:** 0.802

Este modelo se destacó por su robustez, siendo menos sensible a las variaciones en los datos. Es una opción confiable cuando se busca un equilibrio entre precisión y estabilidad del modelo.

Conclusiones

- **Mejor Modelo Global:** La Regresión Logística con `C=1` y `solver='sag'` es recomendada por su excelente equilibrio entre todas las métricas, especialmente su alta precisión y F1-Score. Esto lo hace ideal para escenarios donde se busca maximizar la veracidad de las predicciones.
- **Mejor Modelo para Discriminación (ROC AUC):** Si el objetivo es maximizar la discriminación entre las clases, el modelo KNN con `n_neighbors=15` es la opción más adecuada gracias a su superior área bajo la curva ROC.
- **Mejor Modelo para Robustez:** El Random Forest con 300 estimadores y una profundidad máxima de 300 es preferible en entornos donde la variabilidad en los datos es alta, proporcionando un rendimiento estable y confiable.

Modelos implementados sin framework

Regresión Logística

Extracción de Características

El primer paso en la implementación de la regresión logística sin framework consiste en extraer las características individuales del conjunto de datos de entrenamiento. Cada característica se almacena en variables independientes, y la variable objetivo se define como el conjunto de etiquetas correspondientes. Las características extraídas se agrupan en una lista para facilitar su manejo en las etapas posteriores del entrenamiento.

Inicialización de Parámetros

Se inicializan los parámetros del modelo de regresión logística, que incluyen un intercepto (θ_0) y coeficientes asociados a cada característica (θ_1, θ_2, θ_3, θ_4).

- **Intercepto (θ_0)**: Inicializado en 2, este valor sirve como la constante base que afecta a todas las predicciones, independientemente de los valores de las características.
- **Coeficientes (θ_1, θ_2, θ_3, θ_4)**: Se inicializan en -2 para todas las características. Este valor negativo busca inicialmente .^{em}pujar"la probabilidad hacia la clase negativa, lo que puede ser útil si se espera que la mayoría de las observaciones pertenezcan a dicha clase. Este enfoque es común cuando no se tiene una fuerte priorización sobre el signo de los coeficientes al inicio.

Además, se define una tasa de aprendizaje (α) de 0.05, que controlará la magnitud de los ajustes realizados durante la optimización de los parámetros. La elección de este valor busca balancear entre la velocidad de convergencia y la estabilidad del modelo.

Entrenamiento mediante Descenso de Gradiente

El entrenamiento del modelo se lleva a cabo mediante el método de descenso de gradiente. Este proceso implica realizar 15,000 iteraciones para ajustar los parámetros del modelo con el fin de minimizar la función de costo. En cada iteración, se calcula la hipótesis del modelo utilizando la función sigmoide y se actualizan los parámetros en función del gradiente y la tasa de aprendizaje. El objetivo es encontrar los valores óptimos de los parámetros que minimicen la diferencia entre las predicciones del modelo y las etiquetas reales.

Parámetro	Valor Final
θ_0	1.1337002278238748
θ_1	-1.3632021301541823
θ_2	22.467810847026554
θ_3	-6.03309438083025
θ_4	0.2182455786775828

Cuadro 3

Valores finales de los parámetros θ después de 15,000 iteraciones

Predicción y Evaluación del Modelo

Una vez completado el entrenamiento, se evalúa el modelo de regresión logística utilizando un conjunto de datos de prueba. Para cada observación en el conjunto de prueba, se calcula la probabilidad de pertenencia a la clase positiva utilizando la función sigmoide, y se redondea para obtener la predicción final (0 o 1).

Posteriormente, se calculan las métricas de desempeño del modelo, incluyendo precisión, recall, F1-Score y accuracy.

Resultados

Las métricas obtenidas para el modelo de regresión logística sin framework son las siguientes:

- **Accuracy:** 0.7703
- **Recall:** 0.6835
- **Precisión:** 0.7013
- **F1-Score:** 0.6922

Estas métricas indican un buen rendimiento del modelo, demostrando que la implementación manual es capaz de producir un modelo funcional y preciso.

LSTM sin Optuna

Este código implementa un modelo de aprendizaje profundo utilizando una arquitectura LSTM (Long Short-Term Memory) para realizar una clasificación binaria. Posteriormente, se explicarán los pasos realizados:

Configuración de `EarlyStopping`

Se define un callback llamado `EarlyStopping` que monitoriza la función (*loss*) en los datos de validación durante el entrenamiento. El entrenamiento se detendrá si la pérdida en los datos de validación no mejora después de 50 épocas consecutivas. Esta técnica es útil para prevenir el sobreajuste, ya que evita que el modelo entrene durante demasiadas épocas una vez que ha dejado de mejorar. Además, se restauran los pesos del modelo correspondientes a la mejor época obtenida durante el entrenamiento.

Reformateo de los datos de entrenamiento y validación

Dado que las redes LSTM esperan entradas en un formato específico, los datos de entrenamiento y prueba se reestructuran en un formato tridimensional. En este caso, cada muestra se ajusta a una forma que representa una secuencia temporal con una sola entrada.

Definición del modelo LSTM

El modelo se define utilizando una arquitectura secuencial. Se incluyen dos capas LSTM:

- La primera capa LSTM tiene 64 unidades y está configurada para devolver secuencias completas, lo que significa que pasará toda la secuencia a la siguiente capa LSTM.

- La segunda capa LSTM tiene 32 unidades y está configurada para devolver solo la última salida en la secuencia, que se utiliza para la siguiente capa densa.

Después de las capas LSTM, se añade una capa densa con 16 unidades y una función de activación ReLU, seguida de una capa de salida con una única unidad y una función de activación sigmoide para la predicción binaria (si se sobrevive o no).

Compilación y entrenamiento del modelo

El modelo se compila utilizando el optimizador Adam y una función de pérdida de entropía cruzada binaria, adecuada para tareas de clasificación binaria. Durante el entrenamiento, se utilizan 100 épocas con un tamaño de lote de 32 muestras, reservando el 10 % de los datos de entrenamiento para validación. También se emplea el callback `EarlyStopping` para detener el entrenamiento si no hay mejoras en la pérdida de validación.

Evaluación del modelo

Finalmente, el modelo se evalúa en el conjunto de prueba, obteniendo la pérdida y la precisión. Estos valores se imprimen para dar una idea de la efectividad del modelo en datos no vistos.

LSTM con optimización de hiperparámetros por medio de Optuna

Se utiliza Optuna para optimizar los hiperparámetros del modelo (LSTM), pero ahora con PyTorch. Optuna es una biblioteca de optimización de hiperparámetros automatizada, que busca de manera eficiente los mejores hiperparámetros para un modelo de aprendizaje automático.

Definición del Modelo LSTM

Primero, se define una clase `LSTMModel` que hereda de `nn.Module`, el módulo base para todos los modelos de PyTorch. Esta clase tiene un constructor `(__init__)` que inicializa una capa LSTM y una capa lineal (totalmente conectada) para la salida.

- `self.lstm` es la capa LSTM que toma como parámetros el tamaño de la entrada (`input_size`), el tamaño de las capas ocultas (`hidden_size`), el número de capas LSTM (`num_layers`), y el `dropout`, que es la probabilidad de desactivar neuronas para evitar el sobreajuste.
- `self.fc` es una capa totalmente conectada que mapea la salida de la LSTM a la dimensión deseada de salida (`output_size`).

El método `forward` define cómo los datos de entrada `x` pasan a través del modelo. Inicializa los estados ocultos y de celdas a ceros, luego pasa la entrada a través de la capa LSTM y finalmente a través de la

capa totalmente conectada.

Función objetivo de Optuna

La función `objective` es donde se define la optimización de los hiperparámetros con Optuna.

- Se sugieren varios hiperparámetros:
  - `hidden_size`: Tamaño de las capas ocultas, se selecciona un número entero entre 32 y 128.
  - `num_layers`: Número de capas LSTM, se selecciona un número entero entre 1 y 3.
  - `dropout`: Probabilidad de *dropout*, se selecciona un número flotante entre 0.0 y 0.5.
  - `learning_rate`: Tasa de aprendizaje, se selecciona un número flotante en una escala logarítmica entre $1e-4$ y $1e-2$.
- Se define el modelo LSTM con los hiperparámetros sugeridos.
- Se configura la función de pérdida (`nn.CrossEntropyLoss`) y el optimizador (`optim.Adam`).

Entrenamiento y Evaluación del Modelo

El modelo se entrena durante 100 épocas.

- Para cada lote en el conjunto de entrenamiento, se realiza un feedforward, se calcula la pérdida, y se realiza backpropagation para actualizar los parámetros del modelo.
- El modelo se evalúa después del entrenamiento utilizando el conjunto de prueba. Se calculan las predicciones y se mide la precisión (accuracy) del modelo.

Finalmente, la función `objective` devuelve la precisión del modelo como la métrica a optimizar por Optuna.

XGBoost y Optuna

Para implementar el modelo de XGBoost utilizando Optuna en Python, primero se divide el conjunto de datos en dos subconjuntos: uno para el entrenamiento y otro para la evaluación del modelo. En este caso, se utilizó el 10 % de los datos para el entrenamiento y el 90 % restante para la evaluación. Esta división permite evaluar el rendimiento del modelo con un conjunto de datos que no se ha utilizado durante el entrenamiento, proporcionando una medida más realista de su desempeño.

Para configurar el modelo de XGBoost, se define un diccionario de parámetros dentro de una función denominada *objective*. Este diccionario incluye parámetros esenciales como la métrica de evaluación, el

objetivo del modelo (por ejemplo, clasificación o regresión), la verbosidad (para controlar el nivel de detalle del proceso de entrenamiento) y la tasa de aprendizaje (**eta**), entre otros. La función *objective* tiene el propósito de crear y configurar el modelo, realizar predicciones y calcular la precisión al comparar las predicciones del modelo con los datos reales. Esta función también calcula una métrica de error que permite evaluar la calidad de las predicciones.

En el proceso de entrenamiento, se utilizan matrices **DMatrix** para almacenar los datos de entrada del modelo, tanto para el entrenamiento como para la evaluación. La función *train()* de XGBoost se utiliza para entrenar el modelo con los parámetros definidos en el diccionario de configuración. Entre los parámetros importantes para el entrenamiento se incluye *evals*, que es una tupla que contiene los datos de prueba junto con el string *eval*, indicando que se debe evaluar el modelo con estos datos durante el proceso de entrenamiento.

Además, el parámetro *early_stopping_rounds* se establece en 10, lo que indica que el entrenamiento se detendrá si el error de validación no mejora en 10 iteraciones consecutivas. Este mecanismo ayuda a evitar el sobreajuste del modelo y a reducir el tiempo de entrenamiento al detener el proceso cuando el modelo ya no muestra mejoras significativas. El parámetro *verbose_eval* se configura como *False* para que la función *train()* no imprima los valores de la métrica *logloss* en cada iteración, sino solo el mejor resultado al final, lo cual simplifica la salida de la información durante el entrenamiento.

Finalmente, se imprimen los resultados obtenidos con el conjunto de prueba, mostrando el rendimiento del modelo después del proceso de optimización y entrenamiento.

FNN sin Optuna

Se implementa una red neuronal secuencial por medio de Keras y Tensorflow

Configuración de Early Stopping: Se crea una instancia de

`EarlyStopping` llamada `early_stopping` que monitorea la pérdida de validación (`val_loss`). El parámetro `patience` se establece en 50, lo que significa que el entrenamiento se detendrá si la pérdida de validación no mejora después de 50 épocas. El parámetro `restore_best_weights` se establece en `True` para restaurar los pesos del modelo en la mejor época de validación.

Definición de la red neuronal:

Se define un modelo secuencial llamado `model_FNN` utilizando la clase `Sequential` de Keras. El modelo contiene tres capas densas:

- La primera capa densa tiene 64 neuronas y utiliza la función de activación ReLU. La dimensión de entrada se especifica mediante el parámetro `input_dim`, que se establece en el número de

características de los datos de entrenamiento.

- La segunda capa densa tiene 32 neuronas y también utiliza la función de activación ReLU.
- La capa de salida tiene una neurona y utiliza la función de activación sigmoide, adecuada para la predicción binaria.

Compilación del modelo:

El modelo se compila utilizando el optimizador Adam (`optimizer='adam'`) y la función de pérdida de entropía cruzada binaria (`loss='binary_crossentropy'`), que es adecuada para problemas de clasificación binaria. Se especifica la métrica de precisión (`metrics=['accuracy']`) para evaluar el rendimiento del modelo.

Entrenamiento del modelo:

El modelo se entrena en los datos de entrenamiento (`X_train` y `y_train`) utilizando el método `fit`. Se especifican varios parámetros, como el número de épocas (`epochs=1000`), el tamaño del lote (`batch_size=32`), y la proporción de datos reservados para la validación (`validation_split=0.1`). Además, se pasa la instancia de `early_stopping` en el parámetro `callbacks` para habilitar el criterio de parada anticipada.

Evaluación del modelo:

Después del entrenamiento, el modelo se evalúa en el conjunto de prueba (`X_test` y `y_test`) utilizando el método `evaluate`, que devuelve la pérdida y la precisión del modelo en el conjunto de validación.

Impresión de resultados

: Se imprimen los resultados en el conjunto de datos de prueba.

FNN con optimización de hiperparámetros por medio de Optuna

Se realizó una FNN, a la cual se le optimizaron sus hiperparámetros por medio de la biblioteca Optuna

Definición del Modelo FNN

El modelo FNN se implementa como una clase que hereda de una clase base de módulos de red neuronal. Este modelo consiste en varias capas lineales, funciones de activación ReLU y capas de *dropout*, organizadas de acuerdo con los hiperparámetros de entrada como el tamaño de entrada, el tamaño de las capas ocultas, el número de capas, el tamaño de salida y la tasa de *dropout*.

Definición de la Función Objetivo para Optuna

La función objetivo, llamada `objective`, se utiliza para sugerir los hiperparámetros del modelo FNN. Dentro de esta función, se definen y seleccionan de manera automática los valores óptimos para parámetros como el tamaño de las capas ocultas, el número de capas, la tasa de *dropout* y la tasa de aprendizaje. Estos valores se sugieren utilizando métodos específicos de Optuna, que buscan optimizar la arquitectura y los parámetros del modelo de manera eficiente.

Creación y Optimización del Estudio

El estudio se crea utilizando la función de creación de estudios de Optuna, configurada para maximizar un objetivo. Una vez creado, el estudio se optimiza llamando a su método de optimización, que ejecuta varias pruebas para encontrar la mejor combinación de hiperparámetros. Al final del proceso de optimización, se imprimen los mejores hiperparámetros encontrados y la mejor precisión alcanzada por el modelo.

Resultados de las redes neuronales y XGBoost

Modelo	**Accuracy**
LSTM sin Optuna	0.7703
LSTM con Optuna	0.7799
FNN sin Optuna	0.7703
FNN con Optuna	0.7894
XGBoost con Optuna	0.7392

Cuadro 4

Desempeño de las Redes neuronales con Optuna y sin Optuna (Colab, n.d.-a)

Conclusión general

El proceso de configuración y entrenamiento de un modelo de regresión logística sin frameworks sigue los principios fundamentales del aprendizaje automático. Aunque requiere la implementación manual de todos los pasos, este enfoque proporciona un control total sobre el algoritmo y una comprensión profunda de su funcionamiento interno. A pesar de la ausencia de herramientas automatizadas, la implementación manual demuestra ser eficaz en la construcción de un modelo predictivo robusto y preciso.

Por otro lado, observamos que las redes neuronales con sus hiperparámetros optimizados logran los mejores resultados de las redes, siendo 0.7894 la exactitud más alta encontrada. Esto es debido a la librería de Optuna, la cual utiliza técnicas de optimización avanzadas para la obtención de mejores resultados.

Referencias

Colab, G. (n.d.-a). Google Colab Notebook. https://colab.research.google.com/drive/1cj_rKIRlmuKDyPPeqU40_dNGnuScjJFf?usp=sharing#scrollTo=x3RiQC_6lcIT&uniqifier=5

Colab, G. (n.d.-b). Google Colab Notebook. https://colab.research.google.com/drive/1jQpHeUuVqNaBCKZIP-OfbY4r3ZrLSclu?usp=sharing#scrollTo=x3RiQC_6lcIT

de Datos, C. (n.d.). Árboles de Decisión en Python. https://cienciadedatos.net/documentos/py07_arboles_decision_python

Education, T. P. (2023). Redes Neuronales: Aprendizaje y Resolución de Problemas [August 10]. https://thepower.education/blog/redes-neuronales

for Developers, G. (n.d.). Neural Networks - Multiclass Classification. https://developers.google.com/machine-learning/crash-course/neural-networks/multi-class?hl=es-419

GeeksforGeeks. (2024). What is LSTM - Long Short Term Memory? [June 10]. https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/

IBM. (2021). Decision Tree Models [August 18]. https://www.ibm.com/docs/es/spss-modeler/saas?topic=trees-decision-tree-models

IBM. (2023). Regresión Logística. https://www.ibm.com/docs/es/spss-statistics/saas?topic=regression-logistic

IBM. (n.d.-a). ¿Qué es la regresión logística? https://www.ibm.com/mx-es/topics/logistic-regression

IBM. (n.d.-b). K-Nearest Neighbors (KNN). https://www.ibm.com/mx-es/topics/knn

IBM. (n.d.-c). Random Forest. https://www.ibm.com/mx-es/topics/random-forest

Kaggle. (n.d.). Titanic - Machine Learning from Disaster. https://www.kaggle.com/competitions/titanic/data

Para. (2021). XGBoost Stepwise Tuning using Optuna [March 17].

https://www.kaggle.com/code/para24/xgboost-stepwise-tuning-using-optuna