Salud Mental en el Entorno Laboral

"*Mens sana in corpore sano*"

Introducción

En el mundo laboral, las altas exigencias pueden provocar estrés, ansiedad o malestar psicológico. A pesar de su importancia, **existe aún un estigma importante** en torno a los problemas de salud mental, lo que dificulta su detección temprana y el acceso a recursos de apoyo.

Objetivo del trabajo

Desarrollar un **modelo predictivo** capaz de anticipar si una persona podría necesitar tratamiento psicológico, a partir de sus respuestas en una **encuesta anónima** y estandarizada.

El modelo deberá:

- Identificar patrones relevantes.
- Detectar factores de riesgo.
- Ser interpretable y útil en contextos reales.

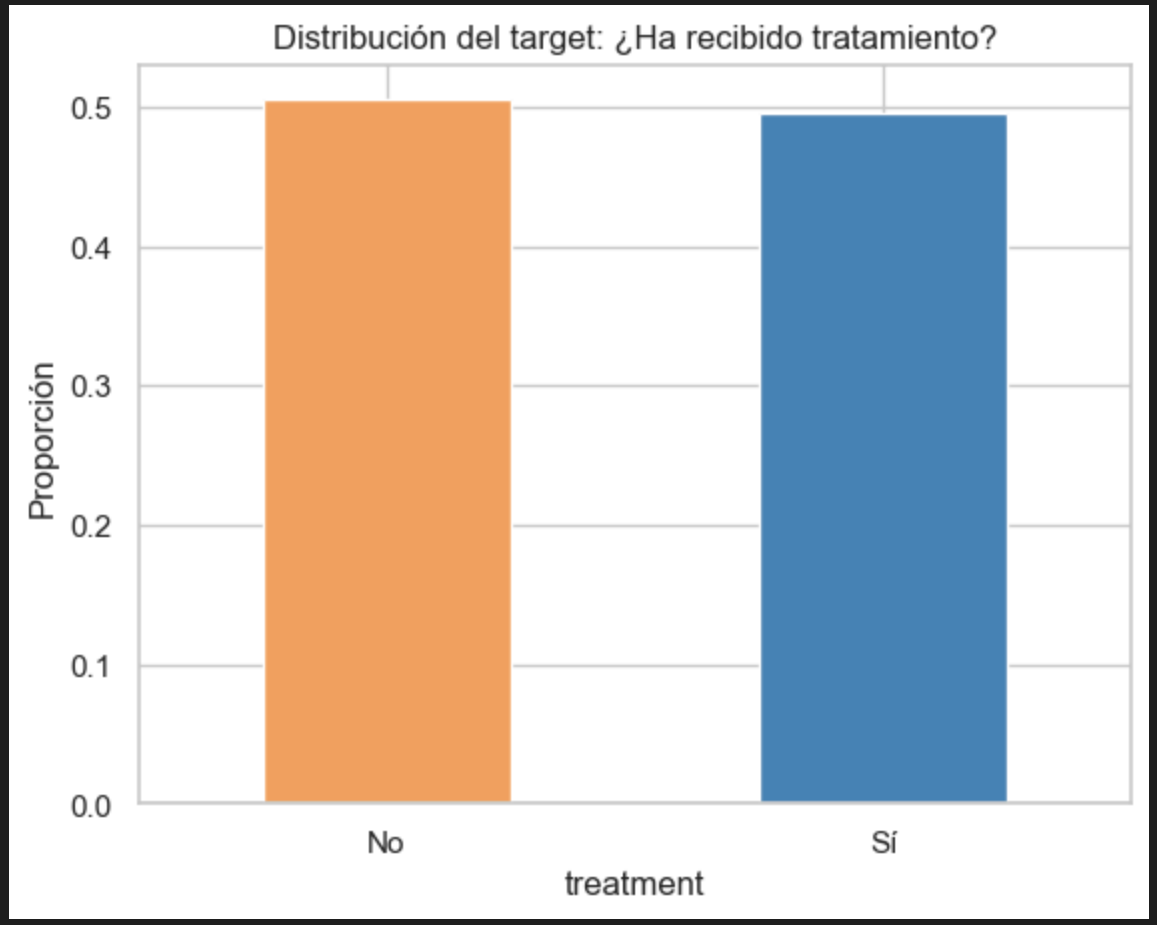
Enfoque y valor

Queremos mostrar cómo el aprendizaje automático puede **aportar valor en ámbitos humanos sensibles**, siempre con responsabilidad y criterio ético.

Análisis exploratorio de datos (EDA)

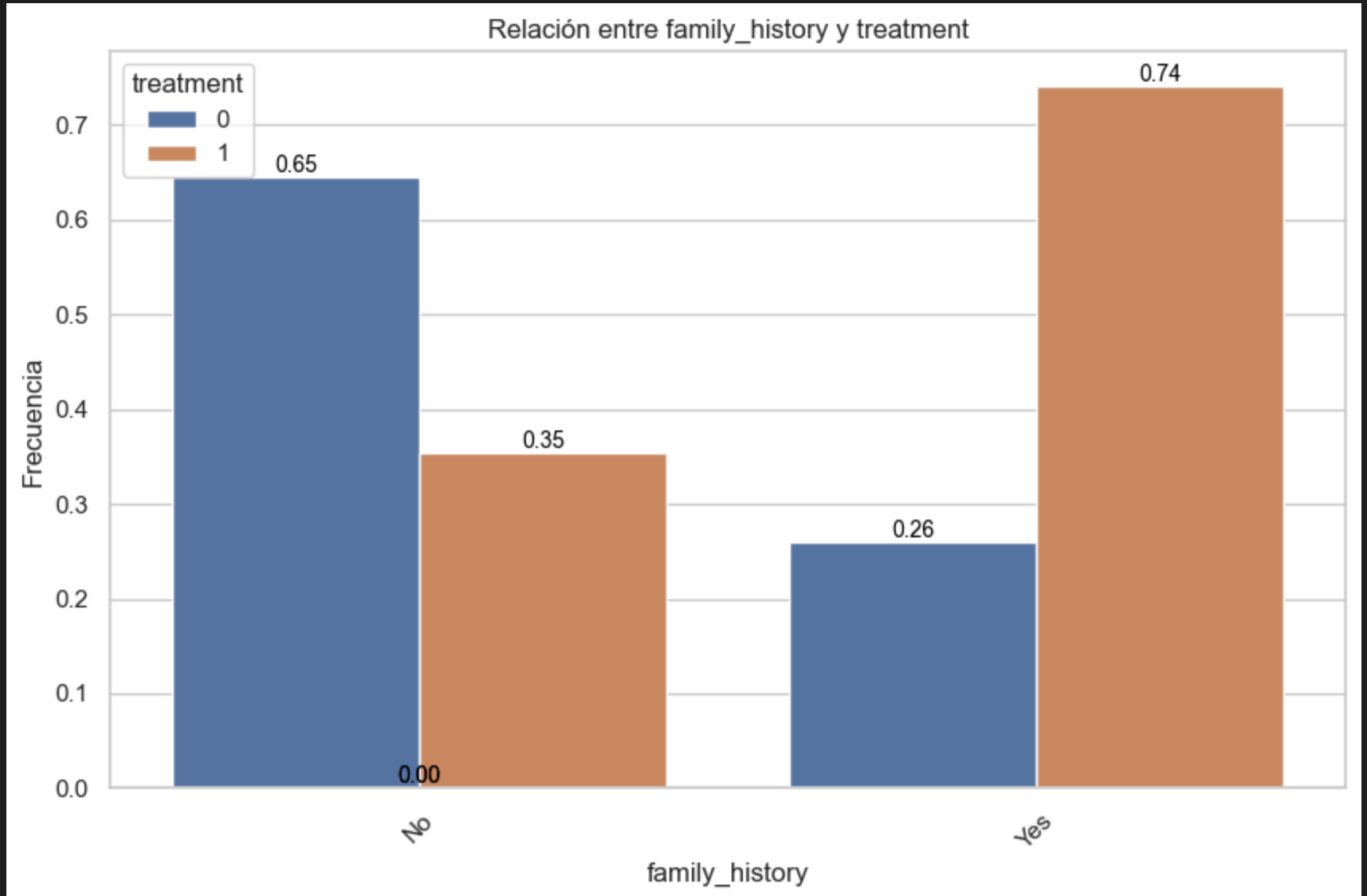
- Exploramos la distribución del target y las variables más relevantes.
- Identificaremos patrones y relaciones útiles para el modelado posterior.

Target: `treatment`

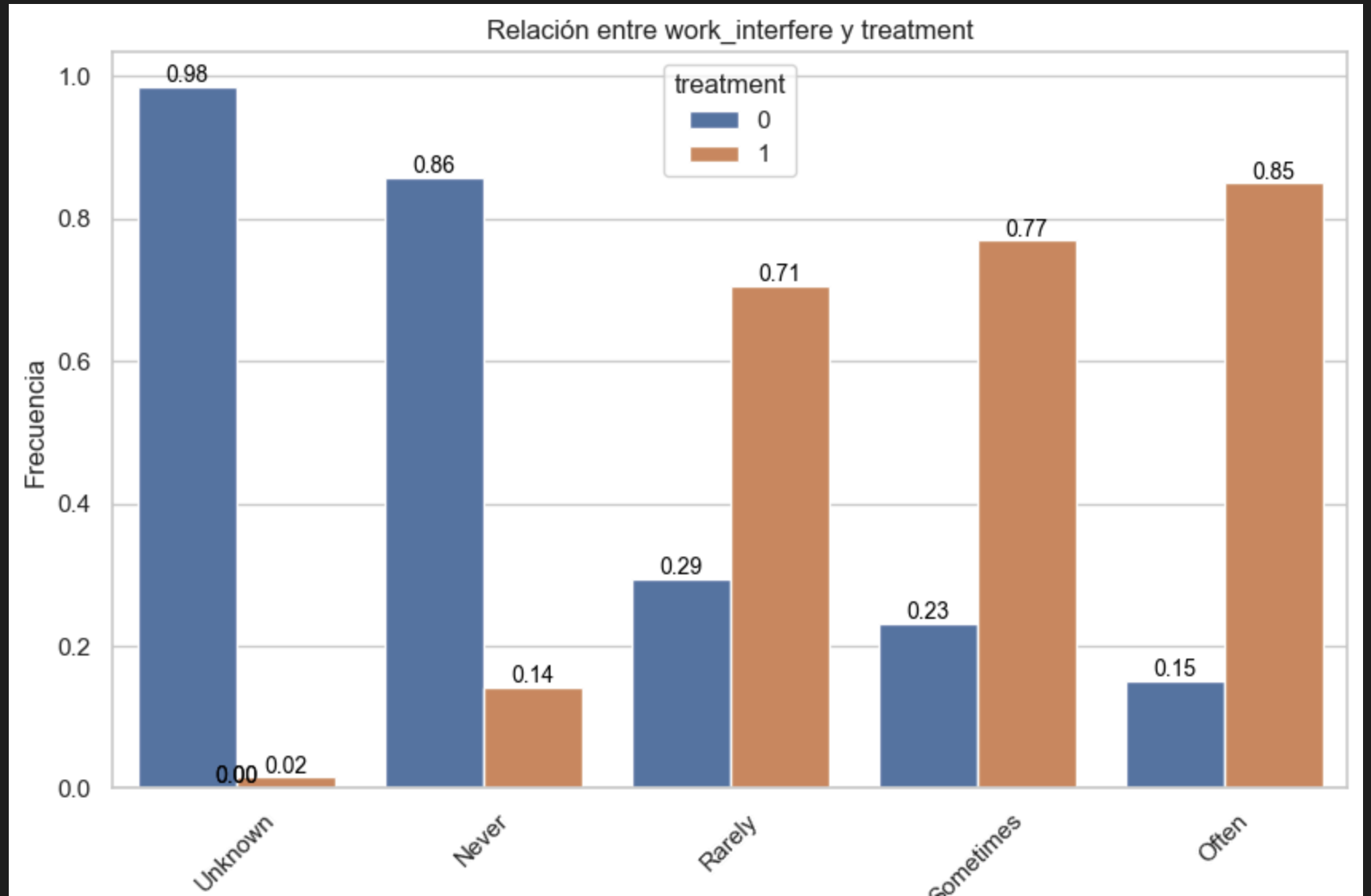


age vs treatment

family_history vs treatment



interfere vs treatment



Feature engineering

En esta fase seleccionamos las variables que se utilizarán como predictores (`X`) y la variable objetivo (`y`), y preparamos los datos para entrenar modelos de clasificación.

Construcción de `X`, `y`

- Separamos las variables por tipo:
  - Ordinales
  - Numéricas
  - Categóricas: variables tipo encuesta
- Creamos los set con lo que trabajaremos:

```
X = df_modificable[ordinales + numericas + categoricas].copy()
y = df_modificable["treatment"]
```

Dos versiones de `X`: sin escalar y escalada

Aplicamos One-Hot Encoding a las categóricas:

```
ohe = OneHotEncoder(drop="first", sparse_output=False)
```

- X_raw: con OneHot, sin escalar age → útil para modelos de árboles
- X_scaled: con OneHot + escalado de age → útil para regresión o KNN

Dos versiones de `X`: sin escalar y escalada

Escalamos `age` con `StandardScaler` en una de las versiones:

```
pre_scaling = ColumnTransformer([
    ("num", StandardScaler(), numericas),
    ("cat", ohe, categoricas)
], remainder="passthrough")
```

Train-test split

Dividimos los datos en conjunto de entrenamiento y conjunto de test para poder **evaluar el rendimiento real** de los modelos sobre datos no vistos.

```
from sklearn.model_selection import train_test_split

X_train_raw, X_test_raw, y_train_raw, y_test_raw = train_test_split(
    X_raw, y, test_size=0.2, stratify=y, random_state=42)

X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled = train_test_split(
    X_scaled, y, test_size=0.2, stratify=y, random_state=42)
```

Baseline: Regresión logística

Usamos un modelo de **regresión logística simple** como baseline.

Como la regresión logística es sensible a la escala, usamos `X_scaled`:

```
baseline = LogisticRegression(max_iter=1000, random_state=42)
baseline.fit(X_train_scaled, y_train_scaled)

y_pred_base = baseline.predict(X_test_scaled)
```

Métricas de evaluación

- **Accuracy**: porcentaje total de aciertos.
- **Precision**: de los casos predichos como positivos, cuántos lo eran realmente.
- **Recall**: de los casos positivos reales, cuántos fueron correctamente identificados.
- **F1 Score**: media armónica entre *precision* y *recall*.

Métricas de evaluación

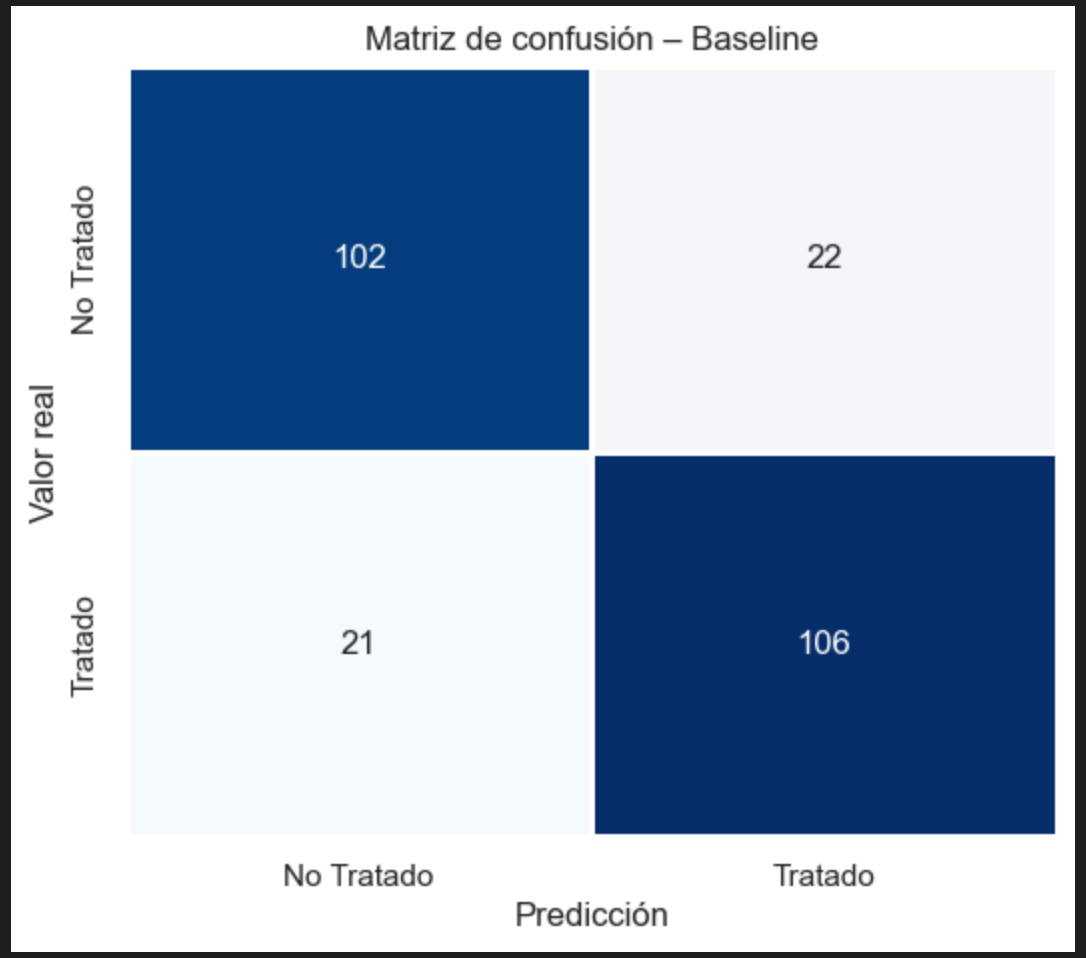
En nuestro contexto, **el recall es especialmente importante**,
ya que nos interesa **detectar correctamente a quienes necesitan tratamiento**.
Aun así, usaremos el **F1 Score** como referencia global por equilibrar ambas.

Métricas del baseline

Métrica	Valor
Accuracy	**0.8287**
Precision	**0.8281**
Recall	**0.8346**
F1 Score	**0.8314**

Estos valores nos servirán como punto de partida para comparar con modelos más avanzados.

Matriz de confusión: baseline



Random Forest

Random Forest es un ensamblado de árboles de decisión que ofrece una buena capacidad de generalización y robustez frente al overfitting.

Utilizaremos en su modelado `X_raw`, sin escalar, y aplicaremos un `GridSearchCV` para encontrar la mejor combinación de hiperparámetros.

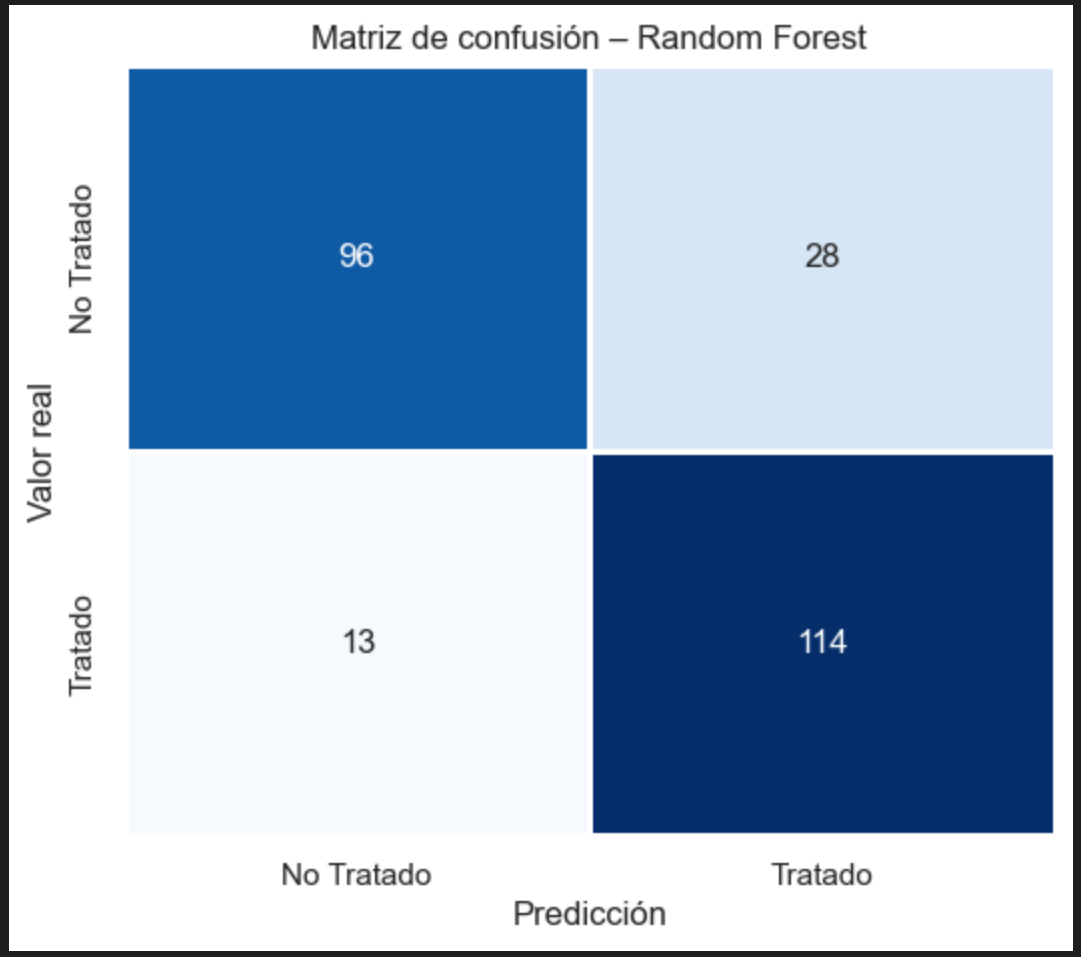
```python
# Instanciamos el modelo
rf = RandomForestClassifier(random_state=42)

# Hiperparámetros optimizados
param_grid = {
    "n_estimators": [100],
    "max_depth": [None],
    "min_samples_split": [5],
    "min_samples_leaf": [1],
    "max_features": ["log2"],
    "bootstrap": [True]
}

# GridSearch con validación cruzada
grid_rf = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    cv=5,
    scoring="f1_macro",
    n_jobs=-1,
    verbose=1
)

grid_rf.fit(X_train_raw, y_train_raw)
y_pred_rf = grid_rf.best_estimator_.predict(X_test_raw)
```

Métricas: Random Forest

Métrica	Valor
Accuracy	**0.8367**
Precision	**0.8028**
Recall	**0.8976**
F1 Score	**0.8476**

En este modelo mejoramos el recall considerablemente, manteniendo un buen equilibrio general.

Matriz de confusión: RF



XGBoost

XGBoost es un modelo de boosting basado en árboles de decisión que ofrece gran rendimiento y control sobre el ajuste, construyendo predictores de forma secuencial para corregir errores del modelo anterior.

Utilizaremos en su modelo `X_raw`, sin escalar, y aplicaremos un `GridSearchCV` para encontrar la mejor combinación de hiperparámetros.

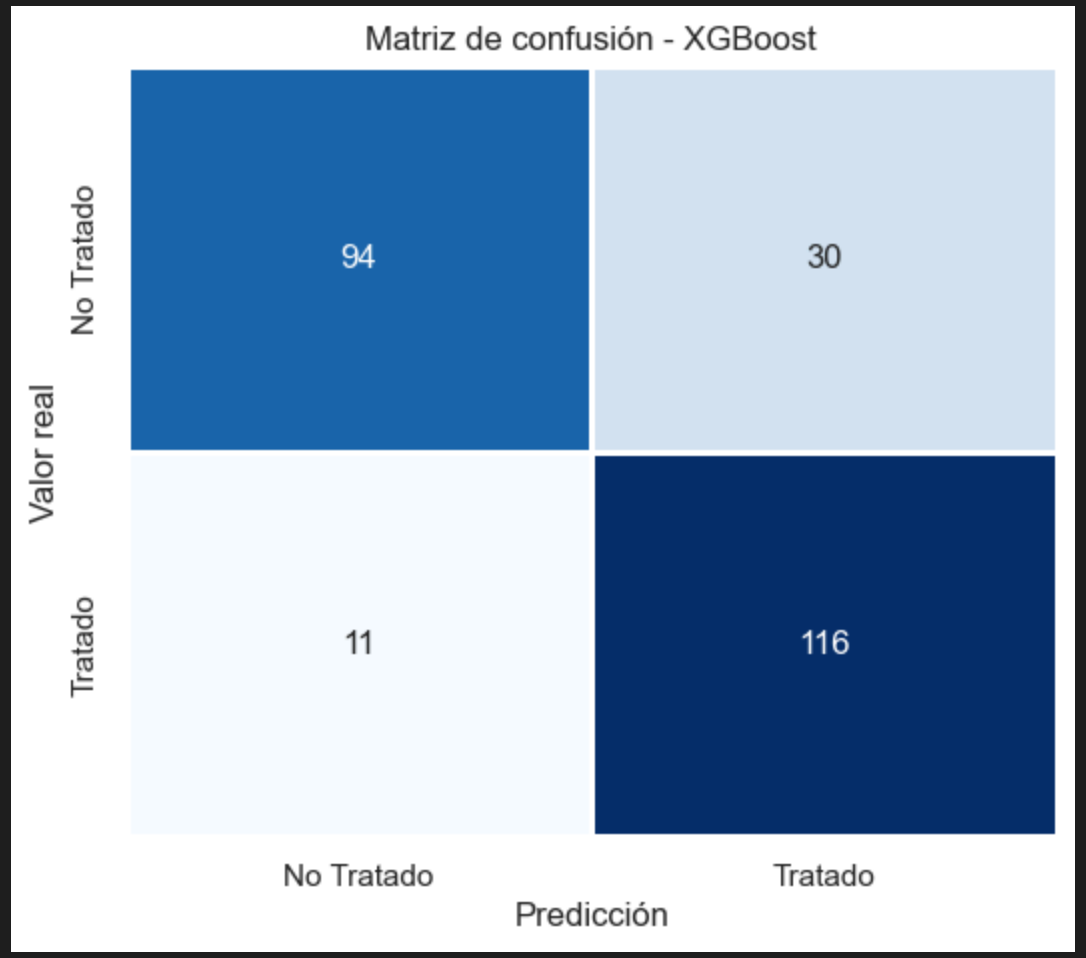
```python
# Instanciamos el modelo
xgb = XGBClassifier(objective="binary:logistic", eval_metric="logloss", random_state=42)

# Hiperparámetros optimizados
param_grid = {
    "n_estimators": [250],
    "max_depth": [3],
    "learning_rate": [0.02],
    "subsample": [1.0],
    "colsample_bytree": [1.0],
    "gamma": [0.05]
}

# GridSearch con validación cruzada
grid_xgb = GridSearchCV(
    estimator=xgb,
    param_grid=param_grid,
    cv=5,
    scoring="f1_macro",
    verbose=1,
    n_jobs=-1
)

grid_xgb.fit(X_train_raw, y_train_raw)
y_pred_xgb = grid_xgb.best_estimator_.predict(X_test_raw)
```

Métricas: XGBoost

Métrica	Valor
Accuracy	**0.8367**
Precision	**0.7945**
Recall	**0.9134**
F1 Score	**0.8498**

En este modelo se logra el **mejor recall** hasta el momento, lo cual es muy relevante para nuestro caso de uso.

Matriz de confusión: XGBoost



Gradient Boosting

Gradient Boosting es un algoritmo que construye árboles de decisión de forma secuencial, donde cada nuevo árbol intenta corregir los errores cometidos por los anteriores.

Utilizaremos en su modelado `X_raw`, sin escalar, y aplicaremos un `GridSearchCV` para encontrar la mejor combinación de hiperparámetros.

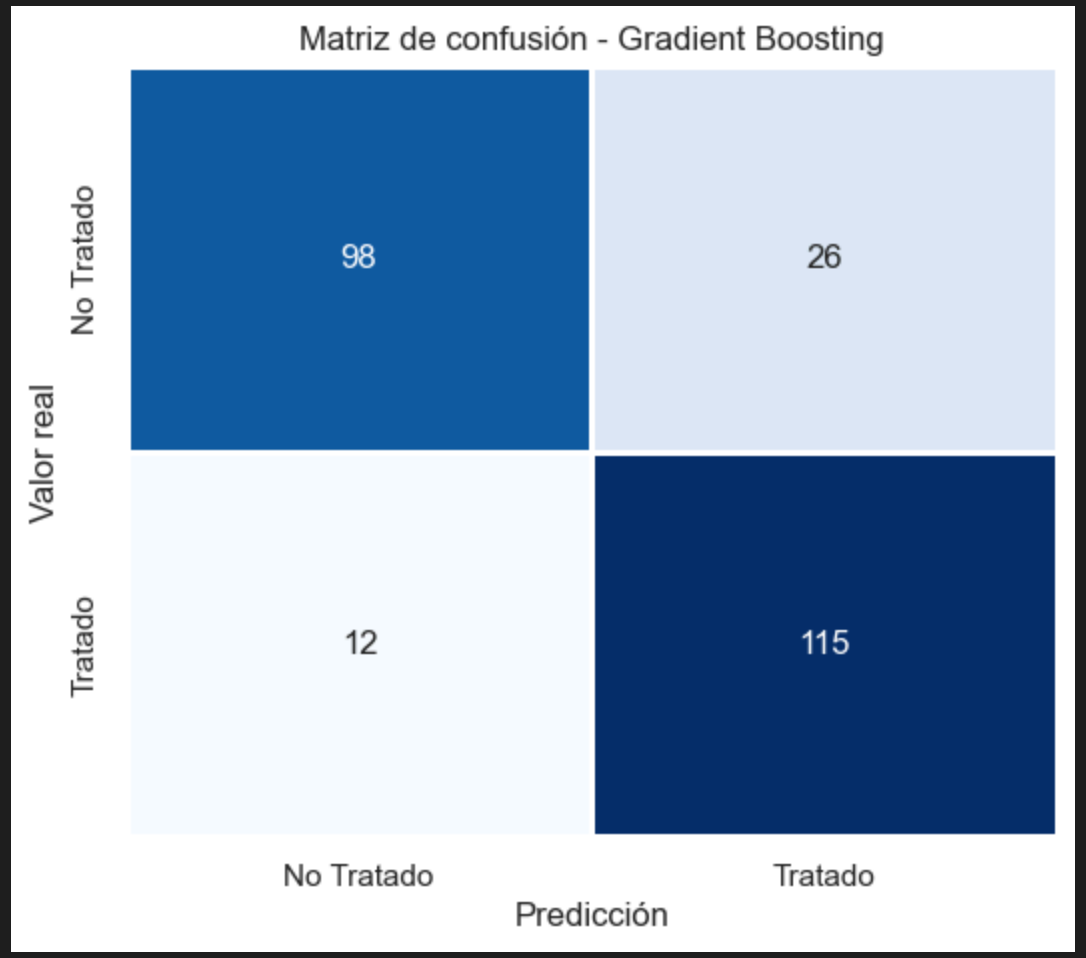
```python
# Instanciamos el modelo base
gbc = GradientBoostingClassifier(random_state=42)

# Hiperparámetros optimizados
param_grid = {
    "n_estimators": [275],
    "max_depth": [4],
    "learning_rate": [0.0075],
    "subsample": [0.9],
    "max_features": ["log2"]
}

# GridSearch con validación cruzada
grid_gbc = GridSearchCV(
    estimator=gbc,
    param_grid=param_grid,
    cv=5,
    scoring="f1_macro",
    n_jobs=-1,
    verbose=1
)

grid_gbc.fit(X_train_raw, y_train_raw)
y_pred_gbc = grid_gbc.best_estimator_.predict(X_test_raw)
```

Métricas: Gradient Boosting

Métrica	Valor
Accuracy	**0.8486**
Precision	**0.8156**
Recall	**0.9055**
F1 Score	**0.8582**

En este modelo mejoramos en prácticamente todos los parámetros, obteniendo el mejor equilibrio general hasta ahora.

Matriz de confusión: GBC



K-Nearest Neighbors (KNN)

K-Nearest Neighbors es un algoritmo basado en la similitud entre instancias. Clasifica según los k vecinos más cercanos.

Es un modelo no paramétrico y muy intuitivo, útil como referencia frente a modelos más complejos. Requiere que los datos estén escalados, por lo que usamos `X_scaled`.

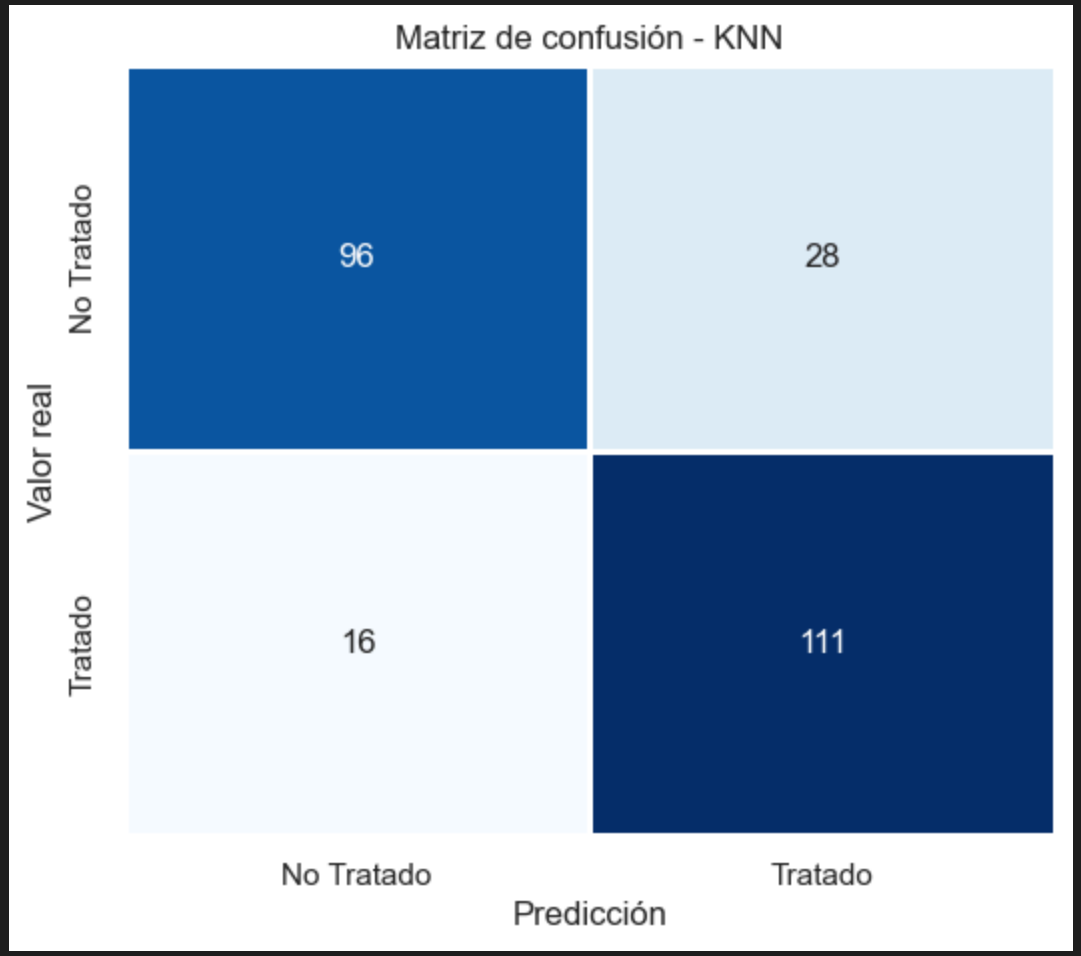
```
# Instanciamos el modelo base
knn = KNeighborsClassifier()

# Hiperparámetros optimizados
param_grid = {
    "n_neighbors": [27],
    "weights": ["uniform"],
    "metric": ["minkowski"],
    "p": [4]
}

# GridSearch con validación cruzada
grid_knn = GridSearchCV(
    estimator=knn,
    param_grid=param_grid,
    scoring="f1_macro",
    cv=5,
    verbose=1,
    n_jobs=-1
)

grid_knn.fit(X_train_scaled, y_train_scaled)
y_pred_knn = grid_knn.best_estimator_.predict(X_test_scaled)
```

Métricas: KNN

Métrica	Valor
Accuracy	**0.8247**
Precision	**0.7986**
Recall	**0.8740**
F1 Score	**0.8346**

KNN ofrece un *recall* decente y una ejecución sencilla, aunque, en general, con peor rendimiento que los demás modelos.

Matriz de confusión: KNN



Comparativa de modelos

Modelo	Accuracy	Precisión	Recall	F1 Score
Baseline	0.8287	0.8281	0.8346	0.8314
RF	0.8367	0.8028	0.8976	0.8476
XGBoost	0.8367	0.7945	0.9134	0.8498
GBC	0.8486	0.8156	0.9055	0.8582
KNN	0.8247	0.7986	0.8740	0.8346

Conclusiones

Gradient Boosting es el modelo seleccionado:

- El mejor **F1 Score**
- La mejor **accuracy**
- El segundo mejor en **precisión** y **recall**

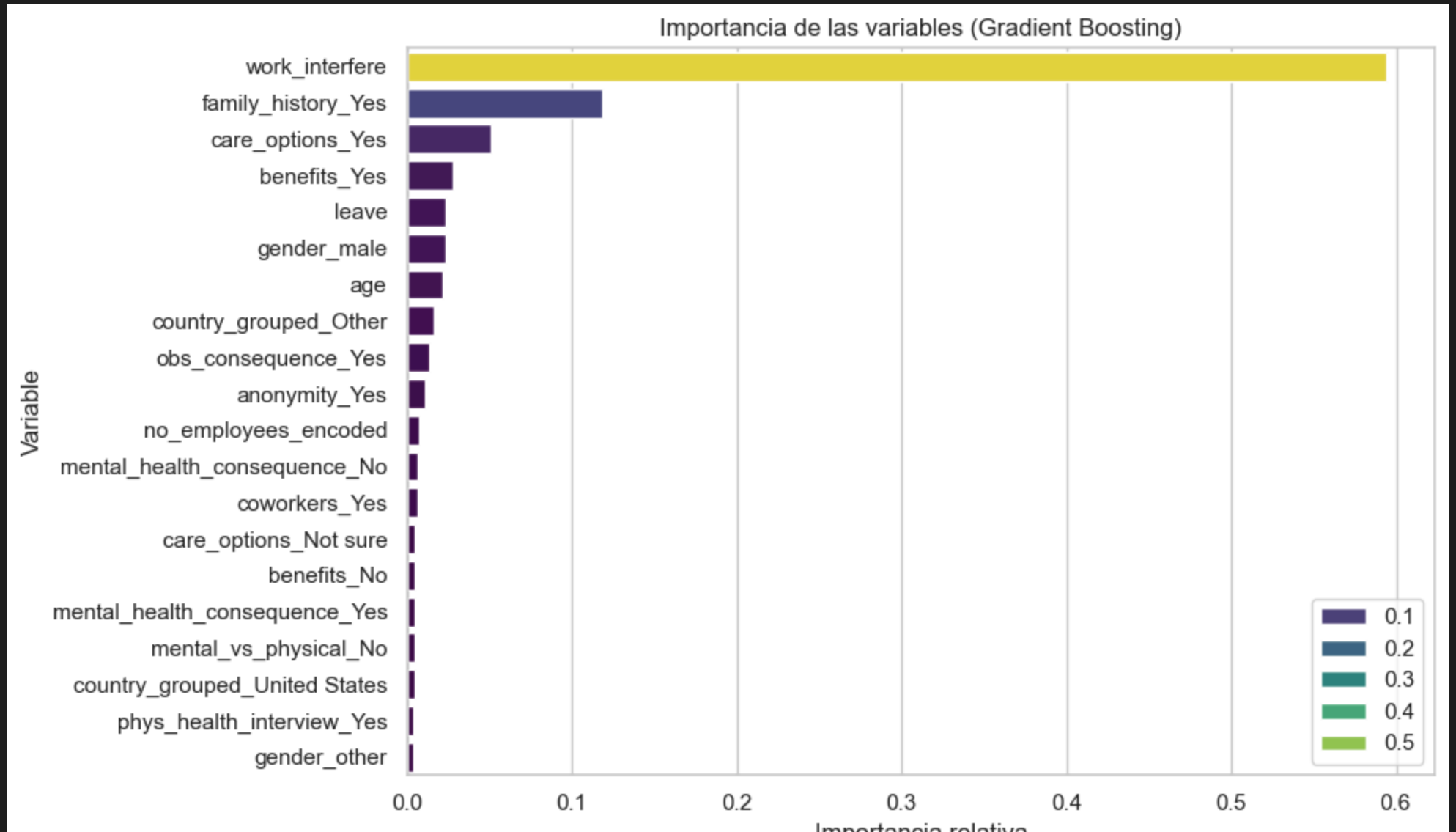
Mejora y análisis del modelo

Comprobar si es posible mantener (o incluso mejorar) el rendimiento reduciendo la complejidad del modelo o comprendiendo mejor su comportamiento.

Análisis de importancia de variables

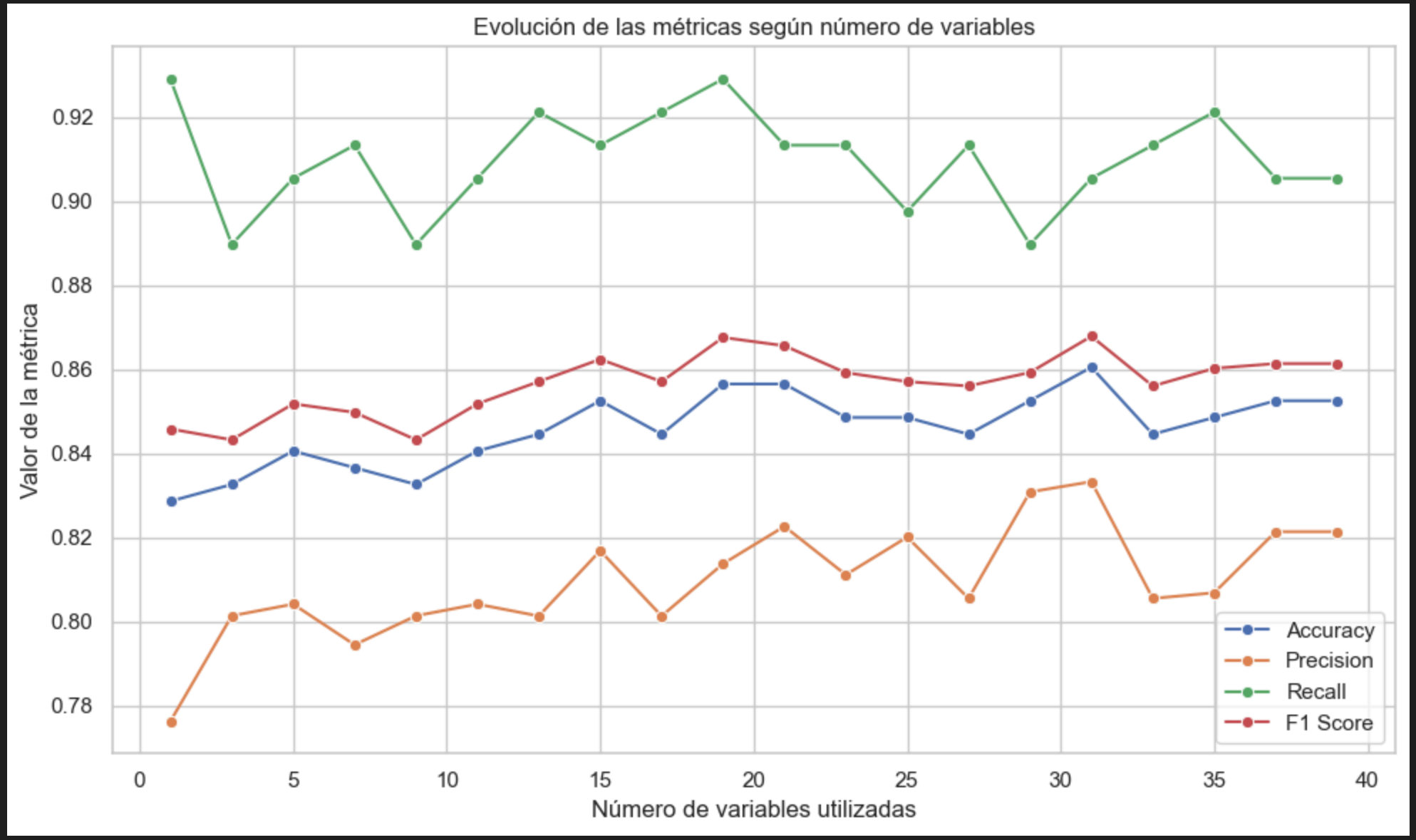
- Detectar qué variables son más influyentes en la predicción.
- Evaluar la posibilidad de eliminar variables con muy poca relevancia.
- Explorar versiones simplificadas del modelo utilizando solo las variables más importantes.

Variables más relevantes



Reentrenamiento con n variables

```python
# Lista para ir guardando los resultados
results = []

# Bucle para ir probando los modelos con n features
for n in range(1, len(feat_imp) + 1, 2):
    top_features = feat_imp["feature"].head(n).tolist()

    # Conjuntos X adaptados a las features elegidas
    X_train_top = X_train_raw[top_features]
    X_test_top = X_test_raw[top_features]

    # Entrenamos con los hiperparámetros óptimos
    model = GradientBoostingClassifier(
        learning_rate=0.0075,
        max_depth=4,
        max_features="log2",
        n_estimators=275,
        subsample=0.9,
        random_state=42
    )
    model.fit(X_train_top, y_train_raw)
    y_pred = model.predict(X_test_top)

    # Guardamos las métricas
    results.append({
        "n_features": n,
        "accuracy": accuracy_score(y_test_raw, y_pred),
        "precision": precision_score(y_test_raw, y_pred),
        "recall": recall_score(y_test_raw, y_pred),
        "f1": f1_score(y_test_raw, y_pred)
    })

# Guardamos en DataFrame
results_df = pd.DataFrame(results)
```


Evolución de las métricas según número de variables
Valor de la métrica
0.92
0.90
0.88
0.86
0.84
0.82
0.80
0.78
0
5
10
15
20
25
30
35
40
Número de variables utilizadas
Accuracy
Precision
Recall
F1 Score

¿Cuántas variables utilizar?

Se detectan **2 picos de rendimiento**:

- Modelo con **31 variables**: mejor **precisión**
- Modelo con **19 variables**: mejor **recall**.

Ambos tienen **accuracy** y **F1 Score** muy similares.

¿Por qué priorizamos el recall?

Nuestro caso trata sobre **salud mental**.

Lo importante es **no dejar pasar** personas que necesiten ayuda.

Preferimos **maximizar el recall**:

- Aunque aumenten los falsos positivos,
- Garantizamos que **nadie que necesite intervención se quede fuera**.

Conclusión: $n = 19$

- Mejor **recall**, clave en este problema.
- Menor complejidad → modelo más **simple e interpretable**.
- Rendimiento muy similar al modelo más grande.

Modelo final

Tras aplicar técnicas de optimización, selección de variables por importancia y validación cruzada, hemos construido un modelo final basado en **Gradient Boosting**.

Este modelo ha sido entrenado utilizando las **19 variables más relevantes** y los **hiperparámetros óptimos**, logrando un gran rendimiento con buena interpretabilidad.

Guardado del modelo

Utilizamos `joblib` para guardar en local el modelo obtenido:

```
# Guardamos el modelo final entrenado
joblib.dump(model_19, "./src/models/optimal_model_mental_health.pkl")
```

Conclusiones (I)

- El mejor modelo es **Gradient Boosting con 19 variables**.
- Presenta el mejor equilibrio entre rendimiento y simplicidad.
- Alta puntuación en F1 Score, precisión y recall.
- Modelo interpretable y robusto, con buen potencial de generalización.

Conclusiones (II)

- Variables más influyentes:
  - `work_interfere`: impacto de la salud mental en el trabajo.
  - `family_history_Yes`: antecedentes familiares.
  - `benefits_Yes`: acceso a recursos en el entorno laboral.
- También destacan variables sociodemográficas: país, edad y género.

Conclusiones (III)

- Permite anticipar casos que podrían requerir intervención psicológica.
- Útil para orientar recursos preventivos en contextos laborales.
- Aporta transparencia gracias al análisis de interpretabilidad.
- Ejemplo claro de cómo aplicar machine learning con impacto social.

Este proyecto va mucho más allá de métricas y predicciones.

Hablamos de personas.

Detectar a tiempo quién necesita ayuda puede marcar la diferencia entre el silencio y la intervención, entre la invisibilidad y el cuidado.

Si los datos pueden ayudarnos a entender mejor la salud mental, tenemos la responsabilidad de usarlos con humanidad, criterio y comprensión.

Gracias