Tecnológico de Monterrey

Campus Estado de México

Materia

Desarrollo de aplicaciones avanzadas de ciencias computacionales

Gpo. 502

Evidencia 1 - Fase 2 - Parte B:

Implementación usando modelos tradicionales

Estudiantes

Eric Manuel Navarro Martínez - A01746219

Gerardo Ríos Mejía - A01753830

Mariluz Daniela Sánchez Morales - A01422953

Pablo Spínola López - A01753922

Profesores

Ariel Ortiz Ramírez

Jorge Adolfo Ramírez Uresti

Miguel González Mendoza

Raúl Monroy

Fecha

14/05/2025

Índice

Optimización

Comenzando por el análisis de la tasa de falsos positivos (FPR) y verdaderos positivos (TPR). Estas métricas son de alto valor, ya que nos van a permitir saber qué tan bueno es cada uno de nuestros modelos al momento de clasificar si los tweets tienen algún índice de un problema mental.

El caso en que la tasa de clasificación de falsos positivos sea muy alta, genera un problema importante, ya que indica que el modelo no detectó de manera correcta que el tuit a clasificar sí debería ser uno con ningún problema mental, causando un posible desperdicio de recursos. Mientras que un valor bajo de verdaderos positivos indica la calidad del modelo al poder detectar de manera adecuada la mayor cantidad de casos donde hay indicios de algún problema mental. Priorizamos ambas métricas, al igual que el accuracy, para determinar la calidad de nuestros modelos al momento de clasificar.

El Área Bajo la Curva (AUC) nos va a ayudar a determinar la capacidad que tiene el modelo de separar en clases distintas, en nuestro caso si tiene o no un problema mental. Esto lo hace a través de tener valores umbrales, 0.5 siendo negativo (no presenta problema mental) y mayor a 0.5 positivo (presenta problema mental). El ROC (Receiver Operating Characteristic) nos muestra una curva que nos va a permitir entender si el modelo está suficientemente balanceado en la detección de falsos positivos y verdaderos positivos para encontrar la sensibilidad y especificidad de nuestro modelo, donde la sensibilidad se define como la probabilidad de que el modelo genere una predicción positiva de un tweet que haya tenido una clasificación positiva, en nuestro caso que detecte un problema mental y el modelo también lo haya detectado, y la especificidad se refiere a la probabilidad de que el modelo realice una predicción negativa de un tweet que haya tenido una clasificación negativa, donde la detección significa que no hay un problema mental y el modelo no lo detectó como tal.

Optimización de desempeño de modelos

Técnicas de optimización

Las métricas de desempeño AUC, ROC, TPR y FPR son las que tienen mayor peso para poder revisar la calidad del modelo, por lo que haremos uso de diversas optimizaciones clave. Comenzando por cross-validation, lo cual nos permite dividir el dataset que estemos usando en k divisiones, en nuestro caso 5, donde en cada una de estas divisiones colocaremos el modelo elegido para que vea constantemente nuevos datos, lo que ayuda a prevenir el overfit y mide la generalización al permitir mostrar cómo es que el modelo se desempeña con valores de entrenamiento y predicción constantemente cambiantes.

Sin embargo, es importante mencionar que el usar cross-validation con valores por defecto no es suficiente para nuestra problemática, ya que a pesar de que nuestro dataset se encuentra considerablemente balanceado, con alrededor de 800 casos donde hay clasificación de problema mental y 700 casos donde se muestra lo contrario, es necesario descartar cualquier posibilidad de desbalance a lo largo de los folds. Debido a que cross-validation

hace los folds de manera aleatoria, es posible que haya un fold casos donde una clase no se vea representada de manera correcta, realizando así una clasificación errónea; debido a esto optamos por usar el llamado Cross Validation with K-folds estratificados, donde la diferencia clave es que al dividir los datos se busca mantener una distribución homogénea para que no haya ninguna división donde una clase se vea representada con una proporción incorrecta.

Otra manera para optimizar los resultados obtenidos para estas métricas es a través de Grid Search, donde, por medio de un arreglo de valores para los parámetros más significativos (grid) específicos para cada modelo, se eligen aquellos que resulten más beneficiosos para la capacidad de clasificación del modelo. La estructura decidida fue que, con un grid de 5 valores por parámetro, se aplicó la validación cruzada con estratificación y, en caso de encontrar que alguno de los mejores parámetros se encuentre en un extremo del grid, manualmente, movemos el rango de parámetros del grid, de modo que este valor quede en medio de los valores del grid. Esto con el objetivo de hacer una búsqueda exhaustiva por encontrar el valor que mejor represente a los parámetros, garantizando la calidad más alta.

Al combinar estas técnicas de optimización buscamos incrementar la robustez de los modelos, logrando que su desempeño sea el mayor posible en torno a estas métricas.

Implementación de técnicas

De manera inicial, se entrenó a cada modelo con los datos default que ofrece la librería de sci-kit learn, manteniendo siempre un estado random de 22, y realizando un cross validation base de 5 folds, sin estratificación, obteniendo así las métricas y valores de desempeño deseados de cada uno, tales como una matriz de confusión, la gráfica ROC, el cálculo del AUC y los valores de TPR y FPR, con resultados bastante buenos y aceptables, sin embargo, dada nuestra propuesta y compromiso de mejora, decidimos realizar las medidas de optimización descritas previamente, con vista a mejorar el desempeño de los modelos en cada una de las métricas mencionadas.

Con base en los datos recaudados en la primera etapa, procedimos a realizar el entrenamiento de los modelos utilizando un grid search con cross validation estratificada, como se mencionó, con el fin de encontrar la mejor configuración que los modelos pueden obtener. Una vez obtenidos los mejores parámetros, se entrenó un segundo modelo con la finalidad de compararlo con el primer modelo sin esta optimización, y evaluar el mejor con el que proceder, calculando las mismas métricas y gráficas mencionadas anteriormente.

Finalmente, según sea el caso, ya sea de mejora o no, se entrena un modelo final con los parámetros ideales, pero esta vez con el dataset completo de entrenamiento (1,500 entradas), con el fin de aprovechar todos estos valores recibidos en el entrenamiento, y una vez entrenado el modelo final, se evalúa con el dataset de prueba, calculando métricas de desempeño y guardando este último modelo en un archivo para su uso futuro.

Comparación de antes y después de la optimización

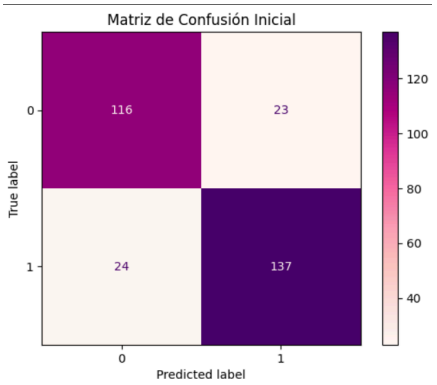
A continuación, se muestra la comparación del desempeño de cada modelo, antes y después de la optimización especificada en la sección previa.

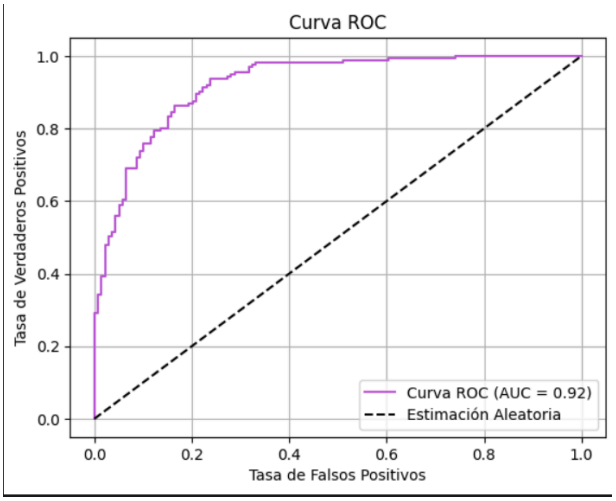
- MLP - Antes de la optimización

```
Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.83      0.83       139
           1       0.86      0.85      0.85       161

    accuracy                           0.84       300
   macro avg       0.84      0.84      0.84       300
weighted avg       0.84      0.84      0.84       300
```






```
AUC: 92.3723
TPR (Tasa de Positivos Verdaderos): 85.0932
FPR (Tasa de Falsos Positivos): 16.5468
```

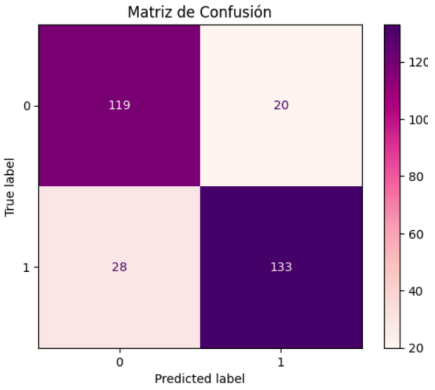
- MLP - Después de la optimización

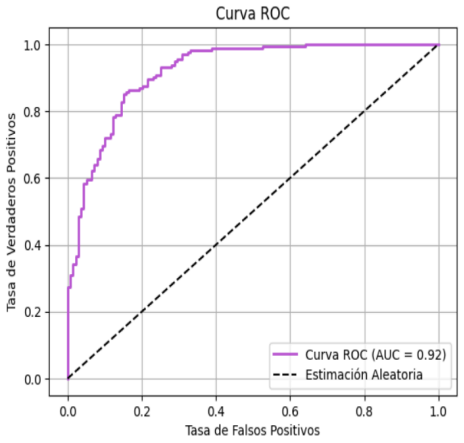
```
MLPClassifier(activation='logistic', hidden_layer_sizes=(300, 200, 100),
              learning_rate='adaptive', max_iter=300, random_state=22)
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.86      0.83       139
           1       0.87      0.83      0.85       161

    accuracy                           0.84       300
   macro avg       0.84      0.84      0.84       300
weighted avg       0.84      0.84      0.84       300
```





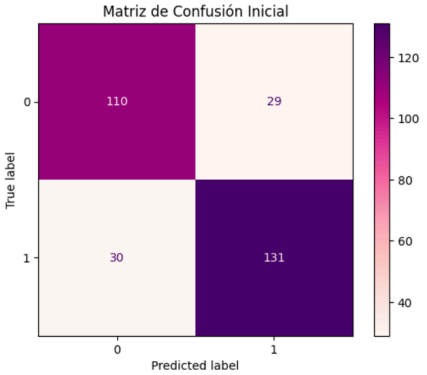

```
AUC: 92.0595
TPR (Tasa de Positivos Verdaderos): 84.3284
FPR (Tasa de Falsos Positivos): 25.0000
```

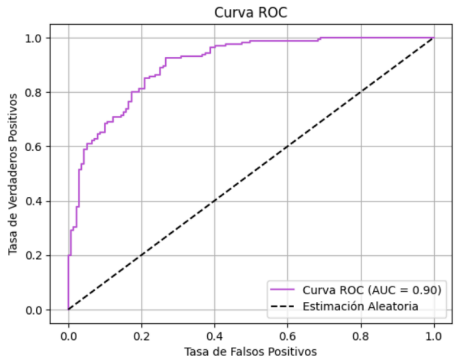
- PAC - Antes de la optimización

```
Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.79      0.79       139
           1       0.82      0.81      0.82       161

    accuracy                           0.80       300
   macro avg       0.80      0.80      0.80       300
weighted avg       0.80      0.80      0.80       300
```






```
AUC: 90.2274
TPR (Tasa de Positivos Verdaderos): 81.3665
FPR (Tasa de Falsos Positivos): 20.8633
```

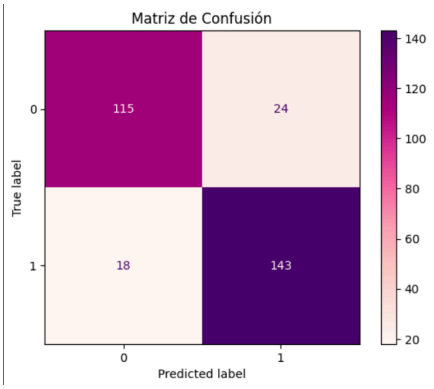
- PAC - Después de la optimización

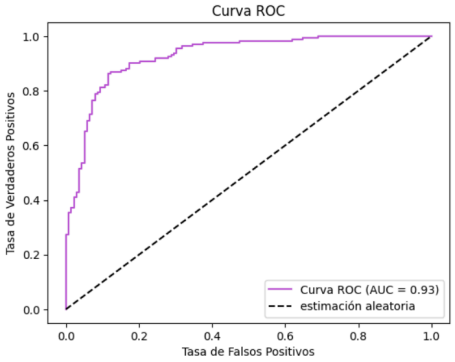
```
PassiveAggressiveClassifier
PassiveAggressiveClassifier(C=0.001, loss='squared_hinge', max_iter=200,
                            random_state=22)
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.83      0.85       139
           1       0.86      0.89      0.87       161

    accuracy                           0.86       300
   macro avg       0.86      0.86      0.86       300
weighted avg       0.86      0.86      0.86       300
```





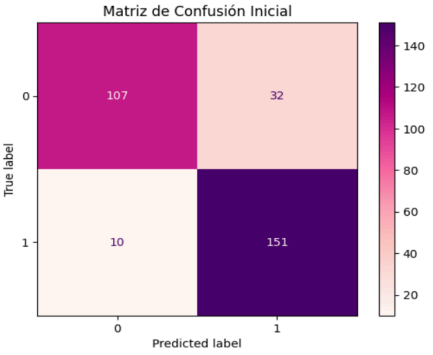

```
AUC: 92.7923
TPR (Tasa de Verdaderos Positivos): 88.8199
FPR (Tasa de Falsos Positivos): 17.2662
```

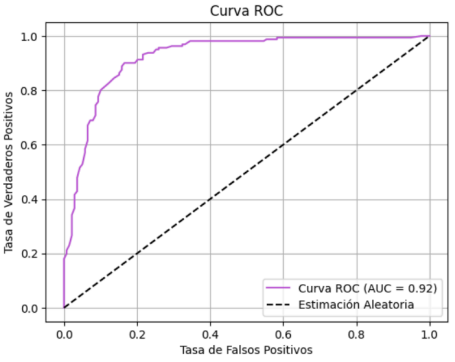
- RF - Antes de la optimización

```
Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.77      0.84       139
           1       0.83      0.94      0.88       161

    accuracy                           0.86       300
   macro avg       0.87      0.85      0.86       300
weighted avg       0.87      0.86      0.86       300
```






```
AUC: 92.3790
TPR (Tasa de Positivos Verdaderos): 93.7888
FPR (Tasa de Falsos Positivos): 23.0216
```

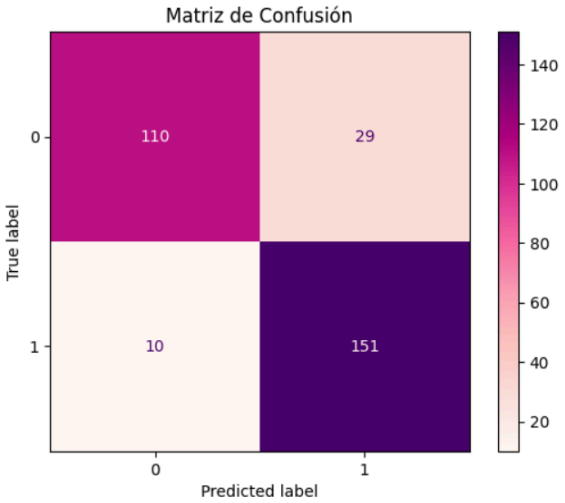
- RF - Después de la optimización

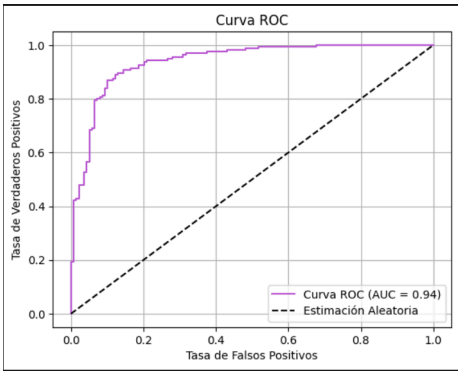
```
RandomForestClassifier
RandomForestClassifier(criterion='entropy', max_features='log2',
                       min_samples_split=6, n_estimators=600, random_state=22)
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.79      0.85       139
           1       0.84      0.94      0.89       161

    accuracy                           0.87       300
   macro avg       0.88      0.86      0.87       300
weighted avg       0.87      0.87      0.87       300
```





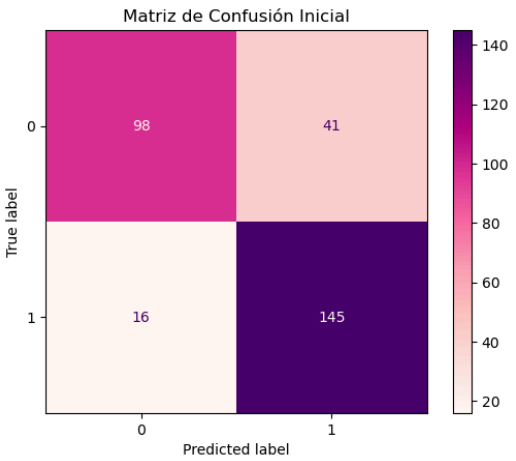

```
AUC: 93.9050
TPR (Tasa de Positivos Verdaderos): 93.7888
FPR (Tasa de Falsos Positivos): 20.8633
```

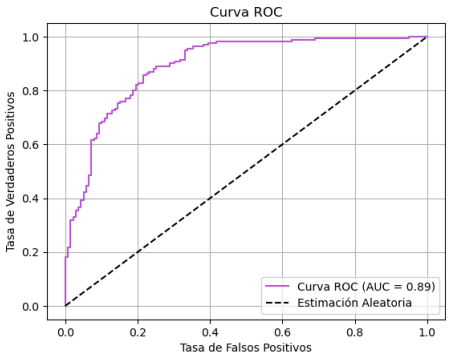
- SVM - Antes de la optimización

```
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.71      0.77       139
           1       0.78      0.90      0.84       161

    accuracy                           0.81       300
   macro avg       0.82      0.80      0.81       300
weighted avg       0.82      0.81      0.81       300
```






```
AUC: 89.2935
TPR (Tasa de Positivos Verdaderos): 90.0621
FPR (Tasa de Falsos Positivos): 29.4964
```

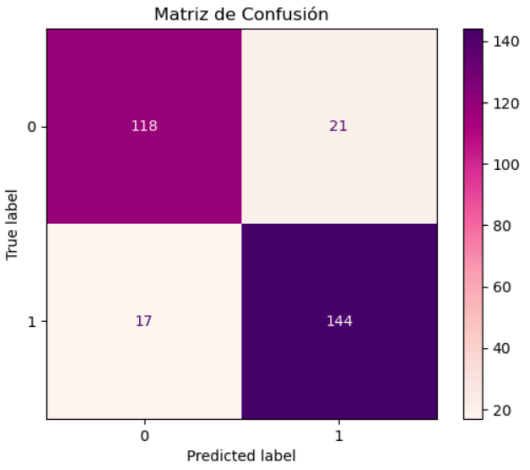
- SVM - Después de la optimización

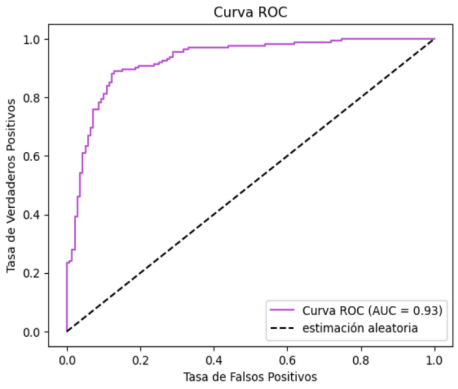
```
SVC
SVC(C=410, gamma=0.001, probability=True, random_state=22)
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.87      0.85      0.86       139
           1       0.87      0.89      0.88       161

    accuracy                           0.87       300
   macro avg       0.87      0.87      0.87       300
weighted avg       0.87      0.87      0.87       300
```





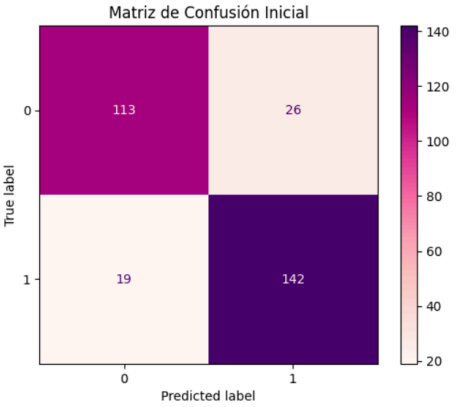

```
AUC: 92.6002
TPR (Tasa de Verdaderos Positivos): 89.4410
FPR (Tasa de Falsos Positivos): 15.1079
```

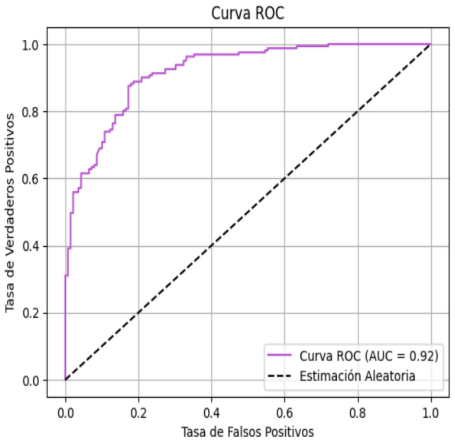
- XGB - Antes de la optimización

```
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.81      0.83       139
           1       0.85      0.88      0.86       161

    accuracy                           0.85       300
   macro avg       0.85      0.85      0.85       300
weighted avg       0.85      0.85      0.85       300
```






```
AUC: 91.8115
TPR (Tasa de Positivos Verdaderos): 88.1988
FPR (Tasa de Falsos Positivos): 18.7050
```

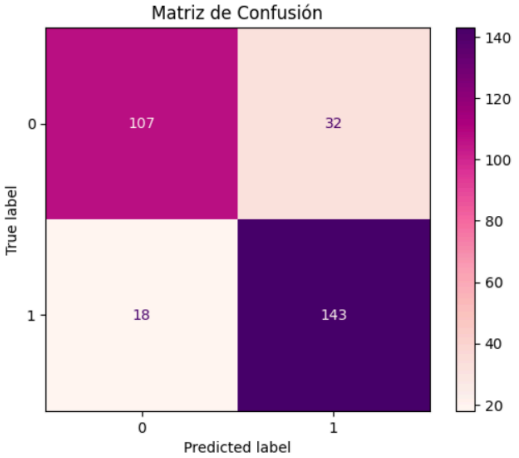
- XGB - Después de la optimización

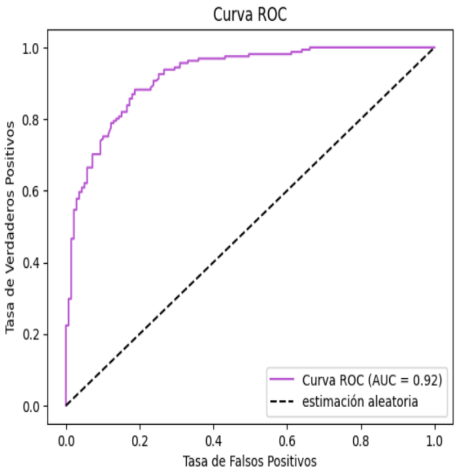
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, feature_weights=None, gamma=None,
              grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.25, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=1, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=600, n_jobs=None,
              num_parallel_tree=None, ...)
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.77      0.81       139
           1       0.82      0.89      0.85       161

    accuracy                           0.83       300
   macro avg       0.84      0.83      0.83       300
weighted avg       0.84      0.83      0.83       300
```






```
AUC: 92.1534
TPR (Tasa de Verdaderos Positivos): 88.8199
FPR (Tasa de Falsos Positivos): 23.0216
```

Como se puede apreciar, los valores default obtuvieron resultados que fueron bastante aceptables, no podemos decir que los resultados fueron del todo erróneos, incluso las métricas obtenidas se pueden apreciar que se observan bastante sólidas, a pesar de ello debemos tomar estos resultados como una opción más, ya que al compararlos con los resultados obtenidos con el Grid Search, es bastante claro que los resultados fueron distintos.

Únicamente existe un caso en el que los valores por default fueron mejores que los valores encontrados por el Grid Search, este caso se encontró con el modelo XGB, estos resultados contrastan totalmente con la mayoría de los modelos que se presentaron, demostrando que los valores default fueron mejores, por lo que únicamente en el caso del modelo XGB se tomarán en cuenta los calores default. Con excepción del modelo XGB, los demás modelos utilizarán los valores de los parámetros obtenidos por el grid search como la base para la etapa final.

Optimización de calidad de código

Habiendo ya especificado la optimización implementada durante el entrenamiento de los modelos tradicionales, en torno a la mejora de resultados obtenidos con la predicción realizada por cada uno de estos, gracias a un stratified K-fold junto con un grid search de los parámetros más significativos, es importante mencionar también que realizó una optimización con respecto a la calidad de código. Dicha optimización consistió en centralizar las funciones utilizadas a lo largo de todo el código fuente, tanto de preprocesamiento como de entrenamiento, debido a su uso común y modularidad, alojándose de esta forma dentro de archivos independientes ubicados en la carpeta src/funciones. Esta organización es beneficiosa gracias a que los diferentes módulos de entrenamiento de nuestros modelos tradicionales iniciales (Random Forest, SVM, MLP, XGBoost y Passive Aggressive), alojados en la carpeta src/modulos, tienen la misma lógica y estructura para barrido de mejores parámetros y entrenamiento, al igual que su evaluación. Dichas funciones las hemos diseñado e implementado con principios de programación estructurada con un enfoque modular, promoviendo así su reutilización constante, al igual que una mayor mantenibilidad y coherencia de código en todos los módulos..

Gracias a este enfoque de optimización estructural, es que nuestro proyecto mejora significativamente en varios aspectos clave:

- Legibilidad: Esta mejora ya que estamos evitando la duplicación innecesaria y reducimos la posibilidad de falta de comprensión a un sólo archivo de código en cada caso (preprocesamiento y entrenamiento), además de una comprensión clara del flujo de datos.
- Mantenimiento: Esto ya que toda modificación o actualización a cualquiera de estas funciones centrales se propaga automáticamente a todos los módulos que la utilizan, sin necesidad de cambios repetitivos que pueden dar lugar a errores.
- Rendimiento del desarrollo y Escalabilidad: Esto es porque, en caso de querer desarrollar un nuevo modelo futuro, su integración al igual que la de sus componentes se realizaría dentro de un marco funcional ya establecido por esta estructura.
- Validación: Gracias a que existe un apartado enfocado a realizar pruebas en el directorio /tests, se vuelve más fácil la validación y prueba de funciones por esta misma razón de concentrar la lógica en estos puntos específicos del problema, garantizando así una mayor robustez y un control de calidad presente a lo largo del ciclo de vida de este proyecto.

Optimización de legibilidad

Otro de los aspectos que consideramos importante para asegurar la calidad del código es la legibilidad. El proyecto está estructurado de forma modular para separar las distintas fases del proceso de detección, el cual está organizado de la siguiente forma:

- data/: Tiene los datos crudos y los preprocesados.
- models/: Contiene los modelos de mejor desempeño en formato pickle.
- src/
    - funciones/: Contiene archivos de las funciones desarrolladas, utilizadas en los módulos.
    - modulos/: Contiene los módulos (archivos Jupyter) de entrenamiento, validación, prueba y evaluación de cada modelo, al igual que el preprocesamiento de datos.
- tests/: Contiene las pruebas unitarias en torno al funcionamiento óptimo de cada función.
- requirements.txt: librerías utilizadas en el proyecto
- install.bat: comandos de instalación de dependencias del proyecto.
- README.md: Contiene información del proyecto e instrucciones de uso y de instalación.

Por otro lado, cada función diseñada en el proyecto cumple con el principio de responsabilidad única y, con el fin de comprender el propósito de ellas, se nombraron acorde a su tarea. Asimismo, se incluyeron comentarios en cada bloque de código para aclarar partes complejas o decisiones técnicas tomadas por el equipo de desarrollo para complementar la legibilidad del código.

Esta estructura garantiza la calidad global del proyecto y la confiabilidad del sistema, ya que de esta manera se simplifica el mantenimiento del código, lo que permite que se hagan

modificaciones seguras sin comprometer el flujo y la aparición de errores. De igual forma, se mejora significativamente la colaboración entre los integrantes del equipo de desarrollo o hasta externos, ya que se puede comprender razonablemente rápido la lógica y composición del código sin revisar detalladamente cada uno de los archivos. Esto, a su vez, permite la escalabilidad del proyecto, dado que se pueden incluir nuevas fuentes de datos o funciones, evitando la reorganización completa del proyecto. Por ende, determinamos que esta optimización de código fortalece la fiabilidad del proyecto, lo que resulta crucial en este escenario de identificación de señales de desórdenes alimenticios en las redes sociales.

Código

Librerías

Librería que permite leer archivos '.csv'.

```
import pandas as pd
```

Librería que permite manipular archivos del sistema operativo.

```
import os
```

Librería que permite usar el intérprete de python de manera más sencilla.

```
import sys
```

Librería que permite guardar nuestro archivo en un pickle para volverlo a correr sin necesidad de reentrenar.

```
import pickle
```

Librería que permite separar en train y test un dataset.

```
from sklearn.model_selection import train_test_split
```

Librería que permite calcular distintas metricas y desplegarlas (classification_report, confusion_matrix, ConfusionMatrixDisplay, roc_curve, roc_auc_score).

```
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    ConfusionMatrixDisplay,
    roc_curve,
    roc_auc_score
)
```

Librería que permite interactuar y hacer uso de la base de datos de caracteres de Unicode.

```
import unicodedata
```

Librería que permite tokenizar y lematizar

```
import spacy
```

Librería que permite el uso de regex.

```
import re
```

Librería que permite usar el traductor de google.

```
from deep_translator import GoogleTranslator
```

Librería que que facilita el análisis y manipulación de texto.

```
from textblob import TextBlob
```

Librería que permite visualizar las métricas del modelo.

```
import matplotlib.pyplot as plt
```

Librería que divide en test y train el dataset, permite usar Grid Search para revisar hiperparametros e incluye Stratified Kfold y Cross validation Score.

```
from sklearn.model_selection import GridSearchCV, StratifiedKFold, cross_val_score
```

Funciones

Entrenamiento

Función: cargar_datos_entrenamiento()
Descripción: Carga el conjunto de datos de entrenamiento desde un archivo CSV, separando todos los atributos (X) de las etiquetas (y).
Entrada: Ninguna.
Salida - X (DataFrame): Atributos del conjunto de entrenamiento.
- y (Series): Etiquetas del conjunto de entrenamiento.

```
def cargar_datos_entrenamiento():
    df = pd.read_csv("../../data/ds_tradicional.csv")
    # Separar características y etiquetas
    X = df.drop(columns=["class"])
    y = df["class"]
    return X, y
```

Función: cargar_datos_prueba()
Descripción: Carga el conjunto de datos de prueba desde un archivo CSV, separando todos los atributos (X) de las etiquetas (y).
Entrada: Ninguna.
Salida: - X (DataFrame): Características del conjunto de prueba.
- y (Series): Etiquetas del conjunto de prueba.

```
def cargar_datos_prueba():
    df = pd.read_csv("../../data/ds_tradicional_TEST.csv")
    # Separar características y etiquetas
    X = df.drop(columns=["class"])
    y = df["class"]
    return X, y
```

Función: imprimir_forma(df)
Descripción: Retorna las dimensiones del DataFrame y las primeras 5 filas para una inspección rápida.
Entrada: - df (DataFrame): Conjunto de datos a inspeccionar.
Salida: - shape (tuple): Dimensiones del DataFrame.
- head (DataFrame): Primeras 5 filas del DataFrame.

```
def imprimir_forma(df):
    return df.shape, df.head(5)
```

Función: division_train_val(X, y)
Descripción: Divide el conjunto de datos en entrenamiento (80%) y validación (20%), manteniendo la proporción de clases.
Entrada: - X (DataFrame): Conjunto de atributos.
- y (Series): Conjunto de las etiquetas clasificatorias.
Salida: - X_train, X_val, y_train, y_val: Subconjuntos para entrenamiento y validación.

```
def division_train_val(X, y):
    X_train, X_val, y_train, y_val = train_test_split(
        X, y,
        test_size=0.2,
```

```
        random_state=22,
        stratify=y
    )
    return X_train, X_val, y_train, y_val
```

Función: reporte_clasificacion(X, y, modelo, lineal=False)

Descripción: Generar predicciones, calcular probabilidades de predicción o funciones de decisión, y crea un reporte de clasificación.

Entrada : - X (DataFrame): Conjunto de atributos.

- y (Series): Conjunto de las etiquetas verdaderas.
- modelo (Model): Modelo entrenado.
- lineal (bool): Indica si el modelo es lineal, en caso de que use una función de decisión en vez de una probabilidad de predicción.

Salida: - y_pred (array): Etiquetas predichas.

- y_res (array): Probabilidades o puntuaciones de decisión.
- reporte (str): Métricas de desempeño del modelo.

```
def reporte_clasificacion(X, y, modelo, lineal=False):
    if not lineal:
        y_pred = modelo.predict(X)
        y_res = modelo.predict_proba(X)[:, 1]
        reporte = classification_report(y, y_pred)
    else:
        y_pred = modelo.predict(X)
        y_res = modelo.decision_function(X)
        reporte = classification_report(y, y_pred)

    return y_pred, y_res, reporte
```

Función: crear_matriz_confusion(y_test, y_pred)

Descripción: Calcula y prepara los valores de la matriz de confusión para su visualización.

Entrada - y_test (array): Etiquetas verdaderas.

- y_pred (array): Etiquetas predichas.

Salida - cm (array): Matriz de confusión en forma de matriz.

- disp (objeto): Objeto para mostrar la matriz de confusión.

```
def crear_matriz_confusion(y_test, y_pred):
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    return cm, disp
```

Función: calcular_roc_auc(y, y_res)

Descripción: Calcula la curva ROC y el valor AUC del modelo.

Entrada: - y (array): Etiquetas verdaderas.

- y_res (array): Probabilidades o puntuaciones del modelo.

Salida: - fpr (array): Tasas de falsos positivos.

- tpr (array): Tasas de verdaderos positivos.
- thresholds (array): Umbrales de decisión.
- auc_score (float): Área bajo la curva ROC.

```
def calcular_roc_auc(y, y_res):
    fpr, tpr, thresholds = roc_curve(y, y_res)
    auc_score = roc_auc_score(y, y_res)
```

```
    return fpr, tpr, thresholds, auc_score
```

Función: metricas_tpr_fpr(cm)
Descripción: Calcula las tasas de verdaderos positivos (TPR) y falsos positivos (FPR) a partir de una matriz de confusión.
Entrada - cm (array): Matriz de confusión en forma de matriz.
Salida - TPR (float): Tasa de verdaderos positivos.
FPR (float): Tasa de falsos positivos.

```
def metricas_tpr_fpr(cm):
    TN, FP, FN, TP = cm.ravel()
    FPR = 0.0 if FP + TN == 0.0 else FP / (FP + TN)
    TPR = 0.0 if TP + FN == 0.0 else TP / (TP + FN)
    return TPR, FPR
```

Función: hacer_pepinillo(modelo, nombre, test=False)
Descripción: Guarda un modelo entrenado en formato pickle. La ubicación depende del entorno (test o producción), siendo por default la ruta fija de modelos.
Entrada: - modelo (objeto): Modelo entrenado a guardar.
nombre (str): Nombre del archivo destino.
test (bool): Indica si se trata de un entorno de prueba (True) o producción (False).
Salida: Ninguna.

```
def hacer_pepinillo(modelo, nombre, test=False):
    if test:
        os.makedirs(os.path.dirname(nombre), exist_ok=True)
        with open(nombre, "wb") as f:
            pickle.dump(modelo, f)
    else:
        with open(f"../../models/{nombre}", "wb") as f:
            pickle.dump(modelo, f)
```

Preprocesamiento

Función: limpiar_texto
Descripción: Limpia y normaliza una cadena de texto, eliminando acentos y convirtiendo todo en minúsculas.
Entrada: texto (str): Cadena de texto que puede contener caracteres especiales o estar en codificación no UTF-8.
Salida: texto limpio (str): Texto en minúsculas, sin acentos ni caracteres especiales.

```
def limpiar_texto(texto):
    if pd.isnull(texto):
        return ""
    try:
        texto = texto.encode("latin1").decode("utf-8")
    except:
        pass
    texto = unicodedata.normalize('NFKD', texto).encode('ascii',
'ignore').decode('utf-8')
    return texto.lower()
```

Función: expandir_abreviaturas

Descripción: Reemplaza abreviaturas presentes en el texto por sus equivalentes completos contenidos en la variable ABREVIATURAS.
Entrada: texto (str): Cadena de texto con posibles abreviaturas.
Salida: texto expandido (str): Texto con las abreviaturas reemplazadas por su forma completa.

```
def expandir_abreviaturas(texto):
    palabras = texto.split()
    palabras_expandidas = [ABREVIATURAS.get(p, p) for p in palabras]
    return ' '.join(palabras_expandidas)
```

Función: procesar_hashtags
Descripción: Extrae los hashtags del texto y devuelve el texto separado de ellos.
Entrada:texto (str): Cadena de texto que posiblemente contiene hashtags (palabras precedidas de '#').
Salida:
texto_sin_hashtags (str): Texto sin los hashtags.
hashtags (list): Lista de hashtags encontrados sin el símbolo '#'.

```
def procesar_hashtags(texto):
    hashtags = re.findall(r"#(\w+)", texto)
    texto_sin_hashtags = re.sub(r"#\w+", "", texto)
    return texto_sin_hashtags.strip(), hashtags
```

Función: procesar_hashtags
Descripción: Elimina URLs, menciones, símbolos especiales y caracteres no alfanuméricos del texto.
Entrada:texto (str): Cadena de texto posiblemente con menciones, enlaces y símbolos innecesarios.
Salida:texto limpio (str): Texto limpio y legible, sin símbolos ni caracteres no deseados.

```
def limpieza_final(texto):
    texto = re.sub(r"http\S+", "", texto)
    texto = re.sub(r"@\w+", "", texto)
    texto = re.sub(r"&", "y", texto)
    texto = re.sub(r"[^a-zA-ZáéíóúüñÁÉÍÓÚÜÑ0-9\s]", "", texto)
    texto = re.sub(r"\s+", " ", texto).strip()
    return texto
```

Función: tokenizar_y_lematizar
Descripción: Tokeniza y lematiza el texto en español usando spaCy, eliminando stopwords, puntuación y espacios.
Entrada: texto (str): Cadena de texto en español coherente, con puntuación y stopwords, seguramente con espacios innecesarios.
Salida: texto procesado (str): Texto con lemas en minúsculas, sin palabras vacías ni signos de puntuación.

```
def tokenizar_y_lematizar(texto):
    doc = nlp(texto)
    # Filtrar tokens que no son stopwords, puntuación o espacios
    tokens = [token.lemma_.lower() for token in doc
              if not token.is_stop and not token.is_punct and not token.is_space]
    return " ".join(tokens)
```

Función: calcular_metricas_estilísticas
Descripción: Calcula métricas básicas de estilo sobre el texto, como longitud y número de palabras.
Entrada: texto (str): Cadena de texto procesado.
target_lang (str): Idioma de destino para la traducción (por defecto "en" para inglés).
Salida: dict: Diccionario con dos entradas; longitud del texto en caracteres y número de palabras.

```
def calcular_metricas_estilisticas(texto):
    palabras = texto.split()
```

```
    caracteres = len(texto)

    return {
        'longitud_texto': caracteres,
        'num_palabras': len(palabras)
    }
```

Función: analizar_palabras_clave
Descripción: Cuenta cuántas palabras clave lematizadas de cada categoría aparecen en el texto.
Entrada: texto (str): Texto lematizado, incluyendo palabras de los hashtags.
Salida: dict: Diccionario con la frecuencia de aparición por categoría de palabras clave.

```
def analizar_palabras_clave(texto):
    frecuencias = {}
    for categoria, palabras in PALABRAS_CLAVE_LEMATIZADAS.items():
        frecuencias[categoria] = sum(1 for palabra in palabras if palabra in texto)
    return frecuencias
```

Función: traducir_si_necesario

Descripción: Traduce el texto al idioma objetivo, por defecto inglés, usando un traductor automático.
Entrada: texto (str): Texto original.
target_lang (str): Idioma de destino para la traducción (por defecto "en" para inglés).
Salida: texto traducido (str) - Texto traducido al idioma especificado o el texto original si ocurre un error.

```
def traducir_si_necesario(texto, target_lang='en'):
    try:
        return GoogleTranslator(source='auto', target=target_lang).translate(texto)
    except Exception as e:
        print(f"Error en traducción: {e}")
        return texto
```

Función: analizar_sentimiento
Descripción: Analiza el sentimiento del texto traducido utilizando TextBlob.
Entrada: texto (str): Texto en cualquier idioma (será traducido automáticamente).
Salida: dict: Diccionario con métricas de sentimiento (polaridad y subjetividad) para la cadena de texto dada.

```
def analizar_sentimiento(texto):

    try:
        # Traducir solo para análisis de sentimiento
        texto_traducido = traducir_si_necesario(texto)
        blob = TextBlob(texto_traducido)
        return {
            'polaridad': blob.sentiment.polarity,
            'subjetividad': blob.sentiment.subjectivity
        }
    except Exception as e:
        print(f"Error en análisis de sentimiento: {e}")
        return {
            'polaridad': 0,
            'subjetividad': 0
        }
```

Función: obtener_hashtags_frecuentes_individuales
Descripción: Filtra hashtags válidos presentes en una fila de entrada y los une en una cadena de texto.

Entrada: hashtags_fila (list): Lista de hashtags extraídos de un tweet.hashtags_validos (set o list): Conjunto de hashtags considerados válidos.
Salida: str - Cadena con los hashtags válidos separados por espacios.

```
def obtener_hashtags_frecuentes_individuales(hashtags_fila, hashtags_validos):
    return " ".join([tag for tag in hashtags_fila if tag in hashtags_validos])
```

Función: extraer_caracteristicas
Descripción: Extrae múltiples características de un tweet aplicando las funciones descritas previamente: limpieza, expansión de abreviaturas, lematización, sentimientos, métricas de estilo, hashtags y palabras clave.
Entrada: tweet (str): Texto original del tweet.
Salida: dict: Diccionario con características procesadas del tweet.

```
def extraer_caracteristicas(tweet):
    """
    # Separación de hashtags
    texto_original, hashtags = procesar_hashtags(tweet)

    # Preprocesamiento inicial
    texto_limpio = limpiar_texto(texto_original)
    texto_expandido = expandir_abreviaturas(texto_limpio)
    texto_limpio_final = limpieza_final(texto_expandido)

    # Análisis de sentimiento sobre texto limpio expandido
    sentimiento = analizar_sentimiento(texto_limpio_final)

    # Lematización
    texto_lematizado = tokenizar_y_lematizar(texto_limpio_final)

    # Métricas estilísticas
    estilisticas = calcular_metricas_estilisticas(texto_lematizado)

    # Preparar texto completo con hashtags sin #
    texto_completo = texto_lematizado + " " + " ".join(hashtags)

    # Análisis de palabras clave sobre texto completo
    palabras_clave = analizar_palabras_clave(texto_completo)

    return {
        "tweet_text": texto_lematizado,
        "hashtags": hashtags,
        "texto_completo": texto_completo,
        "texto_bert": texto_limpio_final + " " + " ".join(hashtags),
        "longitud_texto": estilisticas["longitud_texto"],
        "num_palabras": estilisticas["num_palabras"],
        **palabras_clave,
        **sentimiento
    }
```

Código estándar compartido por cada módulo

En esta sección incluiremos todas las similitudes que hay en todos los módulos de entrenamiento de los modelos, ya que usan y reciclan funcionamiento.

Importa las funciones necesarias para poder realizar los modelos de manera estandarizada:

```
sys.path.append(os.path.abspath("../funciones"))
from funcionesEntrenamiento import (
    cargar_datos_entrenamiento,
    cargar_datos_prueba,
    imprimir_forma,
    division_train_val,
    reporte_clasificacion,
    crear_matriz_confusion,
    calcular_roc_auc,
    metricas_tpr_fpr,
    hacer_pepinillo
)
```

Carga los datasets e imprime la forma de las mismas usando las funciones '*cargar_datos_entrenamiento()'* e '*imprimir_forma(X)'* del archivo funciones.

```
X, y = cargar_datos_entrenamiento()
shape, head = imprimir_forma(X)
print("Shape: ", shape)
head
```

Divide al dataset con distintas características y etiquetas, nuevamente usando funciones del archivo *'funciones'* '*division_train_val(X,y)'* , *'imprimir_forma (X_train)*', *'imprimir_forma'* y *(X_val)*'.

```
X_train, X_val, y_train, y_val = division_train_val(X, y)
train_shape, _ = imprimir_forma (X_train)
val_shape, _ = imprimir_forma(X_val)

print(f'Forma de conjunto de entrenamiento: {train_shape}')
print(f'Forma de conjunto de validacion: {val_shape}')
```

Procedemos a realizar el entrenamiento de nuestro modelo

```
svm_inicial.fit(X_train, y_train)
```

Creamos nuestro reporte del modelo inicial usando la función 'reporte_clasificacion' del archivo 'funciones', usando los valores de validación inicial X, validación inicial Y al igual que el modelo inicial. Los valores los almacenaremos en las variables *'y_pred_inicial'*, *'y_proba_inicial'* y *'reporte_inicial'*

```
y_pred_inicial, y_proba_inicial, reporte_inicial = reporte_clasificacion(X_val, y_val, svm_inicial)

print("Classification Report:")
print(reporte_inicial)
```

Creamos nuestra matriz de confusión con los valores de validación de la columna a predecir y los valores que obtuvimos en la predicción usando el modelo inicial. Los desplegamos y almacenamos en '*cm_inicial'* y '*disp_inicial'*

```
cm_inicial, disp_inicial = crear_matriz_confusion(y_val, y_pred_inicial)
disp_inicial.plot(cmap='RdPu')
plt.title("Matriz de Confusión Inicial")
plt.show()
```

Generamos la curva ROC y los cálculos de la puntuación de AUC usando la función *'crear_matriz_confusion'* del archivo *'funciones'* para el modelo inicial, nuevamente usamos los valores de predicción para la columna objetivo al igual que los resultados de probabilidad en el modelo inicial. Desplegamos el gráfico y almacenamos los valores dentro de '*fpr_inicial'*, '*tpr_inicial'*, '*thresholds_inicial'* y '*auc_score_inicial'*.

```
fpr_inicial, tpr_inicial, thresholds_inicial, auc_score_inicial = calcular_roc_auc(y_val, y_proba_inicial)

plt.figure()
plt.plot(fpr_inicial, tpr_inicial, color='mediumorchid', label=f"Curva ROC (AUC = {auc_score_inicial:.2f})")
plt.plot([0, 1], [0, 1], 'k--', label="Estimación Aleatoria")
plt.xlabel("Tasa de Falsos Positivos")
plt.ylabel("Tasa de Verdaderos Positivos")
plt.title("Curva ROC")
plt.legend(loc="lower right")
plt.grid()
plt.show()
```

Tomamos la matriz de confusión inicial para poder calcular las metricas usando la función *'metricas_tpr_fpr'* del archivo *'funciones'* y a partir de él almacenar los valores de '*TPR_inicial' y 'FPR_inicial'* para el despliegue de las mismas.

```
TPR_inicial, FPR_inicial = metricas_tpr_fpr(cm_inicial)

# Mostrar métricas
print(f"AUC: {(auc_score_inicial * 100):.4f}")
print(f"TPR (Tasa de Positivos Verdaderos): {(TPR_inicial * 100):.4f}")
print(f"FPR (Tasa de Falsos Positivos): {(FPR_inicial * 100):.4f}")
```

Entrenamos usando nuestro grid search único con los valores de validación en predicción y columna objetivo. Posteriormente, imprimimos los mejores parametros que encontramos en nuestra busqueda, para realizar el ajuste manual para nuestros hiperparametros y repetir el proceso dos veces más.

```
grid_search.fit(X_val, y_val)
```

```
grid_search.best_params_
```

Creamos nuestro reporte del modelo entrenado con grid search usando la función 'reporte_clasificacion' del archivo 'funciones', usando los valores de validación X, validación Y al igual que el modelo entrenado con grid search. Los valores los almacenaremos en las variables *'y_pred'*, *'y_proba'* y *'reporte'*

```
y_pred, y_proba, reporte = reporte_clasificacion(X_val, y_val, best_SVM)

print("Classification Report:")
print(reporte)
```

Creamos nuestra matriz de confusión con los valores de validación de la columna a predecir y los valores que obtuvimos en la predicción usando el modelo entrenado con los mejores parámetros de grid search. Los desplegamos y almacenamos en '*cm'* y '*disp'*

```
cm, disp = crear_matriz_confusion(y_val, y_pred)
disp.plot(cmap="RdPu")
plt.title("Matriz de Confusión")
plt.show()
```

Generamos la curva ROC y los cálculos de la puntuación de AUC usando la función *'crear_matriz_confusion'* del archivo *'funciones'* para el modelo entrenado con grid search, nuevamente usamos los valores de predicción para la columna objetivo al igual que los resultados de probabilidad en el modelo. Desplegamos el gráfico y almacenamos los valores dentro de *'fpr'*, *'tpr'*, '*thresholds'* y '*auc_score'*.

```
# Predicciones de scores (para curva ROC)
fpr, tpr, thresholds, auc_score = calcular_roc_auc(y_val, y_proba)

# Gráfica de ROC
plt.plot(fpr, tpr, color='mediumorchid', label=f"Curva ROC (AUC = {auc_score:.2f})")
plt.plot([0, 1], [0, 1], "k--", label='estimación aleatoria')
plt.xlabel("Tasa de Falsos Positivos")
plt.ylabel("Tasa de Verdaderos Positivos")
plt.title("Curva ROC")
plt.legend(loc="lower right")
plt.show()
```

Tomamos la matriz de confusión del modelo entrenado con grid search para poder calcular las métricas usando la función *'metricas_tpr_fpr'* del archivo *'funciones'* y a partir de él almacenar los valores de '*TPR' y 'FPR'* para el despliegue de las mismas.

```
# Cálculo de métricas de desempeño de la matriz de confusión
TPR, FPR = metricas_tpr_fpr(cm)

# Mostrar métricas
print(f"AUC: {(auc_score * 100):.4f}")
print(f"TPR (Tasa de Verdaderos Positivos): {(TPR * 100):.4f}")
print(f"FPR (Tasa de Falsos Positivos): {(FPR * 100):.4f}")
```

Carga los datasets esta vez usando el de entrenamiento e imprime la forma de las mismas usando las funciones '*cargar_datos_prueba()'* e '*imprimir_forma(X)'* del archivo funciones.

```
X_test, y_test = cargar_datos_prueba()

shape_test, head_test = imprimir_forma(X_test)
print("Shape test: ",shape_test)
head_test
```

Creamos nuestro reporte del modelo entrenado con los 1500 datos usando la función 'reporte_clasificacion' del archivo 'funciones', usando los valores de validación X, validación Y al igual que el modelo. Los valores los almacenaremos en las variables *'y_pred_test'*, *'y_proba_test'* y *'reporte_test'*

```
y_pred_test, y_proba_test, reporte_test = reporte_clasificacion(X_test, y_test, modelSVM)

print("Classification Report:")
print(reporte_test)
```

Creamos nuestra matriz de confusión con los valores de validación de la columna a predecir y los valores que obtuvimos en la predicción usando el modelo entrenado 1500 datos. Los desplegamos y almacenamos en '*cm_test'* y '*disp_test'*

```
cm_test, disp_test = crear_matriz_confusion(y_test, y_pred_test)
disp_test.plot(cmap='RdPu')
plt.title("Matriz de Confusión Final")
plt.show()
```

Generamos la curva ROC y los cálculos de la puntuación de AUC usando la función *'crear_matriz_confusion'* del archivo *'funciones'* para el modelo inicial, nuevamente usamos los valores de predicción para la columna objetivo al igual que los resultados de probabilidad en el modelo inicial. Desplegamos el gráfico y almacenamos los valores dentro de '*fpr_test'*, '*tpr_test'*, '*thresholds_test'* y '*auc_score_test'*.

```
# Predicciones de scores (para curva ROC)
fpr_test, tpr_test, thresholds_test, auc_score_test = calcular_roc_auc(y_test, y_proba_test)

# Gráfica de ROC
plt.plot(fpr_test, tpr_test, color='mediumorchid', label=f"Curva ROC (AUC = {auc_score_test:.2f})")
plt.plot([0, 1], [0, 1], "k--", label='estimación aleatoria')
plt.xlabel("Tasa de Falsos Positivos")
plt.ylabel("Tasa de Verdaderos Positivos")
plt.title("Curva ROC")
plt.legend(loc="lower right")
plt.show()
```

Tomamos la matriz de confusión inicial para poder calcular las métricas usando la función *'metricas_tpr_fpr'* del archivo *'funciones'* y a partir de él almacenar los valores de '*TPR_test' y 'FPR_test'* para el despliegue de las mismas.

```
# Cálculo de métricas de desempeño de la matriz de confusión
TPR_Test, FPR_Test = metricas_tpr_fpr(cm_test)

# Mostrar métricas
print(f"AUC: {(auc_score_test * 100):.4f}")
print(f"TPR (Tasa de Verdaderos Positivos): {(TPR_Test * 100):.4f}")
print(f"FPR (Tasa de Falsos Positivos): {(FPR_Test * 100):.4f}")
```

Por último, guardamos nuestro modelo final en un archivo pickle para poderlo correr sin necesidad de reentrenar.

```
hacer_pepinillo(modelSVM, "modelSVM.pkl")
```

Modelo Random Forest

Iniciamos nuestra descripción del módulo de entrenamiento con el modelo de random forest. En primera instancia, creamos un objeto de tipo clasificador de Random Forest, especificando un estado random de 22, y con este mismo modelo realizamos una validación cruzada con el conjunto de datos de entrenamiento, con la finalidad de evaluar correctamente su desempeño como modelo y su capacidad de generalización. Especificamos que queremos 5 folds con una evaluación con base en ROC y AUC, enfocándonos en la predicción correcta de positivos.

```python
rf_inicial = RandomForestClassifier(random_state=22)

scores = cross_val_score(rf_inicial, X_train, y_train, cv=5, scoring='roc_auc')

print("Resultados por fold:", scores)
print("Precisión promedio:", scores.mean())
```

```
Resultados por fold: [0.90935121 0.92248062 0.90121517 0.90178571 0.95204381]
Precisión promedio: 0.9173753041156294
```

Habiendo evaluado las métricas arrojadas por este modelo inicial, por medio del reporte indicado en la sección de estandarización de código.

AUC: 92.3790
TPR (Tasa de Positivos Verdaderos): 93.7888
FPR (Tasa de Falsos Positivos): 23.0216

Procedemos a realizar un grid search integrado con nuestro cross validation, al igual que una división estratificada para cada uno de los 5 folds, con la finalidad de mantener una distribución homogénea a lo largo de cada fold. En cuanto al grid search mencionado, se realizó un diccionario con las métricas más significativas para este modelo de random forest, donde cada llave es el nombre de la métrica, y su contenido es una lista de posibles valores de esta, y que queremos que sean considerados durante el entrenamiento para conocer con el que mejor entrene el modelo. En este caso, la primer métrica o parámetro es el número de árboles, ya que al ser los protagonistas de este modelo y por la cantidad de atributos con que estamos entrenando, este número es de gran importancia. La segunda métrica es la profundidad máxima de los árboles, limitando así su proceso de decisión. La tercer métrica es el número mínimo de números requeridos para dividir un nodo de algún árbol, importante ya que define qué tanta ramificación existirá y, por lo tanto, la profundidad del árbol. Como cuarto parámetro, está el número máximo de atributos al buscar la mejor división, con funciones aplicadas al número de atributos. Como quinto parámetro está el número mínimo de números requeridos para que sea considerado un nodo o una hoja, y este parámetro es importante, ya que es capaz de suavizar el modelo. Finalmente, el sexto parámetro es la función que mide la calidad de una división, ya sea por medio de entropía, pérdida logarítmica o impureza de Gini.

```python
rf = RandomForestClassifier(random_state=22)

param_grid = {
    'n_estimators': [500, 550, 600, 650, 700],
    'max_depth': [None, 2, 4, 6, 8, 10],
    'min_samples_split': [4, 5, 6, 7, 8],
    'max_features': ['sqrt', 'log2', None],
    'min_samples_leaf': [1, 2, 3, 4, 5],
    'criterion': ['gini', 'entropy', 'log_loss']
}

cv = StratifiedKFold(
    n_splits=5,
    shuffle=True,
    random_state=22
)

grid_search = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    cv=cv,
    scoring='roc_auc',
    n_jobs=-1,
    verbose=1,
    error_score='raise'
)
```

Una vez realizado este barrido y validación cruzada, se entrena un modelo final con los mejores parámetros captados a partir del gridsearch con el stratified K-fold, con la totalidad de los datos de entrenamiento, es decir, tanto del conjunto de entrenamiento como de validación, explotando de esta manera la cantidad de datos de entrenamiento y refinando el modelo una vez más para realizar predicciones con los datos de prueba.

```python
modelRF = RandomForestClassifier(
    n_estimators = 600,
    criterion = 'entropy',
    max_depth = None,
    max_features = 'log2',
    min_samples_leaf = 1,
    min_samples_split = 6,
    random_state = 22
)

modelRF.fit(X, y)
```

Modelo Passive Aggressive Classifier

Otro de los modelos seleccionados para la detección de anorexia fue el algoritmo Passive Aggressive Classifier (PAC), que se destaca por ser un modelo eficiente en cuanto a tareas de clasificación binaria se refiere y desempeño con conjuntos de datos grandes. Otra ventaja del PAC es que solamente ajusta sus pesos cuando comete errores en su predicción, por lo que no es necesario recorrer el conjunto de datos varias veces.

Al igual que los otros modelos, se procesó un modelo base con los hiperparámetros por defecto de la clase **PassiveAggressiveClassifier,** esto con el fin de establecer una base para comparar las optimizaciones que se aplicaron a cada uno de los modelos y así tener un análisis más completo del desempeño del modelo en este contexto sin intervención manual. El único parámetro modificado fue el *random_state=22*, esto con el fin de mantener un estándar de comparación contra los otros modelos.

```
pac_inicial = PassiveAggressiveClassifier(random_state=22)

scores = cross_val_score(pac_inicial, X_train, y_train, cv=5,scoring='roc_auc')

print("Resultados por fold:", scores)
print("Precisión promedio:", scores.mean())
```

Una diferencia importante entre este y los otros modelos es que en la validación del modelo se utilizó la función .decision_function(), ya que PAC no proporciona probabilidades, sino scores de decisión, los cuales permiten generar métricas basadas en umbrales, y con ellos se puede graficar la curva ROC.

```
y_pred_inicial, y_scores_inicial, reporte_inicial = reporte_clasificacion(X_val, y_val, pac_inicial, True)

print("Classification Report:")
print(reporte_inicial)
```

```
def reporte_clasificacion(X, y, modelo, lineal=False):
    if not lineal:
        y_pred = modelo.predict(X)
        y_res = modelo.predict_proba(X)[:, 1]
        reporte = classification_report(y, y_pred)
    else:
        y_pred = modelo.predict(X)
        y_res = modelo.decision_function(X)
        reporte = classification_report(y, y_pred)
```

Finalmente, para optimizar el desempeño del modelo, se implementó una búsqueda exhaustiva de los mejores hiperparámetros con GridSearchCV y validación cruzada para el modelo y el contexto en el que se está desempeñando. Los hiperparámetros más significativos para la búsqueda fueron:

- **C**: Es el parámetro de regularización, el cual se encarga de controlar que tanto se penaliza cuando el modelo comete un error.
- **loss:** Es el parámetro que define la función de pérdida que en este caso impacta directamente en la penalización de errores.
- **average**: Es el parámetro que define si los coeficientes se deben promediar durante el entrenamiento, lo cual ayuda a mejorar la generalización de los datos.

```python
# Modelo base para encontrar mejores parámetros
pac = PassiveAggressiveClassifier(random_state=22)

# Parámetros a buscar con GridSearchSV
param_grid = {
    'C': [0.001, 0.01, 0.1, 10],
    'max_iter': [150, 200, 250, 300, 350],
    'loss': ['hinge', 'squared_hinge']
}

# Cross-validation estratificada
cv = StratifiedKFold(
    n_splits=5,
    shuffle=True,
    random_state=22
)

# GridSearch
grid = GridSearchCV(
    estimator=pac,
    param_grid=param_grid,
    cv=cv,
    scoring='roc_auc',
    n_jobs=-1,
    verbose=1,
    error_score='raise'
)
```

Modelo XGB

El tercer modelo seleccionado para la clasificación fue el algoritmo de XGBoost (Extreme Gradient Boosting) pues se destaca por su alta precisión, su manejo efectivo de datos estructurados y velocidad de entrenamiento. De igual forma, este modelo fue procesado inicialmente con los datos por defecto que ofrece la clase **XGBoostClassifier;** sin embargo, solo se modificó el *random_state=22* para mantener el estándar de comparación.

```python
xgb_inicial = XGBClassifier(random_state=22)

scores = cross_val_score(xgb_inicial, X_train, y_train, cv=5,scoring='roc_auc')

print("Resultados por fold:", scores)
print("Precisión promedio:", scores.mean())
```

Como para todos los modelos, se implementó la búsqueda de los mejores hiperparámetros con GridSearch y validación cruzada para probar todas las combinaciones posibles. Los hiperparámetros más significativos para la búsqueda fueron los siguientes:

- **n_estimators**: Este parámetro representa el número total de árboles que se construirán, el cual indica el aprendizaje de patrones descubiertos en el conjunto de datos.
- **max_depth**: Es el parámetro que define la profundidad, es decir, la complejidad de cada árbol generado.
- **learning_rate**: Este parámetro se encarga de regular la contribución de cada árbol al modelo final.
- **subsample**: Es el parámetro que controla la cantidad de observaciones que se usan para entrenar cada árbol.

```python
# Modelo base para encontrar mejores parámetros
xgb = XGBClassifier(random_state=22)

# Parámetros a buscar con GridSearchSV
param_grid = {
    'n_estimators': [300, 400, 500, 600, 700],
    'max_depth': [1, 2, 3, 4, 5],
    'learning_rate': [0.01, 0.05, 0.1, 0.15, 0.20, 0.25, 0.30],
    'subsample': [0.8, 1],
    'eval_metric' : ['logloss', 'auc', 'error', 'aucpr', 'poisson-nloglik', 'gamma-nloglik']
}

# Cross-validation estratificada
cv = StratifiedKFold(
    n_splits=5,
    shuffle=True,
    random_state=22
)

# GridSearch
grid = GridSearchCV(
    estimator=xgb,
    param_grid=param_grid,
    cv=cv,
    scoring='roc_auc',
    n_jobs=-1,
    verbose=1,
    error_score='raise'
)
```

Cabe mencionar que en este caso el mejor modelo resultó ser el inicial con los parámetros por defecto, por lo que, a diferencia de los demás modelos, el modelo XGBoost inicial se guardó en un pickle para facilitar las pruebas posteriores y tener un flujo uniforme.

```
# Modelo entrenado con todos los datos de ds_tradicional
modelXGB = XGBClassifier(random_state=22)
modelXGB.fit(X, y)
```

```
# Guardado del modelo final en un archivo pickle
hacer_pepinillo(modelXGB, "modelXGB.pkl")
```

Modelo SVM

Entrenamos con los valores default del modelo SVM, pero usando un cross validation, esto a través de la función SVC con los parámetros de random_state=22 y probability = True para desplegarlos posteriormente. Igualmente usamos el cross_val_score con nuestro modelo creado, las variables de entrenamiento de X y de y, basándonos en la mejor ROC AUC para su puntaje.

```
svm_inicial = SVC(random_state=22, probability=True)
scores = cross_val_score(svm_inicial, X_train, y_train, cv=5, scoring='roc_auc')

print("Resultados por fold:", scores)
print("Precisión promedio:", scores.mean())
```

Para el modelo SVM, después de investigar, encontramos que los hiperparámetros más significativos en el entrenamiento son C, el cual controla la tolerancia a los errores; gamma, el cual determina implícitamente la distribución de los datos mapeados al nuevo espacio de características; y Kernel, que es el núcleo con el cual el modelo realizará el entrenamiento, tomando en cuenta cómo esté el dataset; se ajusta a un valor. Por lo que creamos un grid que analice varios datos posibles para c, gamma y kernel; posteriormente creamos un stratifiedkfold para la distribución del dataset y que se mantenga balanceado en cada fold. Por último, lo usamos en nuestro grid search para que busque en nuestro modelo la mejor combinación de valores; realizamos este proceso 3 veces en total, manualmente ajustando los valores de C y gamma hasta que encuentre uno que nos satisfaga.

```
model = SVC(random_state=22)

hyperparam_grid = {
    'C': [300, 350, 400, 450, 500],
    'gamma': [0.0005, 0.001, 0.012, 0.014, 0.016],
    'kernel': ['rbf', 'linear', 'poly', 'sigmoid']
}

cv = StratifiedKFold(
    n_splits=5,
    shuffle=True,
    random_state=22
)

grid_search = GridSearchCV(
    estimator=model,
    param_grid=hyperparam_grid,
    refit = True,
    verbose = 3,
    scoring = 'roc_auc',
    n_jobs=-1,
    cv = cv,
    error_score='raise'
)
```

Una vez realizado este barrido y validación cruzada, se entrena un modelo final con los mejores parámetros captados a partir del gridsearch con el stratified K-fold, con la totalidad de los datos de entrenamiento, es decir, tanto del conjunto de entrenamiento como de validación, explotando de esta manera la cantidad de datos de entrenamiento y refinando el modelo una vez más para realizar predicciones con los datos de prueba.

```
modelSVM = SVC(
                C=400,
                gamma=0.001,
                kernel='rbf',
                probability=True,
                random_state= 22
                )
modelSVM.fit(X, y)
```

Modelo MLP

Para realizar el entrenamiento del modelo MLP se utilizó una evaluación usando validación cruzada, a través de la función cross_val_score utilizando una validación de 5 folds (el conjunto de datos se divide en 5 partes, se entrenan 4 y se prueban con 1), finalmente la métrica que nos ayuda a distinguir entre clases es la el AUC de la curva ROC, la cual nos ayuda a minimizar los falso positivos y negativos.

```
mlp_inicial = MLPClassifier(random_state=22)

scores = cross_val_score(mlp_inicial, X_train, y_train, cv=5, scoring='roc_auc')

print("Resultados por fold:", scores)
print("Precisión promedio:", scores.mean())
```

Al terminar la investigación, los hiperparametros que más afectan al modelo son hidden_layer_sizes que nos ayuda a definir la arquitectura de la red (número de capas y neuronas en cada una), activation la cual ayuda a la activación de las neuronas, solver que se encarga de optimizar el algoritmo, learning_rate que nos ayuda a cambiar la tasa de aprendizaje durante el entrenamiento y finalmente max_iter el cual ayuda a definir cuántas iteraciones como máximo se permiten para entrenar el modelo. Posteriormente dentro de la variable cv se definió cómo se realizará una validación cruzada estratificada con 5 particiones de cada fold que actúa mezclando los datos antes de dividirlos. Finalmente se realizó un GridSearch con el fin de encontrar la mejor combinación posible para el modelo.

```
mlp = MLPClassifier(random_state= 22)

param_grid = {
    'hidden_layer_sizes': [
        (300, 200, 100),
        (256,),
        (128,),
        (512,256, 128),
        (512, 256),
        (64,),
        (400, 300, 200, 100),
        (128, 64),
        (150,)
    ],
    'activation': ['logistic', 'relu', 'tanh', 'identity'],
    'solver': ['adam', 'lbfgs', 'sgd'],
    'learning_rate': ['adaptive', 'constant', 'invscaling'],
    'max_iter': [200, 300, 400, 500]
}

cv = StratifiedKFold(
    n_splits=5,
    shuffle=True,
    random_state=22
)

#BUSQUEDA UTILZANDO GRIDSEARCH
grid_search = GridSearchCV(
    estimator=mlp,
    param_grid=param_grid,
    cv=cv,
    scoring='roc_auc',
    n_jobs=-1,
    verbose=1,
    error_score='raise'
)
```

Finalmente se crea el modelo final con los mejores hiperparametros que se encontraron en las fases anteriores para posteriormente entrenar el modelo.

```
best_mlp = MLPClassifier(
    hidden_layer_sizes=(300, 200, 100),
    activation='logistic',
    solver='adam',
    learning_rate='adaptive',
    max_iter=300,
    random_state=22
)

best_mlp.fit(X_train, y_train)
```

Módulo de preprocesamiento

Inicialización del proceso de preprocesamiento

1. Definición de stopwords para vectorización:

   La idea de esta sección inicial es definir un conjunto de palabras clave que realmente no aportan información conceptual útil de la oración, de modo que la vectorización de las oraciones no se vea entorpecida o redundante una vez realizada. La implementación de esto en código consistió en importar una lista de stopwords de una librería especializada, y posteriormente definir stopwords adicionales de forma manual, teniendo así un conjunto extenso y robusto que se obtiene finalmente al juntar ambos conjuntos (stopwords importadas + stopwords adicionales), como se muestra a continuación:

```
# Cargar stopwords en español como lista
stop_words_es = list(stopwords.words('spanish'))
# Agregar stopwords adicionales específicas del dominio
stop_words_adicionales = [
    'rt', 'https', 'http', 'tco', 'twitter', 'com',  # URLs y términos de Twitter
    't', 'co',  # Partes de 't.co'
    'q', 'k', 'd', 'tb', 'tmb', 'pq', 'xq', 'dnd', 'kien', 'salu2', 'aki', 'tqm'  # Abreviaturas comunes
]
stop_words_es.extend(stop_words_adicionales)
# Eliminar duplicados y ordenar
stop_words_es = sorted(list(set(stop_words_es)))
```

2. Definición de posibles abreviaturas:

   Aquí, se define un diccionario de forma que cada llave es una abreviatura, y su contenido es la palabra a la que hace referencia:

```
# Diccionario para abreviaturas comunes en español
ABREVIATURAS = {
    'q': 'que', 'x': 'por', 'd': 'de', 'k': 'que', 'tb': 'también',
    'tmb': 'también', 'pq': 'porque', 'xq': 'porque', 'dnd': 'donde',
    'kien': 'quien', 'salu2': 'saludos', 'aki': 'aquí', 'tqm': 'te quiero mucho',
    'when': 'cuando', 'wtf': 'qué carajos', 'lmao': 'risa', 'lmfao': 'risa',
    'lol': 'risa'
}
```

3. Definición de palabras clave:

   Se agrupan palabras clave que corresponden a conceptos clave de interés para nuestro problema, por lo que se define nuevamente un diccionario, donde cada llave es un concepto clave de relevancia para nuestro análisis, y su contenido es una lista de palabras que forman parte del campo semántico de este concepto. Cabe mencionar que esta lista de palabras es finalmente lematizada debido a que se utilizará sobre un texto lematizado que contenga estas palabras relacionadas:

```
# Palabras clave relacionadas con trastornos alimenticios
PALABRAS_CLAVE = {
    'comida': ['comer', 'comida', 'alimento', 'dieta', 'caloría', 'calorías', 'peso', 'adelgazar', 'delgado', 'delgada'],
    'restriccion': ['no comer', 'ayuno', 'saltar comidas', 'evitar comer', 'prohibir', 'prohibido'],
    'purga': ['vomitar', 'vómito', 'laxante', 'diurético', 'purgar', 'purgante'],
    'imagen_corporal': ['gordo', 'gorda', 'feo', 'fea', 'grasa', 'obeso', 'obesa', 'cuerpo', 'figura'],
    'ejercicio': ['ejercicio', 'gimnasio', 'entrenar', 'quemar calorías', 'sudar']
}
# Lematización las palabras clave del diccionario
PALABRAS_CLAVE_LEMATIZADAS = {
    categoria: [token.lemma_ for token in nlp(" ".join(palabras))]
    for categoria, palabras in PALABRAS_CLAVE.items()
}
```

4. Importación de las funciones para extraer características:

   En este caso, se importan dos funciones críticas para el preprocesamiento diseñado (mismas que serán explicadas a profundidad más adelante, y mismas que se encuentran documentadas en cuanto a código en la siguiente sección de *Funciones*). Esta importación se hizo desde el módulo de preprocesamiento para acceder al script de *funcionesPreprocesamiento.py*, de la siguiente manera:

```
sys.path.append(os.path.abspath("../funciones"))

from funcionesPreprocesamiento import (
    obtener_hashtags_frecuentes_individuales,
    extraer_caracteristicas
)
```

Pipeline principal del preprocesamiento de datos

1. Carga de datos: En primera instancia, se importan ambos conjuntos de datos, tanto de entrenamiento (1,500 entradas de datos) como de pruebas (250 entradas), y se juntan de forma consecutiva en un solo conjunto (1,750 entradas). Cabe mencionar que en este momento este conjunto cuenta únicamente con 4 columnas: *user_id, tweet_id, tweet_text, class*:

```
print("Cargando datos...")
df_train = pd.read_csv("../../data/data_train.csv", encoding="latin1", header=0)
df_test = pd.read_csv("../../data/data_test_fold1.csv", encoding="latin1", header=0)

df = pd.concat([df_train, df_test], ignore_index=True)

print(df.shape)
df.head(-5)
```

```
Cargando datos...
(1750, 4)
```

	user_id	tweet_id	tweet_text	class
0	user0001	0d3ed29586ce	Cheesecake saludable sin azÃºcar y sin lactosa...	control
1	user0002	c3cf897a495b	ser como ellas â□¡â□¡\n #HastaLosHuesos	anorexia
2	user0003	5041d85c45c6	Comida Real o , la clave para estar mÃ¡s sana,...	control
3	user0004	d18285d3c7ec	Entre el cambio de hora y la bajada de las #te...	control
4	user0005	4d81892f3217	Hace mucho tiempo no sentÃa mi cuerpo tan frÃo	anorexia
...	...	...	...	...
1740	user1864	90c3b14b843c	Chuleta de Cerdo al horno sobre cama de ensala...	control

2. Preprocesamiento inicial: Una vez cargado este conjunto y dado que está en formato de dataframe, se llena con una cadena de texto vacía todo valor dentro del atributo de texto que se encuentre sin información (na):

```
df["tweet_text"] = df["tweet_text"].fillna("")
print("Preprocesamiento inicial exitoso :)")
```

3. Extracción de características iniciales: Aquí, buscamos desglosar el texto de en atributos adicionales a partir del contenido de cada tweet, con relevancia considerable; a continuación, se presenta una lista de estas nuevas columnas y su importancia para nuestra problemática:
    - *tweet_text:* Este atributo contiene un texto procesado, al que inicialmente se le removieron los hashtags (*procesar_hashtags(texto)*), luego fue normalizado y limpiado (*limpiar_texto(texto)*), eliminando acentos (esto debido al bajo impacto y faltas de ortografía comunes observadas en los tweets) y convirtiendo todo el texto a minúsculas, luego se expandieron las abreviaturas contenidas con un filtrado con las abreviaturas definidas previamente (*expandir_abreviaturas(texto)*), para posteriormente aplicar diversos filtros con expresiones regulares (*limpieza_final(texto)*), eliminando así cualquier URL, mención, sustituyendo '&' con 'y', eliminando cualquier caracter que no sea alfanumérico y sustituyendo cualquier cantidad de espacios en blanco con un solo espacio. Finalmente, tras haber realizado esta limpieza profunda, se lematiza el texto para tener una forma base de cada palabra y facilitar el procesamiento consecuente (*tokenizar_y_lematizar(texto)*). Es importante mencionar que la definición de cada una de estas funciones mencionadas se encuentra explicada a detalle en la siguiente sección de Funciones.

- *hashtags:* Este atributo consiste en un conjunto de todos los hashtags encontrados en el texto, que fueron extraídos del texto, que serán tratados como palabras clave especiales por su importancia.
- *texto_completo:* Este atributo contiene el texto procesado de *tweet_text*, con los hashtags que estaban previamente en el texto, contenidos en *hashtags*.
- *texto_bert:* Este atributo es similar al anterior, pero sin el texto lematizado, es decir, únicamente con la limpieza profunda previa a la lematización, con los hashtags agregados de vuelta.
- *longitud_texto:* Este atributo contiene la cantidad de los caracteres en el texto lematizado, ya que puede resultar útil o ser una característica relevante para nuestra predicción.
- *num_palabras:* Este atributo contiene la cantidad de palabras del texto lematizado, ya que puede ser de igual manera relevante para nuestro caso.
- Palabras clave**: Aquí, se realiza un barrido de cada palabra del atributo ya mencionado de *texto_completo*, con el diccionario lematizado definido previamente de palabras clave, de tal forma que, a modo de encoding, cada entrada contiene una columna por cada uno de los conceptos clave, indicando la cantidad de palabras clave encontradas en su contenido que correspondan a estos conceptos, cuantificando la relevancia de estos conceptos en el texto.
- *polaridad:* Este atributo corresponde a un análisis de sentimiento, indicando de forma decimal la polaridad, positiva o negativa, del mensaje contenido en el texto, sin lematizar, indicando así la disposición del mensaje y su inclinación emocional.
- *subjetividad:* Este atributo corresponde a un análisis de sentimiento, indicando qué tan subjetivo es el mensaje, aportando así un nivel de análisis profundo y relevante para el entrenamiento.

Es importante mencionar que la definición de cada uno de estos pasos se encuentra en la misma definición de la función *extraer_caracteristicas(texto)*, expuesta en la siguiente sección de Funciones.

```
print("Extrayendo características iniciales...")
features_df = df["tweet_text"].swifter.apply(extraer_caracteristicas).apply(pd.Series)
```

```
print("Características iniciales:")
for caracteristica in features_df.columns:
    print(f'\t- {caracteristica}: {type(caracteristica)}')
```

```
Características iniciales:
        - tweet_text: <class 'str'>
        - hashtags: <class 'str'>
        - texto_completo: <class 'str'>
        - texto_bert: <class 'str'>
        - longitud_texto: <class 'str'>
        - num_palabras: <class 'str'>
        - comida: <class 'str'>
        - restriccion: <class 'str'>
        - purga: <class 'str'>
        - imagen_corporal: <class 'str'>
        - ejercicio: <class 'str'>
        - polaridad: <class 'str'>
        - subjetividad: <class 'str'>
```

Una vez extraídas las características mencionadas previamente, se reemplazan las columnas anteriores, de modo que la columna antigua de *tweet_text* es reemplazada por la nueva columna que ha sido procesada profundamente y lematizada, y se agregan las nuevas columnas creadas a partir de la extracción de características al conjunto principal:

```
# Reemplazar columnas duplicadas
df["tweet_text"] = features_df["tweet_text"]
df["hashtags"] = features_df["hashtags"]

# Agregar columnas nuevas
columnas_nuevas = features_df.drop(columns=["tweet_text", "hashtags"])
df = pd.concat([df, columnas_nuevas], axis=1)

print(f'Cantidad de columnas hasta este punto: {df.shape[1]}')
print("Columnas de nuestro dataset hasta este punto:")
print(f'--> ', end='')
for col in df.columns:
    print(f', {col}', end='')
print('\n')
```

Python

```
Cantidad de columnas hasta este punto: 16
Columnas de nuestro dataset hasta este punto:
--> , user_id, tweet_id, tweet_text, class, hashtags, texto_completo, texto_bert, longitud_texto, num_palabras, com
```

4. Procesar hashtags: El conjunto de hashtags hallado en la columna *hashtags*, es desglosado en columnas, una por cada hashtag de todo el conjunto, con la nomenclatura tag_{hashtag}, a modo de one-hot encoding, de tal manera que en caso de contener ese hashtag, será indicado con un '1' en la columna, y con un '0' en caso de no contenerlo.

```
print("Procesando hashtags...")
mlb = MultiLabelBinarizer()
hashtags_df = pd.DataFrame(mlb.fit_transform(df["hashtags"]),
                           columns=[f"tag_{tag}" for tag in mlb.classes_])
```

Una vez hecho esto, se separa este nuevo dataframe de hashtags en datos de entrenamiento y de prueba, teniendo así ahora un conjunto de diversos hashtags con 1,500 entradas correspondientes al entrenamiento, y un conjunto de diversos hashtags con 250 entradas correspondientes a las pruebas.

```
# Primera separación, para resguardar los hashtags de prueba y no influenciarlos con los vistos en entrenamiento:
hashtags_df_train = hashtags_df.iloc[:-250].copy()
hashtags_df_test = hashtags_df.iloc[-250:].copy()

print(f'Hashtags de entrenamiento: {hashtags_df_train.shape}')
print(f'Hashtags de prueba: {hashtags_df_test.shape}')
```

```
Hashtags de entrenamiento: (1500, 2312)
Hashtags de prueba: (250, 2312)
```

Esta separación se hace con la finalidad de hacer una filtración especial para el entrenamiento, ya que buscamos quedarnos únicamente con aquellos hashtags que sean utilizados en 10 o más tweets del conjunto de entrenamiento, quedándonos así con los más relevantes en el conjunto de tweets y poder entrenar con estos datos. Se hace esta separación porque no nos sirve quedarnos con los tweets relevantes para el conjunto que no será utilizado para el entrenamiento, por lo que el filtro se aplica a ambos conjuntos.

```
# Filtrar hashtags frecuentes
umbral = 10
hashtags_frecuentes = hashtags_df_train.columns[hashtags_df_train.sum() >= umbral]
hashtags_frecuentes_df_train = hashtags_df_train[hashtags_frecuentes]
print(f'{len(list(hashtags_frecuentes_df_train))} hashtags frecuentes de entrenamiento: {list(hashtags_frecuentes_df_train)}')
```

Python

```
59 hashtags frecuentes de entrenamiento: ['tag_Anorexia', 'tag_Bulimia', 'tag_ED', 'tag_RexyBill', 'tag_Salud', 'tag_TCA', 'tag_Thinspo',
```

```
# Filtrarlos para el conjunto de prueba
hashtags_frecuentes_df_test = hashtags_df_test[hashtags_frecuentes]
print(f'{len(list(hashtags_frecuentes_df_test))} filtro aplicado para el conjunto de prueba: {list(hashtags_frecuentes_df_test)}')

print("Prueba:", list(hashtags_frecuentes_df_train) == list(hashtags_frecuentes_df_test))
```

Python

```
59 filtro aplicado para el conjunto de prueba: ['tag_Anorexia', 'tag_Bulimia', 'tag_ED', 'tag_RexyBill', 'tag_Salud', 'tag_TCA', 'tag_Thi
Prueba: True
```

Finalmente, se vuelven a juntar los conjuntos de hashtags, habiendo así filtrado los hashtags más relevantes para el conjunto de entrenamiento y reflejándolo tanto en el de entrenamiento como en el de prueba.

```
hashtags_frecuentes_df = pd.concat([hashtags_frecuentes_df_train, hashtags_frecuentes_df_test], ignore_index=True)
print(hashtags_frecuentes_df.shape)
hashtags_frecuentes_df
```

```
(1750, 59)
```

	tag_Anorexia	tag_Bulimia	tag_ED	tag_RexyBill	tag_Salud	tag_TCA	tag_Thinspo	tag_adelgazar	tag_alimentacionsaludable	tag_ana	...
0	0	0	0	0	0	0	0	0	0	0	...
1	0	0	0	0	0	0	0	0	0	0	...
2	0	0	0	0	0	0	0	0	0	0	...
3	0	0	0	0	0	0	0	0	0	0	...
4	0	0	0	0	0	0	0	0	0	0	...
...	...	...	...	...	...	...	...	...	...	...	...
745	0	0	0	0	0	0	0	0	0	0	...
746	0	0	0	0	0	0	0	0	0	0	...
747	0	0	0	0	0	0	0	1	1	0	...
748	0	0	0	0	0	0	0	0	0	0	...
749	0	0	0	0	0	0	0	0	0	0	...

50 rows × 59 columns

5. Preparar texto completo: Se genera una nueva columna de *texto_completo*, conteniendo el texto procesado lematizado junto con cada hashtag del texto original sin el signo '#' y separados entre si con un solo espacio, siempre y cuando sea un hashtag con una frecuencia de 5 o mayor en todo el conjunto de datos.

```
umbral_bajo = 5
hashtags_vectorizacion = hashtags_df.columns[hashtags_df.sum() >= umbral_bajo]
hashtags_validos = {col.replace('tag_', '') for col in hashtags_vectorizacion}
df["hashtags_frecuentes_bajos"] = df["hashtags"].apply(lambda h: obtener_hashtags_frecuentes_individuales(h, hashtags_validos))
df["texto_completo"] = df["tweet_text"] + " " + df["hashtags_frecuentes_bajos"]
```

6. Vectorización TF-IDF: Aplicamos una vectorización del tipo TF-IDF al texto completo realizado en el paso anterior, ya que este tipo de vectorización nos permite distinguir características más relevantes dentro de nuestro conjunto según la frecuencia de cada una. Decidimos utilizar un máximo de 1,300 características debido a la extensión del conjunto de datos y un rango de n-gramas de 1 a 3, captando así características con cierto contexto e información relevante al momento de definir frecuencias de características. Es aquí donde utilizamos el conjunto definido de stopwords, y especificamos además la frecuencia máxima y mínima de características para poder entrar en nuestra vectorización (85% y de 2, respectivamente). Finalmente utilizamos un escalado sublinear por lo mismo de la extensión del conjunto, y una similitud de coseno definida por el producto punto de los vectores por estar utilizando l2. Además, separamos la columna de clase para predicción.

```
print("Aplicando vectorización TF-IDF...")
vectorizer = TfidfVectorizer(
    max_features=1300,
    ngram_range=(1,3),
    stop_words=stop_words_es,
    min_df=2,
    max_df=0.85,
    sublinear_tf=True,
    norm='l2'
)
X_tfidf = vectorizer.fit_transform(df["texto_completo"])
y = df["class"]
```

```
print("\nInformación sobre las características TF-IDF:")
print(f"Número total de características: {X_tfidf.shape[1]}")
print(f"Número de muestras: {X_tfidf.shape[0]}")
print("\nTop 10 términos más importantes:")
feature_names = vectorizer.get_feature_names_out()
idf_values = vectorizer.idf_
top_terms = sorted(zip(feature_names, idf_values), key=lambda x: x[1], reverse=True)[:10]
for term, idf in top_terms:
    print(f"{term}: {idf:.2f}")
```

```
Información sobre las características TF-IDF:
Número total de características: 1300
Número de muestras: 1750

Top 10 términos más importantes:
3x8: 7.37
agregar: 7.37
antojo: 7.37
arte: 7.37
be: 7.37
cabra: 7.37
camarón: 7.37
dietar: 7.37
fortalecer: 7.37
kg peso: 7.37
```

7. Construcción del dataset final: Para construir nuestro dataset final, convertimos nuestra vectorización TF-IDF en un dataframe, para poderlo integrar más adelante.

```
print("Creando DataFrame final...")
tfidf_df = pd.DataFrame(X_tfidf.toarray(), columns=[f"tfidf_{i}" for i in range(X_tfidf.shape[1])])

tfidf_df
```

```
Creando DataFrame final...
```

	tfidf_0	tfidf_1	tfidf_2	tfidf_3	tfidf_4	tfidf_5	tfidf_6	tfidf_7	tfidf_8	tfidf_9	...	tfidf_1290	tfidf_1291	tfidf_1292	tfidf_1293	tfidf_1294	tfidf_1295	tfidf_1296	tfidf_1297	tfidf_1298	tfidf_1299
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1745	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1746	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1747	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1748	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1749	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
1750 rows × 1300 columns
```

Una vez realizado esto, es importante agrupar las columnas generadas a lo largo de este preprocesamiento según su aportación al entrenamiento. De forma más específica, las columnas fueron clasificadas en 4 aspectos (sin contar la vectorización); columnas base (*tweet_id, tweet_text, texto_completo, texto_bert*), columnas métricas (*longitud_texto, num_palabras*), columnas de palabras clave (*comida, restriccion, purga, imagen_corporal, ejercicio*) y columnas sentimiento (*polaridad, subjetividad*).

```
# Seleccionar columnas para el DataFrame final
columnas_base = ["tweet_id", "tweet_text", "texto_completo", "texto_bert"]
columnas_metricas = ["longitud_texto", "num_palabras"]
columnas_palabras_clave = ["comida", "restriccion", "purga", "imagen_corporal", "ejercicio"]
columnas_sentimiento = ["polaridad", "subjetividad"]
```

Ya que contamos con esta clasificación, podemos armar nuestro dataset final.

```python
# Crear el DataFrame final sin la columna class
df_final = pd.concat([
    df[columnas_base],
    df[columnas_metricas],
    df[columnas_palabras_clave],
    df[columnas_sentimiento],
    hashtags_frecuentes_df,
    tfidf_df
], axis=1)

print(f'Número total de columnas finales: {df_final.shape[1]}')
df_final.head(5)
```

```
Número total de columnas finales: 1372
```

	tweet_id	tweet_text	texto_completo	texto_bert	longitud_texto
0	0d3ed29586ce	cheesecakir saludable azucar lactosa mermerlad...	cheesecakir saludable azucar lactosa mermerlad...	cheesecake saludable sin azucar y sin lactosa ...	59
1	c3cf897a495b			ser como ellas HastaLosHuesos	0

8. Normalización de características: Se escalan las columnas numéricas debido a su relación dispar, de modo que se encuentren en una misma escala y el entrenamiento se vuelva más eficaz. las columnas a escalar son *longitud_texto, num_palabras, polaridad* y *subjetividad*.

```python
columnas_a_escalar = ["longitud_texto", "num_palabras", "polaridad", "subjetividad"]
scaler = StandardScaler()
df_final[columnas_a_escalar] = scaler.fit_transform(df_final[columnas_a_escalar])
```

Además, la columna de clase, es decir, la columna 'y' alrededor de la cual se va a entrenar para realizar predicciones, se convierte en una columna binaria, facilitando en gran medida el entrenamiento clasificatorio, siendo 1 padecimiento de anorexia y 0 el caso sano de control. Finalmente, se separan por completo los conjuntos de datos de entrenamiento y de prueba, habiendo preprocesado exitosamente los datos.

```python
# Convertir etiquetas a valores numéricos y agregar al final
df_final['class'] = df['class'].map({'control': 0, 'anorexia': 1})

df_final_train = df_final.iloc[:-250].copy()
df_final_test = df_final.iloc[-250:].copy()

print(f'Tamaño de dataset de entrenamiento: {df_final_train.shape}')
df_final_train
```

9. Guardar datasets con todas las columnas generadas: Se guarda en dos archivos .csv los datasets de entrenamiento y de prueba, con todas el dataset final construido con las columnas generadas.

```
print("Guardando resultados...")
df_final_train.to_csv("../../data/NO_USAR_tweets_procesados_TRAIN.csv", index=False, encoding="utf-8")
df_final_test.to_csv("../../data/NO_USAR_tweets_procesados_TEST.csv", index=False, encoding="utf-8")
```

10. Guardar datasets para entrenamiento y pruebas de modelos tradicionales: Se guarda en dos archivos .csv los datasets de entrenamiento y de prueba, con todas las columnas categóricas y numéricas, para el entrenamiento, validación y prueba que usarán nuestros modelos tradicionales de clasificación.

```
print("Guardando resultados para modelos tradicionales...")
ds_tradicional_train = df_final_train.drop(columns=["tweet_id", "tweet_text", "texto_completo", "texto_bert"])
ds_tradicional_test = df_final_test.drop(columns=["tweet_id", "tweet_text", "texto_completo", "texto_bert"])

ds_tradicional_train.to_csv("../../data/ds_tradicional.csv", index=False, encoding="utf-8")
ds_tradicional_test.to_csv("../../data/ds_tradicional_TEST.csv", index=False, encoding="utf-8")
```

Gracias a este preprocesamiento profundo y robusto, es que adquirimos un conjunto de datos de calidad para el entrenamiento de nuestros modelos.

```python
    print("\nDataset final guardado con las siguientes columnas:")
    print("\nColumnas base:", columnas_base)
    print("\nMétricas estilísticas:", columnas_metricas)
    print("\nPalabras clave:", columnas_palabras_clave)
    print("\nAnálisis de sentimiento:", columnas_sentimiento)
    print("\nHashtags frecuentes:", list(hashtags_frecuentes))
    print("\nTotal de características TF-IDF:", X_tfidf.shape[1])
    print("\nDistribución de clases de entrenamiento:")
    print(df_final_train['class'].value_counts())
    print("\nDistribución de clases de prueba:")
    print(df_final_test['class'].value_counts())
```

```
Dataset final guardado con las siguientes columnas:

Columnas base: ['tweet_id', 'tweet_text', 'texto_completo', 'texto_bert']

Métricas estilísticas: ['longitud_texto', 'num_palabras']

Palabras clave: ['comida', 'restriccion', 'purga', 'imagen_corporal', 'ejercicio']

Análisis de sentimiento: ['polaridad', 'subjetividad']

Hashtags frecuentes: ['tag_Anorexia', 'tag_Bulimia', 'tag_ED', 'tag_RexyBill', 'tag_S

Total de características TF-IDF: 1300

Distribución de clases de entrenamiento:
class
1    804
0    696
Name: count, dtype: int64

Distribución de clases de prueba:
class
1    134
0    116
Name: count, dtype: int64
```

Pruebas Unitarias

Preprocesamiento

Esta prueba se encarga de cerciorarse de que la función *'limpiar_texto'* remueva caracteres (signo de admiración e interrogación de cierre y comas) y se encuentre en minúsculas.

```python
def test_limpiar_texto_normal():
    texto = "¡Hola! ¿Cómo estás? Bien, ¿y tú?"
    resultado = limpiar_texto(texto)
    assert resultado == "hola! como estas? bien, y tu?", f"Resultado inesperado: {resultado}"
```

Esta prueba se encarga de asegurarse de que, en caso de pasar la función '*limpiar_texto',* y no haya texto válido, esta envie un error.

```
def test_limpiar_texto_con_nulo():
    resultado = limpiar_texto(None)
    assert resultado == "", f"Esperado '', pero se obtuvo: {resultado}"
```

Esta prueba se encarga de revisar que la función '*limpiar_texto',* remueva caracteres especiales como acentos y la Virgulilla de la 'ñ' para sustituirlo por una 'n'.

```
def test_limpiar_texto_con_caracteres_especiales():
    texto = "niño con café golpea piñata"
    resultado = limpiar_texto(texto)
    assert resultado == "nino con cafe golpea pinata"
```

Esta prueba se encarga de revisar que la función '*expandir_abreviaturas',* sea capaz de entender abreviaturas como *'q'* para la palabra que, *'xq'* como porque y *'tmb'* como también.

```
def test_expandir_abreviaturas_conocidas():
    texto = "q haces xq no vienes tmb"
    resultado = expandir_abreviaturas(texto)
    assert resultado == "que haces porque no vienes también"
```

Esta prueba se encarga de revisar que la función '*expandir_abreviaturas'* no corrija abreviaturas si no detecta, pero corrija abreviaturas en caso de que haya.

```
def test_expandir_abreviaturas_mixto():
    texto = "hola q tal"
    resultado = expandir_abreviaturas(texto)
    assert resultado == "hola que tal"
```

Esta prueba se encarga de revisar que la función '*expandir_abreviaturas'* no corrija abreviaturas si no hay en el texto.

```
def test_expandir_abreviaturas_sin_abreviaturas():
    texto = "hola como estas"
    resultado = expandir_abreviaturas(texto)
    assert resultado == "hola como estas"
```

Esta prueba se encarga de revisar que la función '*procesar_hashtags'* pueda procesar más de un hashtag en una cadena de texto y separar en un diccionario de strings sin el símbolo *'#'* y la cadena de texto sin los hashtags.

```
def test_procesar_hashtags_multiples():
    texto = "Amo el #cafe y la #vida"
    texto_limpio, hashtags = procesar_hashtags(texto)
    assert texto_limpio == "Amo el  y la"
    assert hashtags == ["cafe", "vida"]
```

Esta prueba se encarga de revisar que la función '*procesar_hashtags'* pueda procesar una cadena de texto sin hashtags y retornar tanto un diccionario vacío como la cadena original.

```python
def test_procesar_hashtags_sin_hashtags():
    texto = "Buenos días profe, ¿Cómo está?"
    texto_limpio, hashtags = procesar_hashtags(texto)
    assert texto_limpio == "Buenos días profe, ¿Cómo está?"
    assert hashtags == []
```

Esta prueba se encarga de revisar que la función '*procesar_hashtags'* pueda procesar un hashtag en una cadena de texto, sin importar la longitud del hashtag y si es lo único en la cadena; por lo que devuelve una cadena de texto vacía y un diccionario con el valor único del hashtag sin el símbolo *'#'*.

```python
def test_procesar_hashtags_unico():
    texto = "#yosoypablo"
    texto_limpio, hashtags = procesar_hashtags(texto)
    assert texto_limpio == ""
    assert hashtags == ["yosoypablo"]
```

Esta prueba se encarga de revisar que la función '*limpieza_final'* pueda remover hipervínculos y menciones de usuarios que sean precedidos por el símbolo *'@'*.

```python
def test_limpieza_final_urls_y_menciones():
    texto = "Visita http://ejemplo.com y menciona a @usuario"
    resultado = limpieza_final(texto)
    assert resultado == "Visita y menciona a", f"Resultado inesperado: {resultado}"
```

Esta prueba se encarga de revisar que la función '*limpieza_final'* pueda remover caracteres especiales.

```python
def test_limpieza_final_simbolos():
    texto = "Hola & bienvenid@! #fiesta :)"
    resultado = limpieza_final(texto)
    assert resultado == "Hola y bienvenid fiesta", f"Resultado inesperado: {resultado}"
```

Esta prueba se encarga de revisar que la función '*limpieza_final'* pueda remover espacios en caso de que haya más de uno seguido.

```python
def test_limpieza_final_espacios_extra():
    texto = "  Esto    tiene   muchos   espacios   "
    resultado = limpieza_final(texto)
    assert resultado == "Esto tiene muchos espacios"
```

Esta prueba se encarga de revisar que al aplicar la función '*tokenizar_y_lematizar'* en un texto, se haya lematizado de forma correcta

```python
def test_tokenizar_y_lematizar_basico():
    texto = "Los gatos están durmiendo en la casa"
    resultado = tokenizar_y_lematizar(texto)
    assert "gato" in resultado and "dormir" in resultado, f"Lematización incorrecta: {resultado}"
```

Esta prueba se encarga de revisar que al aplicar la función '*tokenizar_y_lematizar'* en un texto, se haya lematizado de forma correcta

```python
def test_tokenizar_y_lematizar_con_stopwords():
    texto = "Este es un texto de ejemplo para probar"
    resultado = tokenizar_y_lematizar(texto)
    assert "este" not in resultado and "es" not in resultado, f"Stopwords no filtradas: {resultado}"
```

Esta prueba se encarga de revisar que la función '*tokenizar_y_lematizar'* genere una cadena vacía de texto en caso de que el texto esté compuesto de stopwords y palabras sin aportación.

```python
def test_tokenizar_y_lematizar_vacio():
    texto = ""
    resultado = tokenizar_y_lematizar(texto)
    assert resultado == "", f"Esperado '', obtenido: {resultado}"
```

Esta prueba se encarga de revisar que la función '*calcular_metricas_estilisticas'* esté calculando de manera adecuada el número de palabras y la longitud de una cadena de texto dada.

```python
def test_metricas_texto_normal():
    texto = "Este es un ejemplo de texto."
    resultado = calcular_metricas_estilisticas(texto)
    assert resultado['num_palabras'] == 6, f"Número de palabras incorrecto: {resultado['num_palabras']}"
    assert resultado['longitud_texto'] == len(texto), f"Longitud incorrecta: {resultado['longitud_texto']}"
```

Esta prueba se encarga de revisar que la función '*calcular_metricas_estilisticas'* esté calculando de manera adecuada el número de palabras y la longitud de una cadena de texto vacía.

```python
def test_metricas_texto_vacio():
    texto = ""
    resultado = calcular_metricas_estilisticas(texto)
    assert resultado['num_palabras'] == 0
    assert resultado['longitud_texto'] == 0
```

Esta prueba se encarga de revisar que la función '*calcular_metricas_estilisticas'* esté calculando de manera adecuada el número de palabras y la longitud de una cadena de texto dada.

```python
def test_metricas_texto_con_espacios():
    texto = "   Palabra   otra   más   "
    resultado = calcular_metricas_estilisticas(texto)
    assert resultado['num_palabras'] == 3
```

Esta prueba se encarga de revisar que la función '*analizar_palabras_clave',* pueda revisar si encuentra presencia en la cadena de texto si hay presencia de palabras clave.

```python
def test_analizar_palabras_clave_presencia():
    texto = "me voy a saltar la comida y provocar el vómito"
    resultado = analizar_palabras_clave(texto)
    assert resultado['comida'] == 1
    assert resultado['restriccion'] == 2
    assert resultado['purga'] == 1
```

Esta prueba se encarga de revisar que la función '*analizar_palabras_clave',* pueda revisar en una cadena encuentre pocas palabras clave.

```python
def test_analizar_palabras_clave_ausencia():
    texto = "no hay coincidencias"
    resultado = analizar_palabras_clave(texto)
    assert resultado['comida'] == 0
    assert resultado['restriccion'] == 1
    assert resultado['purga'] == 0
    assert resultado['imagen_corporal'] == 0
    assert resultado['ejercicio'] == 0
```

Esta prueba se encarga de revisar que la función '*traducir_si_necesario',* no traduzca en caso de que esté en español ya la cadena.

```python
def test_traducir_texto_espanol_a_ingles():
    texto = "Estoy feliz"
    resultado = traducir_si_necesario(texto)
    assert isinstance(resultado, str)
    assert "happy" in resultado.lower()
```

Esta prueba se encarga de revisar que la función '*traducir_si_necesario'*, traduzca al español en caso de que esté no lo esté la cadena original.

```python
def test_traducir_texto_ingles_a_ingles():
    texto = "I am happy"
    resultado = traducir_si_necesario(texto)
    assert isinstance(resultado, str)
    assert "happy" in resultado.lower()
```

Esta prueba se encarga de revisar que la función '*analizar_sentimiento_negativo'*, haga un análisis sentimental de una cadena con sentimiento positivo y la catalogue como tal.

```python
def test_analizar_sentimiento_positivo():
    texto = "Estoy muy feliz con mi cuerpo"
    resultado = analizar_sentimiento(texto)
    assert resultado['polaridad'] > 0, f"Se esperaba sentimiento positivo: {resultado}"
```

Esta prueba se encarga de revisar que la función '*analizar_sentimiento_negativo'*, haga un análisis sentimental de una cadena con sentimiento negativo y la catalogue como tal.

```python
def test_analizar_sentimiento_negativo():
    texto = "Esto fue una pérdida de tiempo, muy mal"
    resultado = analizar_sentimiento(texto)
    assert resultado['polaridad'] < 0, f"Se esperaba sentimiento negativo: {resultado}"
```

Esta prueba se encarga de revisar que la función '*obtener_hashtags_frecuentes_individuales'* retorne una cadena con los hashtags que se repiten en ambos diccionarios.

```python
def test_obtener_hashtags_frecuentes_individuales():
    hashtags_fila = ["felicidad", "amor", "vida", "random"]
    hashtags_validos = {"felicidad", "vida", "salud"}

    resultado = obtener_hashtags_frecuentes_individuales(hashtags_fila, hashtags_validos)
    assert resultado == "felicidad vida", f"Resultado inesperado: {resultado}"
```

Esta prueba se encarga de comprobar que la función '*obtener_hashtags_frecuentes_individuales'* devuelva una cadena vacía cuando ninguno de los hashtags evaluados sean válidos.

```python
def test_obtener_hashtags_sin_validos():
    hashtags_fila = ["invalido1", "invalido2"]
    hashtags_validos = {"felicidad", "vida"}

    resultado = obtener_hashtags_frecuentes_individuales(hashtags_fila, hashtags_validos)
    assert resultado == "", f"Esperado '', pero se obtuvo: {resultado}"
```

Este test se encarga de que la función '*obtener_hashtags_frecuentes_individuales'* retorne correctamente todos los hashtags cuando todos los presentes en la entrada son considerados válidos.

```python
def test_obtener_hashtags_todos_validos():
    hashtags_fila = ["vida", "felicidad"]
    hashtags_validos = {"vida", "felicidad"}
```

Este test se encarga de verificar que la función '*obtener_hashtags_frecuentes_individuales'* pueda manejar correctamente un tweet que no contenga un hashtag.

```python
def test_obtener_hashtags_lista_vacia():
    hashtags_fila = []
    hashtags_validos = {"vida", "salud"}

    resultado = obtener_hashtags_frecuentes_individuales(hashtags_fila, hashtags_validos)
    assert resultado == ""
```

Este test se encarga de verificar el comportamiento de la función *'extraer_caracterisitcas'* cuando se llega a procesar un tweet con hashtags separándolo por texto, hashtags, y usarlo en bert.

```python
def test_extraer_caracteristicas_basico():
    tweet = "Estoy en el hospital y me siento bien. #salud #bienestar"
    resultado = extraer_caracteristicas(tweet)

    assert "tweet_text" in resultado
    assert "hashtags" in resultado
    assert "texto_completo" in resultado
    assert "texto_bert" in resultado
    assert isinstance(resultado["hashtags"], list)
    assert "salud" in resultado["texto_completo"]
    assert isinstance(resultado["polaridad"], float)
```

Este test se encarga de verificar el comportamiento de la función *'extraer_caracterisitcas'* cuando se llega a procesar un tweet que no contiene hashtags, pero que contenga texto regular.

```python
def test_extraer_caracteristicas_sin_hashtags():
    tweet = "Nada que ver aquí, solo texto normal."
    resultado = extraer_caracteristicas(tweet)

    assert resultado["hashtags"] == []
    assert "texto_completo" in resultado
    assert isinstance(resultado["num_palabras"], int)
```

Este test verifica que la función *'extraer_caracterisitcas'* pueda manejar correctamente un tweet vacío.

```python
def test_extraer_caracteristicas_vacio():
    tweet = ""
    resultado = extraer_caracteristicas(tweet)
    assert resultado["tweet_text"] == ""
    assert resultado["hashtags"] == []
    assert resultado["num_palabras"] == 0
    assert resultado["polaridad"] == 0
```

Entrenamiento

test_cargar_datos_entrenamiento: Verifica que la función elimina la columna class de las características y que las etiquetas se obtienen correctamente.

```python
def test_cargar_datos_entrenamiento(monkeypatch, mock_df):
    def mock_read_csv(path):
        return mock_df

    monkeypatch.setattr(pd, "read_csv", mock_read_csv)
    X, y = cargar_datos_entrenamiento()
    assert "class" not in X.columns
    assert y.tolist() == [0, 1, 0]
```

test_cargar_datos_entrenamiento_forma: Comprueba que las dimensiones de las características son las esperadas (3 filas, 1368 columnas).

```
def test_cargar_datos_entrenamiento_forma(monkeypatch, mock_df):
    def mock_read_csv(path):
        return mock_df

    monkeypatch.setattr(pd, "read_csv", mock_read_csv)
    X, y = cargar_datos_entrenamiento()
    assert 1368 == X.shape[1]
    assert 3 == X.shape[0]
```

test_cargar_datos_prueba: Asegura que la función elimina la columna class y extrae las etiquetas correctamente del conjunto de prueba.

```
def test_cargar_datos_prueba(monkeypatch, mock_df):
    def mock_read_csv(path):
        return mock_df

    monkeypatch.setattr(pd, "read_csv", mock_read_csv)
    X, y = cargar_datos_prueba()
    assert "class" not in X.columns
    assert y.tolist() == [0, 1, 0]
```

test_cargar_datos_prueba_forma: Verifica que el conjunto de prueba tiene las dimensiones adecuadas (3 filas, 1368 columnas).

```
def test_cargar_datos_prueba_forma(monkeypatch, mock_df):
    def mock_read_csv(path):
        return mock_df

    monkeypatch.setattr(pd, "read_csv", mock_read_csv)
    X, y = cargar_datos_prueba()
    assert 1368 == X.shape[1]
    assert 3 == X.shape[0]
```

test_imprimir_forma_pequeña: Verifica que se retorna correctamente la forma y el encabezado de un DataFrame pequeño.

```python
def test_imprimir_forma_pequeña():
    df = pd.DataFrame({
        "col1": [1, 2, 3, 4, 5, 6],
        "class": [0, 1, 0, 1, 1, 0]
    })
    shape, head = imprimir_forma(df)
    assert shape == (6, 2)
    assert head.shape == (5, 2)
```

test_imprimir_forma_grande: Valida que se imprime la forma y encabezado de un DataFrame grande con muchas columnas.

```python
def test_imprimir_forma_grande(monkeypatch, mock_df):
    def mock_read_csv(path):
        return mock_df

    monkeypatch.setattr(pd, "read_csv", mock_read_csv)
    X, _ = cargar_datos_prueba()
    shape, head = imprimir_forma(X)
    assert shape == (3, 1368)
    assert head.shape == (3, 1368)
```

test_division_train_val_shapes: Confirma que la división entre entrenamiento y validación es del 80%-20%.

```python
def test_division_train_val_shapes():
    X = pd.DataFrame({'a': range(100)})
    y = pd.Series([0]*50 + [1]*50)
    X_train, X_val, y_train, y_val = division_train_val(X, y)

    assert len(X_train) == 80
    assert len(X_val) == 20
    assert len(y_train) == 80
    assert len(y_val) == 20
```

test_division_train_val_stratification: Comprueba que se mantiene la proporción de clases al hacer la división (estratificación).

```python
def test_division_train_val_stratification():
    X = pd.DataFrame({'a': range(100)})
    y = pd.Series([0]*70 + [1]*30)
    _, _, y_train, y_val = division_train_val(X, y)

    assert y_train.value_counts(normalize=True).round(1).tolist() == [0.7, 0.3]
    assert y_val.value_counts(normalize=True).round(1).tolist() == [0.7, 0.3]
```

test_reporte_clasificacion_proba: Verifica que la función retorna predicciones, residuales y un reporte como texto conteniendo métricas de clasificación.

```python
def test_reporte_clasificacion_proba():
    X, y = make_classification(n_samples=100, n_features=5, random_state=1)
    model = RandomForestClassifier(random_state=1)
    model.fit(X, y)

    y_pred, y_res, reporte = reporte_clasificacion(X, y, model)

    assert len(y_pred) == len(y)
    assert len(y_res) == len(y)
    assert isinstance(reporte, str)
    assert "precision" in reporte.lower()
```

test_reporte_clasificacion_lineal: Valida que con un modelo lineal también se genera un reporte de métricas correctamente.

```python
def test_reporte_clasificacion_lineal():
    X, y = make_classification(n_samples=100, n_features=5, random_state=1)
    model = LogisticRegression()
    model.fit(X, y)

    y_pred, y_res, reporte = reporte_clasificacion(X, y, model, lineal=True)

    assert len(y_pred) == len(y)
    assert len(y_res) == len(y)
    assert isinstance(reporte, str)
    assert "recall" in reporte.lower()
```

test_crear_matriz_confusion_basica: Comprueba que se genera correctamente la matriz de confusión y el objeto visualizable.

```python
def test_crear_matriz_confusion_basica():
    y_test = [0, 1, 0, 1, 0, 1]
    y_pred = [0, 1, 1, 1, 0, 0]
    cm, disp = crear_matriz_confusion(y_test, y_pred)
    assert cm.shape == (2, 2)
    assert isinstance(disp, ConfusionMatrixDisplay)
    assert cm[0][0] == 2  # TN (True negatives)
    assert cm[1][1] == 2  # TP (True positives)
```

test_crear_matriz_confusion_binaria_perfecta: Valida que la matriz de confusión refleja una clasificación perfecta sin errores.

```
def test_crear_matriz_confusion_binaria_perfecta():
    y_test = [1, 0, 1, 0]
    y_pred = [1, 0, 1, 0]
    cm, _ = crear_matriz_confusion(y_test, y_pred)
    assert np.array_equal(cm, np.array([[2, 0], [0, 2]])) # Aquí se forma una matriz perfecta, sin falsos
```

test_calcular_roc_auc_resultados_correctos: Verifica que se devuelven correctamente las curvas ROC y el valor del AUC.

```
def test_calcular_roc_auc_resultados_correctos():
    y_true = np.array([0, 0, 1, 1])
    y_scores = np.array([0.1, 0.4, 0.35, 0.8])
    fpr, tpr, thresholds, auc_score = calcular_roc_auc(y_true, y_scores)
    assert isinstance(fpr, np.ndarray)
    assert isinstance(tpr, np.ndarray)
    assert isinstance(thresholds, np.ndarray)
    assert isinstance(auc_score, float)
    assert 0 <= auc_score <= 1
```

test_calcular_roc_auc_auc_valido: Asegura que el valor AUC calculado es correcto comparado con roc_auc_score.

```
def test_calcular_roc_auc_auc_valido():
    y_true = [0, 1, 1, 0, 1]
    y_scores = [0.1, 0.9, 0.8, 0.4, 0.6]
    _, _, _, auc_score = calcular_roc_auc(y_true, y_scores)
    assert round(auc_score, 2) == round(roc_auc_score(y_true, y_scores), 2)
```

test_metricas_tpr_fpr_valores_correctos: Verifica que se calculan correctamente la TPR (sensibilidad) y la FPR a partir de la matriz de confusión.

```
def test_metricas_tpr_fpr_valores_correctos():
    cm = np.array([[50, 10],
                   [5, 35]])      # TN=50, FP=10, FN=5, TP=35
    TPR, FPR = metricas_tpr_fpr(cm)
    assert round(TPR, 2) == 0.88  # 35 / (35 + 5)
    assert round(FPR, 2) == 0.17  # 10 / (10 + 50)
```

test_metricas_tpr_fpr_edge_case: Valida que la función maneja correctamente casos extremos (por ejemplo, sin verdaderos negativos).

```python
def test_metricas_tpr_fpr_edge_case():
    cm = np.array([[0, 0],
                   [0, 1]])  # TN=0, FP=0, FN=0, TP=1
    TPR, FPR = metricas_tpr_fpr(cm)
    assert TPR == 1.0
    assert FPR == 0.0
```

test_hacer_pepinillo_crea_archivo_temporal: Verifica que la función guarda correctamente el modelo en un archivo .pkl.

```python
def test_hacer_pepinillo_crea_archivo_temporal(tmp_path):
    modelo = LogisticRegression()
    ruta_modelo = tmp_path / "modelo.pkl"

    hacer_pepinillo(modelo, str(ruta_modelo), test=True)

    assert ruta_modelo.exists()
```

test_hacer_pepinillo_modelo_valido: Asegura que el modelo guardado puede cargarse nuevamente y conserva su tipo original.

```python
def test_hacer_pepinillo_modelo_valido(tmp_path):
    modelo = LogisticRegression()
    ruta_modelo = tmp_path / "modelo_valido.pkl"

    hacer_pepinillo(modelo, str(ruta_modelo), test=True)

    with open(ruta_modelo, "rb") as f:
        modelo_cargado = pickle.load(f)

    assert isinstance(modelo_cargado, LogisticRegression)
```

Comparación final

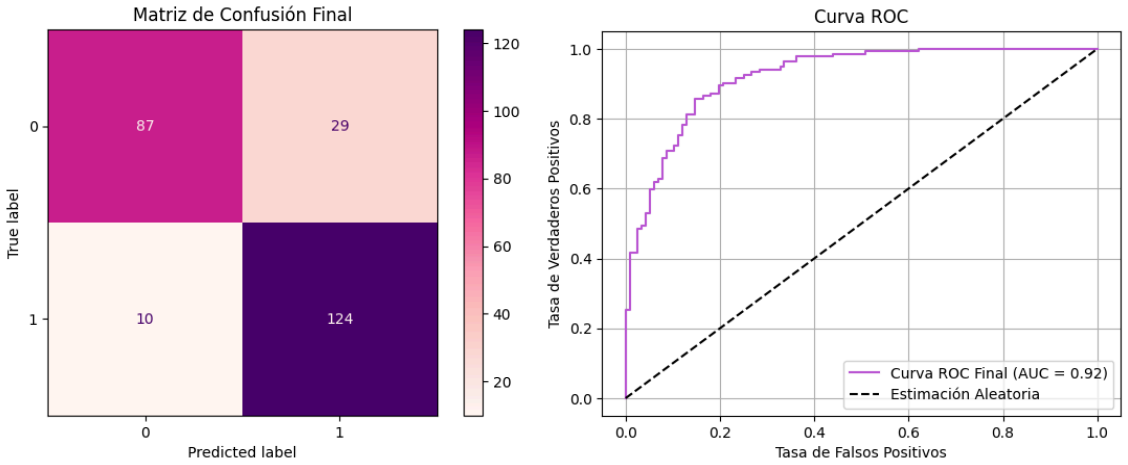
Para comparar los diferentes modelos aplicados en un conjunto de datos de prueba, se utilizaron las métricas AUC, ROC, TPR y FPR, con el objetivo de valorar su desempeño en términos de su capacidad para diferenciar entre clases positivas y negativas. Estas métricas proporcionan una valoración tanto numérica como visual, lo que simplifica el procedimiento para establecer cuál de los modelos está mejor equilibrado en términos de sensibilidad y especificidad.

Modelo	AUC	TPR	FPR
Random Forest	92.1577%	92.5373%	25%
SVM	92.0033%	92.5373%	25.8621%
PAC	91.5015%	92.5373%	23.2759%
MLP	90.0347%	84.3284%	25%
XGBoost	89.4139%	80.5970%	22.4138%

Análisis por modelo

- **Random Forest**

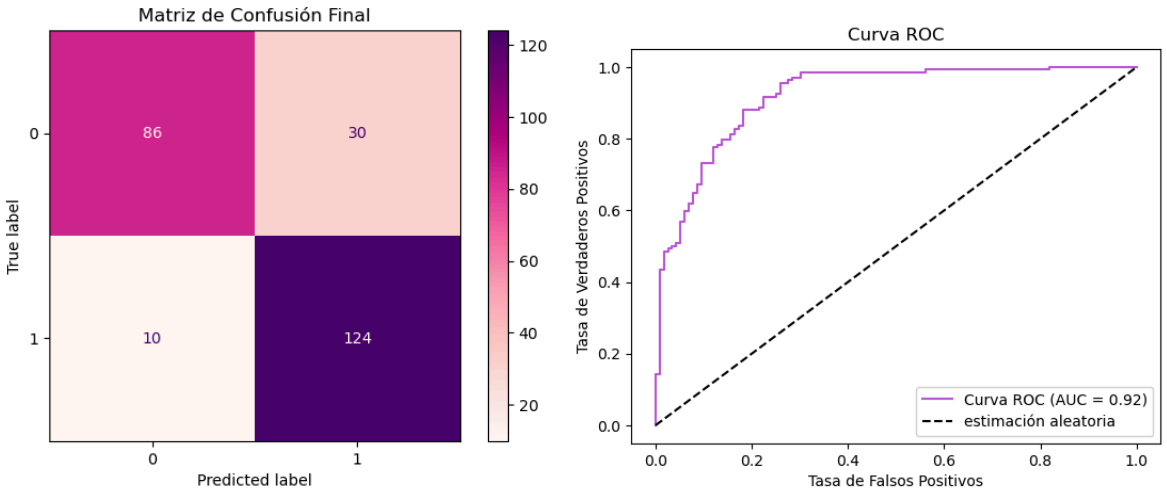
Matriz de Confusión y Gráfica de Curva ROC



El Random Forest presentó el mejor desempeño general, considerando que tiene un AUC de 92.16%, indicando que tiene una gran capacidad de distinción de clases positivas y negativas. Su TPR demuestra que identifica correctamente la mayoría de los casos que presentan señales de anorexia; sin embargo, esto es a costa de una tasa alta de falsos positivos (FPR 25%). A pesar de todo, es el modelo con mejor equilibrio de todos, por lo que se considera un excelente candidato para la clasificación de textos por su precisión y cobertura.

- **SVM**

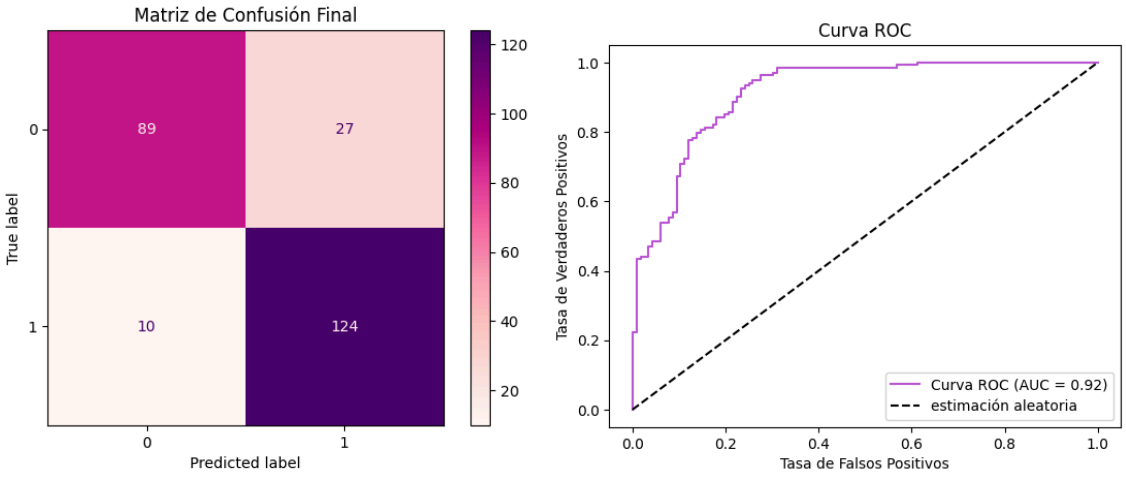
Matriz de Confusión y Gráfica de Curva ROC



Por otro lado, tenemos al SVM, que se acerca en cuanto a desempeño al Random Forest pues tiene un AUC del 92%, que de igual forma demuestra su excelente desempeño clasificatorio. Y si bien empata con la tasa de verdaderos positivos, su FPR resulta mayor con un 25.86%, lo cual puede limitar la confiabilidad de la predicción en casos donde los falsos positivos sean costosos. Sin embargo, este modelo se mantiene como un complemento significativo para la detección de señales de anorexia en redes sociales por su destacable poder discriminativo.

- **PAC**

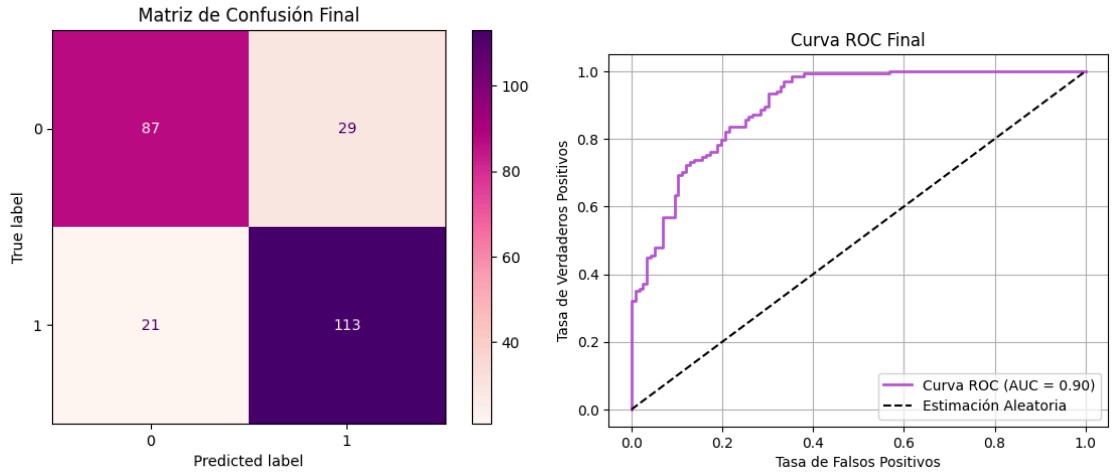
Matriz de Confusión y Gráfica de Curva ROC



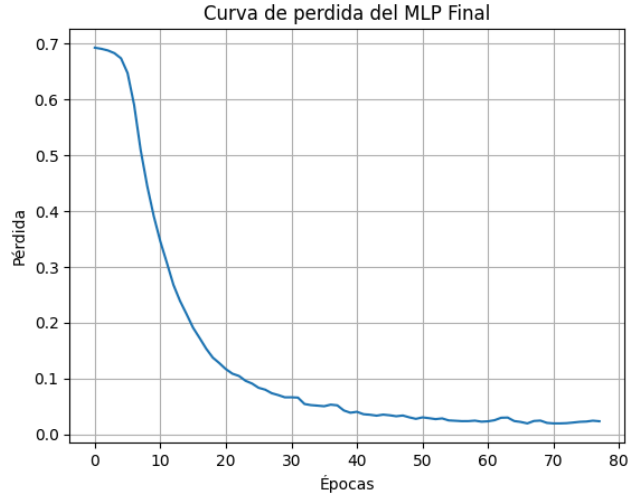
El Passive Aggressive Classifier también demuestra un buen desempeño, pues cuenta con un AUC del 91.50% sin embargo, cuenta con un TPR igual de alto que los dos anteriores descritos, lo que inicialmente lo coloca como un buen clasificador; además, su ventaja más destacable es el FPR más bajo con un 23.28% y, por ende, comete menos errores en la clasificación de casos negativos. Este modelo presenta un balance atractivo en la detección de señales de anorexia y, al ser eficiente computacionalmente, es de los mejores modelos para asegurar una excelente clasificación por su sensibilidad y especificidad.

- **MLP**

Matriz de Confusión y Gráfica de Curva ROC



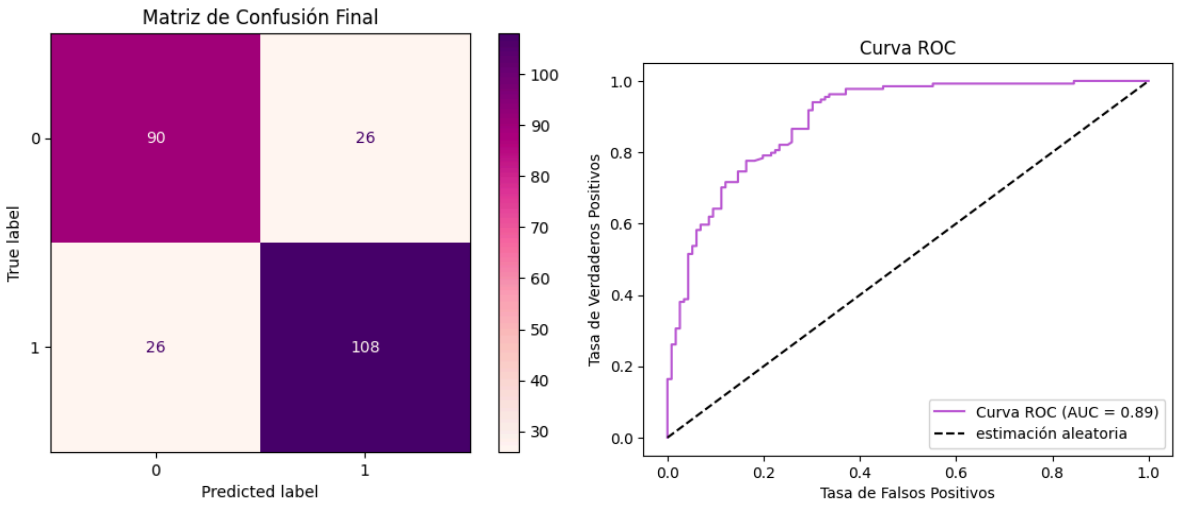
Curva de pérdida



Respecto al modelo de Multilayer Perceptron (MLP), este muestra una reducción más marcada en su rendimiento, con un AUC de 90.03% que aún es aceptable en comparación con los demás modelos. Sin embargo, es en su TPR donde evidencia una disminución considerable en la identificación de casos positivos. Además, el 25% de FPR influye en la calidad de la sensibilidad del modelo, lo que indica que este modelo posee áreas de oportunidad significativas que se deben explotar; sin embargo, su AUC resulta favorable para colaborar con los demás modelos para llegar a una conclusión más robusta.

- **XGBoost**

Matriz de Confusión y Gráfica de Curva ROC



Finalmente, el modelo XGBoost, si bien se destaca por su buen desempeño general, en este caso resultó tener el AUC más bajo con un 89.41% e igual forma su TPR de 80.60%, lo que indica que presenta dificultades en la clasificación de clases positivas. Sin embargo, su ventaja es que cuenta con el TPR % más bajo de todos los modelos (22.41%), que sugiere que le es más fácil identificar casos negativos, por lo que puede ser una herramienta útil para evitar los falsos positivos en la detección de anorexia.

Conclusión

Esta fase de entrenamiento está enfocada en la identificación de señales relevantes en textos con trastornos alimenticios, principalmente anorexia, por medio de modelos tradicionales y un pipeline de procesamiento de datos diseñado rigurosamente. Dados los resultados, vemos claramente la importancia de tener un preprocesamiento sólido, una estrategia de optimización de hiperparámetros adecuada y un marco de evaluación robusto.

Esto se evidencia en la comparación final de los modelos, donde en términos generales se logró un buen equilibrio entre las métricas escogidas, resaltando a Random Forest como el clasificador más destacado, seguido de SVM y PAC, y aunque los modelos MLP y XGBoost poseen áreas de oportunidad que serían relevantes explotar, en conjunto pueden realizar una clasificación excelente entre casos positivos y negativos.

Estos resultados positivos son resultado del preprocesamiento avanzado efectuado previo al entrenamiento, ya que no solo se limpiaron los textos para su adecuada legibilidad, sino que se creó un dataset enriquecido con atributos especiales y beneficiosos para los modelos escogidos, pues al estar divididos de esta forma, facilitan la identificación de patrones en los textos. Estas características clave fueron:

- **Métricas estilísticas**: Estas aportan contexto sobre la densidad y extensión del texto.
- **Variables semánticas y temáticas**: Muestran explícitamente palabras clave relacionadas con conductas alimenticias que pueden servir para identificar indicios de trastornos.
- **Análisis de sentimiento**: Esto añadió características clave que son útiles para detectar contenido sensible.
- **Hashtags frecuentes**: Esta característica ayuda a dar contexto al texto más claramente, pues captura palabras clave específicas del problema.

- **TF-IDF**: Estas características fueron clave, pues condensaron el texto en información numérica útil para que los modelos detectaran patrones eficientemente.

Además, se llevó a cabo una estrategia clave en el entrenamiento, la cual fue la optimización exhaustiva del desempeño con la técnica de GridSearch, lo cual fue favorable para la mayoría de los modelos, pues maximizó su rendimiento y aseguró que los resultados fueran el perfecto balance entre sensibilidad y especificidad. Estas fueron evaluadas correctamente con las métricas adecuadas (AUC-ROC, TPR y FPR), lo cual brinda confiabilidad al proceso de pruebas y al desempeño general de los modelos.

En conclusión, este trabajo demuestra que la calidad del desempeño de los modelos no radica en el algoritmo elegido, sino en un sistema robusto que integra preprocesamiento contextualizado, representación enriquecida de los datos, optimización cuidadosa y evaluación crítica con métricas apropiadas, y que como resultado se obtienen herramientas de detección precisas y confiables en el área donde se apliquen.

Bibliografía

Bobbitt, Z. (2021, 9 septiembre). *What is Considered a Good AUC Score?* Statology. https://www.statology.org/what-is-a-good-auc-score/

Bobbitt, Z. (2021, agosto 9). *How to Interpret a ROC Curve (With Examples)*. Statology. https://www.statology.org/interpret-roc-curve/

Carrascosa, I. P. (2025, 7 enero). *A Complete Guide to Cross-Validation*. Statology. https://www.statology.org/complete-guide-cross-validation/

Google Developers. (s.f.). *Normalización: Curso intensivo de introducción al aprendizaje automático*. Google Developers. https://developers.google.com/machine-learning/crash-course/numerical-data/normalization?hl=es-419

scikit-learn developers. (s.f.). *sklearn.ensemble.RandomForestClassifier*. scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

ScienceDirect. (s.f.). *Receiver Operating Characteristic - an overview | ScienceDirect Topics*. ScienceDirect. https://www.sciencedirect.com/topics/engineering/receiver-operating-characteristic#:~:text=Receiver%20Operating%20Characteristic%20(ROC)%20is%20a%20plot%20between%20TPR%20or,or%20thresholds)%20%5B10%5D.

scikit-learn developers. (s.f.). *sklearn.feature_extraction.text.TfidfVectorizer*. scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html