PREDICCIÓN DE VIRALIDAD DE TWEETS SEGÚN LOS ME GUSTA Y RETWEETS

Marc Pascual Desentre
Alejandro Lobo Millan

Escola Politècnica Superior d'Enginyeria de Vilanova i la Geltrú
Mineria de Dades
Primer Quadrimestre
Curs 2024/2025

Índice:

1- Descripción del dataset original

Fuente del dataset

El dataset fue obtenido de Kaggle, específicamente de la siguiente página: Donald Trump Tweets Dataset. Fue recopilado por un usuario en Reddit y contiene información detallada sobre los tweets de Donald Trump del año 2015 y 2016.

Para importarlo al entorno de trabajo, utilizamos el siguiente código:

```python
%matplotlib inline
import pandas
import matplotlib.pyplot as plt
from pandas.plotting import scatter_matrix
import numpy as np

pandas.set_option('display.max_columns', None)
pandas.set_option('display.expand_frame_repr', False)
pandas.set_option('display.precision', 3)
df = pandas.read_csv('data.csv', sep=',', na_values="")
```

Descripción del problema

El objetivo de este análisis es determinar si un tweet puede considerarse viral o no, utilizando características como el contenido textual del tweet y métricas de interacción (retweets y likes). En concreto, dividiremos el texto de los tweets en palabras y analizaremos qué términos tienden a repetirse en los tweets más populares.

¿Por qué escogimos este problema?

Creemos que este análisis es relevante porque proporciona información valiosa sobre el impacto y la propagación del contenido en redes sociales, en particular para una figura tan polémica como Donald Trump, quien tuvo un papel destacado en las elecciones de Estados Unidos en 2016.
Además, este estudio podría ser útil para campañas futuras, ya que permite identificar patrones en los tweets que generan mayor interacción. Sin embargo, reconocemos que la viralidad también podría depender de otros factores, como el contexto temporal en el que se publicaron ciertos tweets.

Detalles del dataset

Número original de ejemplos:

El dataset contiene 8,716 filas que són los tweets publicados por Donald Trump.

Número original de columnas:

El dataset contiene 11 columnas, que representan diferentes aspectos del contenido y metadatos de los tweets.

Descripción de las columnas:

Date: Fecha en que se publicó el tweet.
Time: Hora en que se publicó el tweet.
Tweet_Text: El contenido textual del tweet.
Type: Tipo de tweet (por ejemplo, original, respuesta, retweet).
Media_Type: Tipo de contenido multimedia adjunto (imagen, video, etc.).
Hashtags: Hashtags utilizados en el tweet.
Tweet_Id: Identificador único del tweet.
Tweet_Url: URL del tweet.
twt_favourites_IS_THIS_LIKE_QUESTION_MARK: Número de favoritos/likes en el tweet.
Retweets: Número de veces que el tweet fue retuiteado.

Análisis general del dataset

Aunque las únicas columnas que finalmente usaremos son Tweet_Text, twt_favourites_IS_THIS_LIKE_QUESTION_MARK (likes) y Retweets, hemos revisado todas las columnas para identificar posibles sesgos o relaciones que puedan influir en nuestro modelo. Nuestro objetivo es asegurarnos de que la viralidad dependa únicamente del texto y que no existan características externas que sesguen los resultados.

Python
```
data.isnull().any()
```

Python
```
Date                                                False
Time                                                False
Tweet_Text                                          False
Type                                                False
Media_Type                                           True
```

```
Hashtags                                          True
Tweet_Id                                         False
Tweet_Url                                        False
twt_favourites_IS_THIS_LIKE_QUESTION_MARK        False
Retweets                                         False
Unnamed: 10                                       True
Unnamed: 11                                       True
dtype: bool
```

Observamos que las columnas Media_Type, Hashtags, Unnamed: 10, y Unnamed: 11 contienen valores faltantes. Sin embargo, estas columnas no son relevantes para nuestro análisis, por lo que serán eliminadas en el preprocesamiento.
También identificamos que las columnas Unnamed: 10 y Unnamed: 11 están completamente vacías, probablemente debido a un error de quien subió el dataset.

Relación entre likes, retweets y el tiempo

Nos interesaba investigar si existía una relación entre los likes, retweets y la proximidad a la fecha de las elecciones de 2016 (8 de noviembre de 2016). Para ello:

Creamos una nueva columna Days_to_Election, que calcula la cantidad de días restantes hasta las elecciones y después generamos un gráfico de dispersión para observar cómo cambian los likes y retweets con respecto al tiempo:

Python

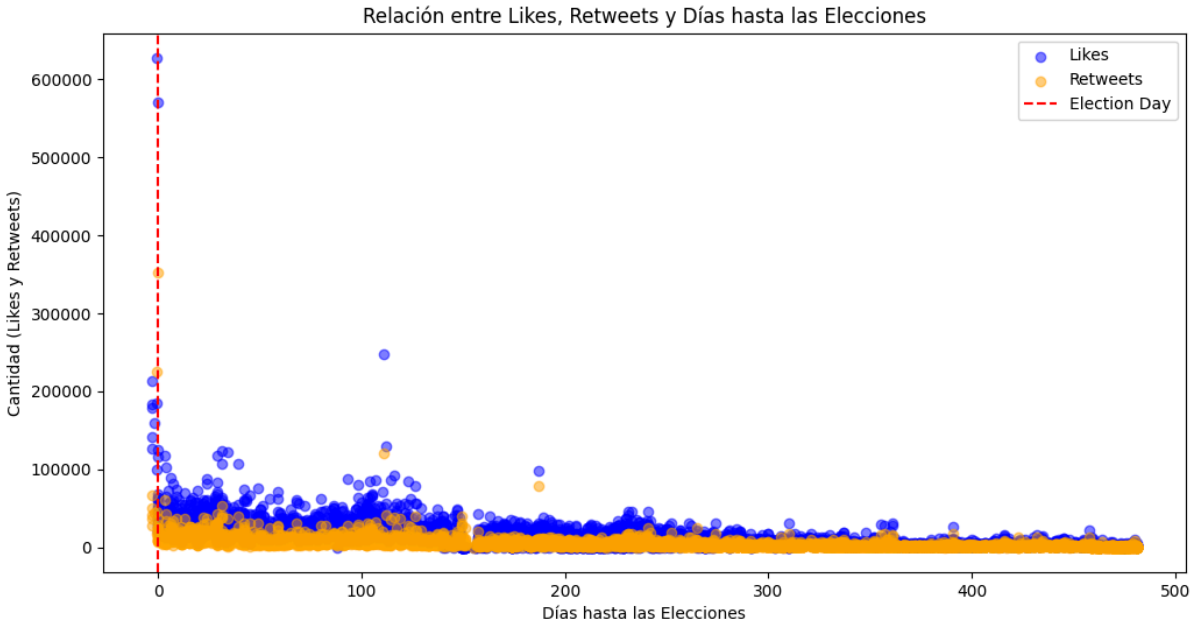
```python
# Convertir la columna Date al formato correcto (año-mes-día)
df['Date'] = pandas.to_datetime(df['Date'], format='%y-%m-%d')

# Calcular la cantidad de días hasta las elecciones
election_date = pandas.to_datetime('2016-11-08')
df['Days_to_Election'] = (election_date - df['Date']).dt.days

# Verificar las primeras filas para confirmar que las fechas y días son
correctos
print(df[['Date', 'Days_to_Election']].head())

# Scatter plot para likes y retweets frente a días hasta las elecciones
plt.figure(figsize=(12, 6))
plt.scatter(df['Days_to_Election'],
df['twt_favourites_IS_THIS_LIKE_QUESTION_MARK'], alpha=0.5, label='Likes',
color='blue')
```

```
plt.scatter(df['Days_to_Election'], df['Retweets'], alpha=0.5,
label='Retweets', color='orange')
plt.axvline(0, color='red', linestyle='--', label='Election Day')
plt.xlabel('Días hasta las Elecciones')
plt.ylabel('Cantidad (Likes y Retweets)')
plt.title('Relación entre Likes, Retweets y Días hasta las Elecciones')
plt.legend()
plt.show()
```



Resultados y observaciones

1. Tendencia cercana a las elecciones:
    - Observamos un incremento significativo en los likes y retweets en los días cercanos a las elecciones (especialmente en los días justo antes del 8 de noviembre de 2016). Esto confirma que el contexto temporal influye en la viralidad de los tweets.
2. Decisión:
    - Aunque reconocemos que este efecto temporal puede sesgar nuestro modelo, decidimos ignorarlo por el momento. Más adelante, en el preprocesamiento, podríamos considerar técnicas para balancear este impacto.

2- Descripción del preprocesamiento de datos

El preprocesamiento de los datos se llevó a cabo para limpiar, simplificar y optimizar el dataset con el fin de facilitar el análisis y la modelización. A continuación, se describen los pasos realizados:

Simplificación de los Datos

- Eliminación de columnas irrelevantes: Se eliminaron columnas que no aportaban información relevante para el análisis:
    - Tweet_Text: Después de limpiar el texto, se utilizó una versión procesada(donde dividimos el texto procesado por palabras).
    - Columnas de metadatos como Tweet_Id, Tweet_Url, Date, Time, Media_Type, Type, Hashtags.
    - Columnas vacías como Unnamed: 10 y Unnamed: 11.
    - Columnas relacionadas con twt_favourites_IS_THIS_LIKE_QUESTION_MARK y Retweets, ya que se usaron para calcular la variable objetivo Viral.

Manejo de Valores Nulos

- Las columnas Media_Type y Hashtags contenían valores nulos, pero se consideraron irrelevantes para el análisis y se eliminaron completamente

Preprocesamiento del texto del tweet.

Para analizar el contenido textual de los tweets, se aplicaron las siguientes técnicas de limpieza y transformación:

1. Eliminación de etiquetas HTML: Se usó una expresión regular para eliminar etiquetas HTML del texto del tweet (cleanhtml).
2. Eliminación de signos de puntuación: Se eliminaron caracteres especiales y puntuación (cleanpunc).
3. Eliminación de stopwords: Se eliminaron palabras comunes del idioma inglés (stopwords) utilizando la lista de NLTK.
4. Lematización/Stemización de palabras: Se redujeron las palabras a su forma base utilizando el SnowballStemmer de NLTK.
5. Filtrado de palabras alfabéticas: Se conservaron solo las palabras alfabéticas (se descartaron números y símbolos).
6. Eliminación de palabras cortas: Se descartaron las palabras con menos de tres caracteres.

Ejemplo del programa de preprocesamiento del tweet:

- Original: "Fantastic day! Met President Obama - first really good meeting, #ThankYou!"
- Procesado: "fantast day met presid obama first realli good meet"

Definicion y calculo del target(objetivo)

Se definió una nueva variable objetivo llamada Viral, basada en el nivel de interacción (Likes y Retweets combinados):

- Los tweets con una interacción total mayor o igual a dos veces la suma de las medianas de likes y retweets (1.05 * (mediana_likes + mediana_retweets)) se clasificaron como mucho.
- Los demás se clasificaron como poco.

Hemos seguido el criterio anterior ja que balancea muy bien nuestro target siendo el siguiente:

```
Python
Viral
poco     3773
mucho    3602
Name: count, dtype: int64
```

Este enfoque binario y balanceado permite más adelante aplicar los modelos de datamining sin problemas

Extracción de Características

Después del preprocesamiento del texto, la columna cleaned_tweet se vectorizó utilizando CountVectorizer:

- El texto se ha transformado en un modelo bag-of-words, donde cada columna representa una palabra única que aparece en algún tweet del dataset. Cada fila corresponde a un tweet, y el valor numérico en cada celda indica cuántas veces aparece esa palabra en ese tweet. Este proceso convierte el texto en una matriz numérica, ignorando el orden de las palabras.
- Para reducir el número de columnas hemos eliminado las palabras que aparecen menos de 5 veces (min_df=5).

Al final hemos acabado con 1477 palabras(columnas) y 7375 tweets(filas)

Dataset despues del preproceso.

Después de vectorizar el texto, el dataset final se dividió en:

X: Contiene los datos vectorizados de los tweets.

y: Contiene la variable objetivo Viral.

Analisis

Como un análisis curioso, identificamos las 10 palabras que más se repiten en el conjunto de datos de tweets. A continuación, presentamos las palabras más comunes junto con la cantidad de veces que aparecen:

```python
Top 10 most frequent words and their counts:
trump: 779 veces
great: 705 veces
thank: 675 veces
hillari: 367 veces
makeamericagreatagain: 347 veces
america: 311 veces
make: 304 veces
get: 295 veces
new: 280 veces
poll: 280 veces
```

Vemos que las palabras que más se repiten están claramente relacionadas con las elecciones presidenciales de Estados Unidos y, en particular, con los temas clave de la campaña de Donald Trump. Entre estas destaca su emblemática frase de campaña “makeamericagreatagain,” que aparece con alta frecuencia y refuerza su mensaje de identidad política. También observamos otras palabras relacionadas, como "Trump," “hillari”, "America," y "poll(sondeo/encuesta)," que reflejan la constante comunicación sobre sí mismo, la nación, y las estrategias electorales. Este análisis destaca cómo los tweets de Trump se centraban en reforzar su marca y mantener la atención en temas específicos que movilizaban a su base de votantes.

3- Criterios de evaluación de modelos de minería de datos

División de los datos de training, validation y test de manera fija

Podemos dividir los datos de nuestro set de manera fija, esta práctica es muy sencilla y si realizamos los análisis pertinentes podemos obtener unos resultados aceptables con un bajo coste computacional.

Python

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


# Ratios de Training, Validación y Test
split_ratios = [(0.7, 0.1), (0.6, 0.1), (0.5, 0.2)]  # Train,
Validation ratios
results = []


for train_ratio, val_ratio in split_ratios:
    # Primer paso: dividir en entrenamiento y temporal
    X_train, X_temp, y_train, y_temp = train_test_split(X, y,
test_size=1-train_ratio, random_state=42)


    # Segundo paso: dividir el conjunto temporal en validación y
prueba
    X_val, X_test, y_val, y_test = train_test_split(
        X_temp, y_temp, test_size=val_ratio/(val_ratio +
(1-train_ratio-val_ratio)), random_state=42
    )


    # Crear y entrenar el modelo
    model = DecisionTreeClassifier()
    model.fit(X_train, y_train)


    # Evaluar el modelo
```

```
    val_accuracy = accuracy_score(y_val, model.predict(X_val))
    test_accuracy = accuracy_score(y_test, model.predict(X_test))


    # Guardar resultados
    results.append({
        'train_ratio': train_ratio,
        'val_ratio': val_ratio,
        'test_ratio': 1-train_ratio-val_ratio,
        'val_accuracy': val_accuracy,
        'test_accuracy': test_accuracy
    })


# Imprimir resultados
for res in results:
    print(f"Train: {res['train_ratio']*100}%, Validation:
{res['val_ratio']*100}%, Test: {res['test_ratio']*100}%")
    print(f"Validation Accuracy: {res['val_accuracy']:.4f}, Test
Accuracy: {res['test_accuracy']:.4f}")
    print()
```

Train: 70.0%, Validation: 10.0%, Test: 20.0%
Validation Accuracy: 0.6807, Test Accuracy: 0.6707

Train: 60.0%, Validation: 10.0%, Test: 30.0%
Validation Accuracy: 0.6786, Test Accuracy: 0.6504

Train: 50.0%, Validation: 20.0%, Test: 30.0%
Validation Accuracy: 0.6655, Test Accuracy: 0.6551

Podemos apreciar que haciendo diferentes pruebas con nuestro dataset obtenemos diferentes resultados y que en este caso la mejor combinación fija que podríamos utilizar para dividir nuestros datos es la segunda donde no tenemos malos resultados de validación y tenemos mejores resultados aún de test.

En el desarrollo de nuestro análisis y evaluación de modelos, adoptamos diferentes procedimientos para garantizar que el conjunto de validación fuera representativo y que los resultados obtenidos fueran consistentes y confiables.

División inicial de los datos

Hemos utilizado un procedimiento de división train-test split, separando el 70% de los datos para el entrenamiento y el 30% para la validación. Esta proporción fue seleccionada debido a que proporcionó los mejores resultados de accuracy durante las pruebas preliminares, equilibrando la cantidad de datos necesarios para el entrenamiento del modelo y la evaluación de su rendimiento. Además, la división fue realizada de manera estratificada (stratified split) para garantizar que las proporciones de las clases en el conjunto de entrenamiento y el de validación fueran representativas del conjunto de datos original.

```python
(X_train, X_test,  y_train, y_test) = train_test_split(X, y2, test_size=.3,
random_state=1)
```

Validación cruzada

Para evaluar el rendimiento de los modelos de manera más robusta, se utilizó la técnica de validación cruzada con 10 pliegues (10-fold cross-validation). Esta técnica divide el conjunto de entrenamiento en 10 partes iguales y entrena el modelo 10 veces, usando cada pliegue a su vez como conjunto de validación y los demás como conjunto de entrenamiento. Este proceso ayuda a reducir la variabilidad del rendimiento del modelo y proporciona una evaluación más confiable. Elegimos 10 pliegues debido a que es un valor comúnmente utilizado que proporciona un buen compromiso entre la robustez de la evaluación y el tiempo de cómputo requerido.

Además, se empleó Grid Search para la optimización de hiperparámetros, específicamente para ajustar el número de vecinos (n_neighbors), el tipo de ponderación de los vecinos (weights), y la métrica de distancia (metric). Grid Search realiza una búsqueda exhaustiva de todas las combinaciones posibles de estos parámetros, y al combinarse con la validación cruzada, nos permite encontrar la mejor configuración para maximizar el rendimiento del modelo.

```python
kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
clf = GridSearchCV(knc, param_grid=params,cv=10,n_jobs=-1)  # If cv is
integer, by default is Stratifyed
clf.fit(X_train, y_train)
```

Métrica de evaluación

La métrica principal utilizada para evaluar el rendimiento de los modelos fue el accuracy debido a su simplicidad y relevancia en problemas donde las clases están balanceadas. Sin embargo, fue necesario evaluar más detalladamente el rendimiento, empleamos métricas complementarias como el F1-score, la precisión y el recall.

Para complementar y enriquecer nuestro análisis, también nos apoyamos en diversas gráficas y matrices gráficas, como las matrices de confusión, las cuales nos ayudaron a visualizar de manera más clara el comportamiento de los modelos en cuanto a la clasificación de las clases. Estas representaciones gráficas facilitaron la interpretación de los resultados y nos permitieron detectar posibles áreas de mejora, como la identificación de falsos positivos o falsos negativos.

En conclusión, los procedimientos seguidos garantizaron que los modelos fueran evaluados de manera rigurosa y representativa, utilizando tanto divisiones iniciales estratificadas como validación cruzada con un número adecuado de divisiones.

```python
print(sklearn.metrics.accuracy_score(y_test, pred))
print(classification_report(y_test, pred))
cm = (confusion_matrix(y_test, pred))
plt.figure(figsize=(6, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Poco",
"Mucho"], yticklabels=["Poco", "Mucho"])
plt.show()
# Reporte de clasificación
report = sklearn.metrics.classification_report(y_test, pred,
output_dict=True)
# Visualización de métricas globales del reporte
metrics = ['precision', 'recall', 'f1-score']
report_df = pandas.DataFrame(report).T
report_df = report_df[metrics]
# Plot de métricas
report_df.iloc[:-3].plot(kind='bar', figsize=(10, 6), colormap='viridis',
edgecolor='black')
plt.title("Precisión, Recall y F1-score por clase")
plt.ylabel("Score")
plt.xlabel("Clases")
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
```

Single Fold Validation vs K-fold Cross Validation

Python

```
%matplotlib inline
import pandas
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import sklearn
import sklearn.datasets as ds
import sklearn.model_selection as cv
import sklearn.neighbors as nb
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import f1_score, classification_report,
confusion_matrix
from sklearn.feature_selection import SelectKBest, f_classif
import warnings
warnings.filterwarnings("ignore")
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Cargar los datos (esto ya lo tienes)
X = pandas.read_csv('X.csv', sep=',', na_values="")
y = pandas.read_csv('y.csv', sep=',', na_values="")

# 1. **Single Fold Validation** - División entre datos de entrenamiento y
test (70-30)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3,
random_state=42)

# Inicializar el clasificador KNN
knn = KNeighborsClassifier(n_neighbors=5)

# Entrenamiento del modelo
knn.fit(X_train, y_train)

# Predicciones
y_pred = knn.predict(X_val)

# Calcular la precisión y otras métricas
accuracy = accuracy_score(y_val, y_pred)
print("\nSingle Fold Validation Results:")
print(f"Accuracy: {accuracy}")

# 2. **K-Fold Cross-Validation** - Usando 5 pliegues
```

```
cv = 10

# Inicializar KNN nuevamente
knn = KNeighborsClassifier(n_neighbors=5)

# Realizar K-Fold Cross-Validation
scores = cross_val_score(knn, X, y, cv=cv, scoring='accuracy')

# Mostrar los resultados de la validación cruzada
print("\nK-Fold Cross-Validation Results:")
print(f"Average accuracy: {scores.mean()}")
```

Single Fold Validation Results:
Accuracy: 0.6037053773158608

K-Fold Cross-Validation Results:
Average accuracy: 0.5919848282607657

4. Métodos machine learning:

4.1. Naive Bayes:

Hipótesis de Independencia de Variables en Naïve Bayes

El modelo Naïve Bayes se basa en la hipótesis de que las variables (en este caso, las palabras representadas en el bag-of-words) son independientes entre sí, dado el valor de la clase. Esta suposición suele ser razonablemente válida en bag-of-words, ya que las palabras individuales tienden a aparecer de manera aislada en muchos dominios. Sin embargo, en un entorno de elecciones, como el que estamos analizando, es probable que exista una mayor dependencia entre palabras debido a la co-ocurrencia de términos clave relacionados con temas específicos o frases como "make america great again".

Para comprobar si las palabras en el bag-of-words tienen relación, calculamos las correlaciones entre las palabras usando una matriz de correlación (np.corrcoef). Establecimos un umbral bajo de 0.5, donde 1 indica una correlación perfecta positiva(siempre juntas) y -1 una correlación perfecta negativa(nunca juntas).

```python
from sklearn.feature_selection import mutual_info_classif
from sklearn.metrics import pairwise_distances
word_correlations = np.corrcoef(X, rowvar=False)
top_correlated_pairs = []
threshold = 0.5
for i in range(word_correlations.shape[0]):
    for j in range(i + 1, word_correlations.shape[1]):
        if abs(word_correlations[i, j]) > threshold:
            top_correlated_pairs.append((i, j, word_correlations[i, j]))
print("Top Correlated Pairs")
print(len(top_correlated_pairs))
```

```python
Top Correlated Pairs
53
```

Tras analizar las correlaciones, encontramos que solo 53 pares de palabras superan este umbral, lo que sugiere que la mayoría de las palabras no tienen una fuerte dependencia entre sí.

¿Hay suficientes datos para estimar probabilidades fiables?

Con un dataset de tamaño 7375 filas y un bag-of-words con 1477 características, podemos afirmar que hay un número razonable de datos para estimar probabilidades. Sin embargo, la calidad de estas estimaciones depende de:

- Distribución de palabras: Si ciertas palabras aparecen muy raramente, las probabilidades asociadas a esas palabras serán menos confiables. Para solucionar esto anteriormente ya hemos eliminado palabras que se dicen poco.
- Balance de clases: Asegurarnos de que las clases (mucho y poco) estén bien representadas para evitar que el modelo sesgue su predicción hacia la clase mayoritaria. En nuestro caso la proporción de casos es de:

  poco 3773
  mucho 3602
  Es decir tenemos el "target" totalmente balanceado.

Ajuste de Parámetros y Evaluación del Modelo
Decisión de utilizar MultinomialNB.

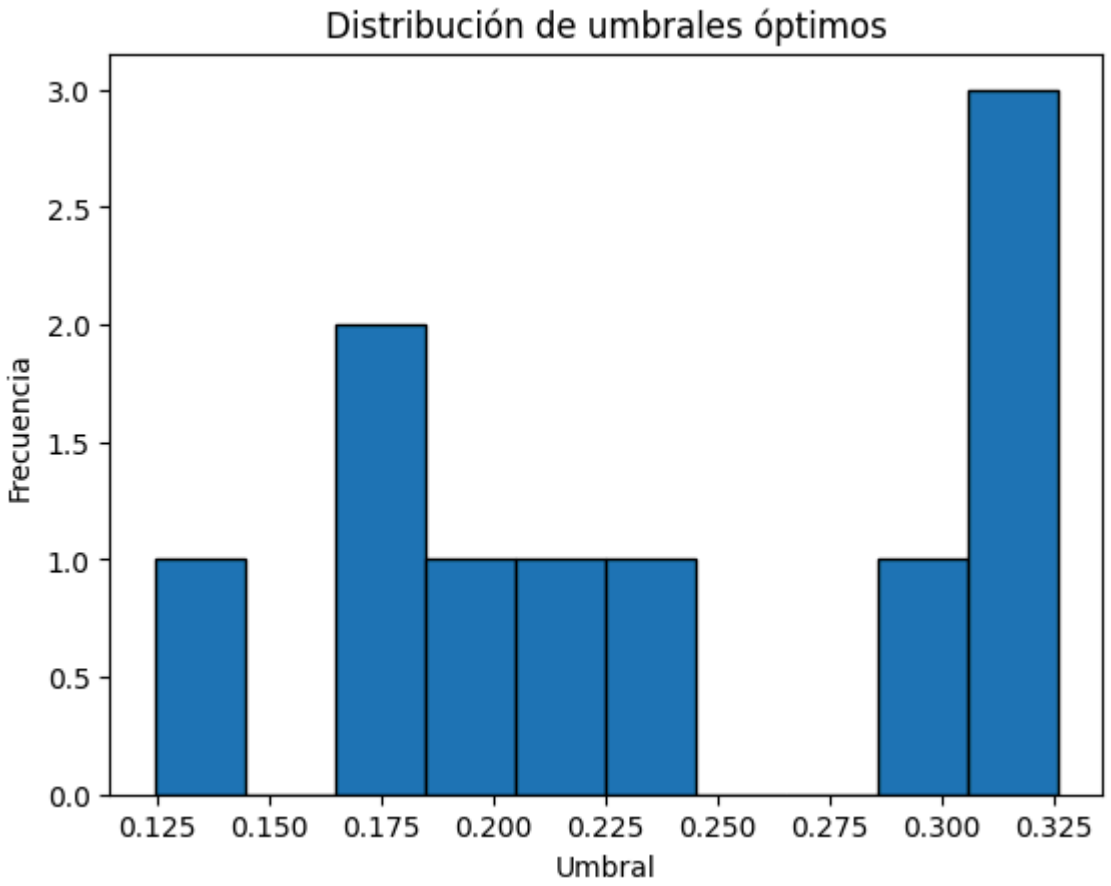
Hemos elegido el modelo Multinomial Naïve Bayes porque es especialmente adecuado para datos representados mediante conteos discretos, como el caso del bag-of-words, donde cada característica corresponde al número de veces que una palabra aparece en un documento. A diferencia de:

- Gaussian Naïve Bayes: Esta variante asume que las características siguen una distribución normal, lo cual no se ajusta bien a nuestro caso.
- Bernoulli Naïve Bayes: Esta variante resulta útil cuando las características se modelan como variables binarias (presencia/ausencia de una palabra), en lugar de contar las veces que aparece. SI en vez de tener cuantas veces se repite la palabra y tenemos la presencia o ausencia de esta quizás hubiese sido un modelo interesante.

Ajuste de Parámetros y Evaluación del Modelo
Optimización del umbral

En vez de emplear un umbral de 0.5 por defecto, se utilizó cross validation (con 10 pliegues) sobre el conjunto de entrenamiento para encontrar el umbral que maximizará el F1-score de la clase minoritaria o el F1 macro. Esto permite ajustar la sensibilidad del modelo, mejorando su capacidad para identificar correctamente la clase "mucho" sin sacrificar demasiado la precisión global.

El valor de umbral obtenido en los 10 pliegues es de 0.2358900931572922. Esto significa que, en promedio, para lograr el mejor equilibrio entre precisión y recall, se debe considerar una instancia como perteneciente a la clase positiva ("mucho") si su probabilidad predicha supera el 23.6%.



En este gráfico vemos que el umbral obtenido en los 10 pliegues no es uniforme, hay una variación entre 0.125 y 0.325. Esta variación indica que:

- Nuestro modelo es sensible a la partición de los datos. Es decir que depende de cómo se hace la partición de los datos puede cambiar significativamente el umbral.
- Nuestro modelo tiende a un umbral más bajo que 0.5. Lo que indica que es mejor ser más flexible al asignar una instancia a la clase "mucho"

Métricas de Evaluación:

Python

```
              precision    recall  f1-score   support

  0.0(mucho)       0.50      0.63      1066
  1.0(poco)        0.66      0.92      0.77      1147

    accuracy                           0.72      2213
   macro avg       0.76      0.71      0.70      2213
weighted avg       0.75      0.72      0.70      2213
```

Las métricas (Accuracy, Precision, Recall y F1-score) reflejan un mejor equilibrio entre las clases con el umbral optimizado. Se obtuvo:

- Accuracy ~0.72 (72%):El modelo clasifica correctamente alrededor del 72% de las instancias en el conjunto de prueba. Esta métrica global considera ambas clases por igual, sin distinguir entre errores en la clase mayoritaria o minoritaria.

  Clase "mucho" (etiquetada como 0.0):

- Precisión (precision): 0.85
  De todas las predicciones que el modelo hizo como "mucho", el 85% efectivamente pertenecían a esa clase. Esto indica que cuando el modelo predice "mucho" suele estar muy seguro.
- Recall: 0.50
  De todos los ejemplos que realmente eran "mucho", solo el 50% fueron detectados correctamente por el modelo. Aunque la precisión es alta, el modelo aún pierde la mitad de los casos reales de "mucho".
- F1-score: 0.63
  El F1-score, que armoniza precisión y recall, es 0.63. Esto refleja un desempeño aceptable, pero con margen de mejora en el recall.

  Clase "poco" (etiquetada como 1.0):

- Precisión: 0.66
  De todas las predicciones que el modelo hizo como "poco", el 66% eran correctas. Es una precisión menor que en "mucho", pero sigue siendo razonable.
- Recall: 0.92
  De todos los ejemplos que realmente eran "poco", el 92% fueron detectados correctamente. Esto indica que el modelo es muy efectivo en no perder casos de "poco".
- F1-score: 0.77
  El F1-score es alto, mostrando que el balance entre precisión y recall es bastante bueno para esta clase.

Métricas de Evaluación: ejecutado sin balancear el umbral.

```
Python
0.7144148215092635
              precision    recall  f1-score   support

         0.0       0.72      0.67      0.69      1066
         1.0       0.71      0.76      0.73      1147

    accuracy                           0.71      2213
   macro avg       0.71      0.71      0.71      2213
weighted avg       0.71      0.71      0.71      2213
```

En resumen, sin el ajuste del umbral, el modelo mantiene un desempeño razonablemente bueno, con una accuracy cercana al 71% y métricas equilibradas entre

ambas clases. No obstante, el ajuste del umbral encontrado previamente logra maximizar ligeramente el F1-score, sobre todo en la clase minoritaria, lo que sugiere que la optimización del umbral es una estrategia valiosa para mejorar el balance entre precisión y recall, incluso si el cambio en la métrica global (accuracy) no resulta espectacular. Esto confirma que la selección del umbral puede marcar la diferencia cuando se busca un rendimiento más justo entre las distintas clases.

Intervalo de confianza

```python
Intervalo de Confianza al 95%: (0.696, 0.733)
```

El intervalo de confianza al 95% para la accuracy del modelo Naïve Bayes fue de (0.696,0.733)(0.696, 0.733)(0.696,0.733). Esto significa que estamos 95% seguros de que la verdadera accuracy del modelo en datos similares se encuentra en este rango. Este intervalo refleja una variabilidad pequeña y un rendimiento estable del modelo, con una accuracy observada de 71.4%. Además, permite comparar su rendimiento con otros modelos de forma estadísticamente sólida.

Conclusión:

En general, el modelo Multinomial Naïve Bayes logra un desempeño decente con los datos y las características empleadas. La mayoría de las palabras actúa como si fueran independientes entre sí, y disponer de una base equilibrada en las clases “mucho” y “poco” facilita estimar probabilidades de forma más confiable para ambas.

La optimización del umbral no cambia radicalmente la precisión global (accuracy), pero sí mejora el equilibrio entre las métricas, en especial el F1-score de la clase minoritaria. Esto sugiere que ajustar el umbral es una estrategia útil para obtener un rendimiento más justo entre las distintas clases. En definitiva, aunque el cambio no sea espectacular, resulta evidente que la selección del umbral puede marcar una diferencia positiva en el equilibrio entre precisión y recall.

4.2. KNN:

Descripción del procedimiento para elegir el mejor valor de k

Para seleccionar el mejor número de vecinos (k) para el modelo K-NN, seguimos estos pasos:

1. Prueba de diferentes valores de k:
   Probamos valores impares de k (para evitar empates) en el rango de 1 a 30.
   Usamos validación cruzada con 10 particiones estratificadas para medir el rendimiento promedio del modelo con cada valor.
   Representamos los resultados en un gráfico donde comparamos las tasas de accuracy para distintos valores de k, tanto con ponderación uniforme (weights='uniform') como con ponderación por distancia (weights='distance').
2. Resultados iniciales:
   Observamos que los mejores resultados de accuracy se obtenían con la ponderación por distancia, donde los vecinos más cercanos tienen un mayor peso. Este método mejoró la capacidad del modelo para clasificar correctamente los datos.
3. Optimización con GridSearch:
   Para ajustar el modelo al máximo, usamos GridSearchCV explorando los valores de:
   - Número de vecinos (n_neighbors) en el rango de 1 a 30.
   - Ponderación (weights), probando 'uniform' y 'distance'.
4. Como resultado, identificamos que el mejor valor para k fue 11 y que el esquema de ponderación por distancia proporcionó el mejor rendimiento.

Ajuste de otros parámetros: Métrica de distancia

El algoritmo K-NN usa distancias para identificar los vecinos más cercanos. En este caso, utilizamos la métrica de distancia euclidiana (por defecto), que es adecuada para datos escalados.

Aunque no exploramos métricas alternativas como la distancia Manhattan, nos aseguramos de escalar los datos previamente con StandardScaler. Esto garantiza que todas las características tengan la misma influencia en los cálculos de distancia, evitando que una variable con mayor rango domine el modelo.

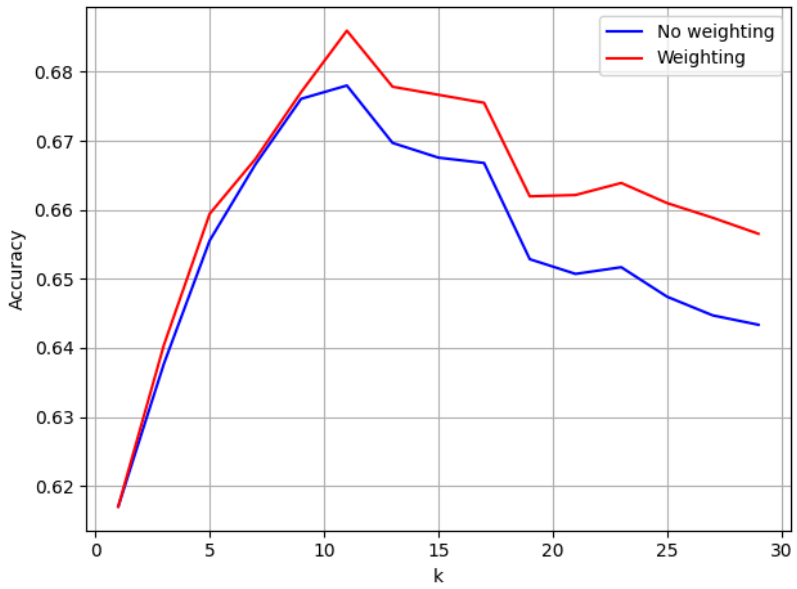
Eliminación de características irrelevantes

El algoritmo K-NN es especialmente sensible a características irrelevantes, ya que estas pueden "desviar" el cálculo de las distancias entre puntos. Para abordar este problema, aplicamos un proceso de selección de características usando SelectKBest con el test ANOVA F (f_classif).

1. Pruebas realizadas:
   Probamos seleccionando diferentes cantidades de características relevantes (por ejemplo, 10, 20, 200), evaluando cómo afectaban al rendimiento del modelo. Descubrimos que seleccionar las 200 características más importantes mejoró notablemente los resultados.
2. Impacto en el rendimiento:
   Esta selección ayudó a eliminar el ruido en los datos y a centrarnos en las variables que realmente aportan información útil para la clasificación. Esto no solo mejoró la precisión, sino que también redujo el tiempo de entrenamiento del modelo.

Gráfico con la variación de k

Creamos un gráfico donde se muestra cómo varía el accuracy del modelo en función del número de vecinos (k). La línea azul representa los resultados sin ponderación por distancia (weights='uniform'), mientras que la línea roja muestra los resultados con ponderación por distancia (weights='distance').



El gráfico evidencia que el uso de la ponderación por distancia y un valor óptimo de k mejoran significativamente el rendimiento del modelo.

Resultado del modelo y conclusión

```
Python
precision    recall  f1-score   support

       mucho       0.74      0.52      0.61      1066
        poco       0.65      0.83      0.73      1147

    accuracy                           0.68      2213
   macro avg       0.70      0.68      0.67      2213
weighted avg       0.69      0.68      0.67      2213
```

El análisis del modelo K-NN muestra que:

- La elección de k y el esquema de ponderación son cruciales para obtener un buen rendimiento.
- El uso de ponderación por distancia mejora significativamente los resultados, especialmente para valores de k moderados.
- La selección de características con SelectKBest optimiza tanto la precisión como el tiempo de entrenamiento.
- Aunque el modelo tiene un accuracy aceptable (68%), la clase "mucho" sigue siendo más difícil de clasificar correctamente
-

Intervalo de Confianza

```
Python
Interval 95% confidence: 0.681 +/- 0.019

(0.6620158849157826, 0.7008399668691249)
```

El intervalo de confianza para la precisión del modelo en el conjunto de prueba fue calculado al 95%. Esto da una idea de la variabilidad esperada en la precisión del modelo en diferentes ejecuciones con datos similares:
Precisión obtenida: 68.1%
Intervalo de confianza: 68.1% ± 1.9% (es decir, entre 66.2% y 70.1%).

El valor del intervalo es relativamente estrecho, lo que indica una estabilidad razonable en el desempeño del modelo.

4.3. Decision Trees:

Parámetros utilizados

Para entrenar el modelo, se utilizó un árbol de decisión con los siguientes parámetros principales:

- Criterio de división: Entropy. Este criterio mide la ganancia de información en cada división, lo que permite elegir los nodos que maximizan la separación entre clases (tweets virales y no virales).
- Min Impurity Decrease: 0.001. Este parámetro establece el mínimo nivel de reducción de la impureza necesario para continuar dividiendo un nodo, lo que ayuda a limitar la complejidad del árbol y evitar divisiones irrelevantes.

Interpretación del Árbol de Decisión

El árbol generado analiza las palabras presentes en los tweets para predecir si serán virales o no. Las características principales utilizadas en los nodos superiores son palabras o combinaciones de palabras que tienen un impacto significativo en la viralidad. Por ejemplo:

1. En los niveles superiores, se observan divisiones basadas en palabras que tienen alta correlación con la viralidad, como "urgente", "gratis" o "promoción". Estas palabras suelen estar relacionadas con mensajes que generan mayor interacción.
2. En los niveles inferiores, se encuentran reglas más específicas, como la combinación de palabras que refuerzan o contradicen la viralidad, por ejemplo: "Si el tweet contiene 'descuento' pero no contiene 'gracias', entonces se clasifica como no viral."

Ejemplo de Reglas Relevantes

Algunas de las reglas más destacadas del árbol incluyen:

- "Si un tweet contiene la palabra 'hillari', es probable que sea viral."
- "Si un tweet no contiene 'hillari' pero incluye 'crook', entonces probablemente no sea viral."
- "Si un tweet contiene 'hillari', no contiene 'crook' i no contiene 'tie' entonces es clasificado como muy viral."

Estas reglas permiten interpretar cómo el modelo toma decisiones basadas en el contenido de los tweets y ofrecen un nivel de interpretabilidad útil para entender el comportamiento del modelo.

Distribución de ejemplos en las hojas

En las hojas del árbol (nodos finales), se analizó cómo se mezclan los ejemplos de tweets virales (+) y no virales (–):

Muchas hojas contienen únicamente ejemplos de una clase (por ejemplo, todos virales). Esto indica que las reglas en esas ramas son altamente confiables. Algunas hojas tienen una mezcla de ejemplos virales y no virales. En estos casos, el modelo tiene menos confianza en la predicción, ya que las características utilizadas para clasificar esos tweets no logran separar las dos clases de manera efectiva.

Fiabilidad del modelo

- Se observó una precisión general del 70%, lo que indica que el árbol clasifica correctamente bastantes de los tweets. Sin embargo, la complejidad del árbol sugiere que puede haber problemas de sobreajuste en algunas ramas.

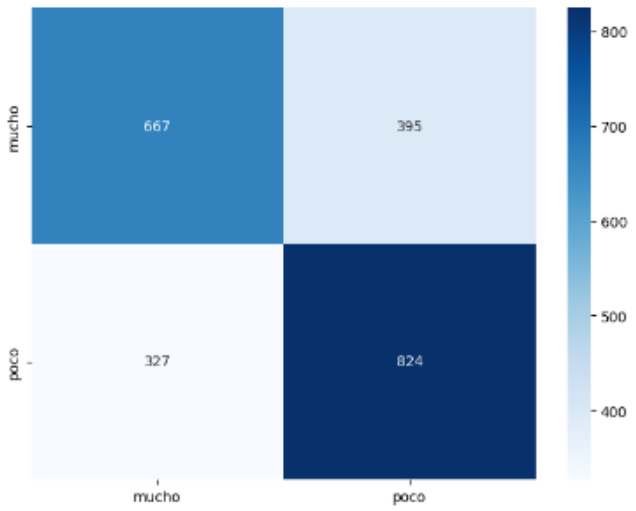
Ajuste de Parámetros y Mejora en el Desempeño

En la experimentación con el árbol de decisión, se observó una mejora significativa en la capacidad de detección de la viralidad al reducir el parámetro min_impurity_decrease de 0.001 a 0.0001. Este parámetro controla la cantidad mínima de reducción de la impureza que debe ocurrir para que un nodo continúe dividiéndose. Es decir, un valor más pequeño permite que el árbol realice divisiones más finas y detalladas en los nodos.

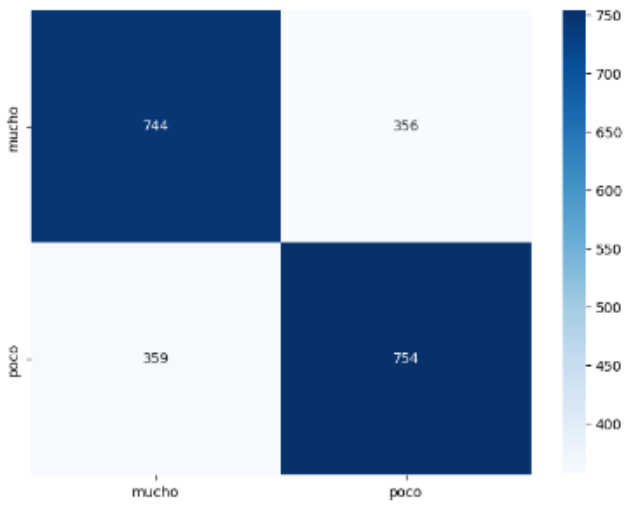
Impacto de la Reducción de la Impureza en el Modelo

Al reducir la impureza mínima, el árbol permite realizar más divisiones, como resultado, el modelo pudo diferenciar mejor entre los tweets virales y no virales, lo que se reflejó en una mejora en las métricas de evaluación, como la precisión y la capacidad de detectar correctamente los tweets virales.

Impurity = 0.001



Impurity = 0.0001



Intervalo de Confianza

```python
Confidence interval:  (0.6570475254345632, 0.6961392307092682)
```

El intervalo de confianza al 95% para la precisión del modelo de Árbol de Decisión es (0.657,0.696)(0.657, 0.696)(0.657,0.696). Esto significa que, con un nivel de confianza del 95%, la precisión real del modelo en un conjunto de datos similar probablemente se encuentre dentro de este rango. La precisión observada en el conjunto de prueba se encuentra en el centro de este intervalo, lo que indica que el modelo tiene un rendimiento estable, aunque con un margen de incertidumbre debido a la variabilidad inherente en los datos. Este intervalo también permite evaluar la fiabilidad del modelo al compararlo con otros enfoques.

4.4. Support Vector Machines:

Elección del Kernel y Parámetros

El uso de Support Vector Machines (SVM) en problemas de clasificación requiere tomar decisiones importantes respecto al kernel y sus parámetros (como C o gamma). Estas decisiones afectan tanto el desempeño como el tiempo de cómputo:

1. Kernel Lineal:
   Es el kernel más simple y es adecuado para datos de alta dimensionalidad como bag-of-words, ya que realiza una separación lineal en el espacio de características. Es más eficiente en términos computacionales y suele ser el punto de partida al trabajar con representaciones de texto.
2. Kernel Polinomial y RBF:
   Los kernels polinomial y RBF son más complejos y permiten capturar relaciones no lineales entre características. Sin embargo, este poder adicional conlleva un mayor costo computacional y riesgo de sobreajuste si los parámetros no se ajustan cuidadosamente. Ambos fueron evaluados, pero resultaron ser más lentos y menos efectivos en este caso.

Por esto, optamos por simplificar considerablemente el problema, ajustando las características y parámetros para ejecutar todos los kernels de manera eficiente**.**

Métodos para Acelerar el Entrenamiento

El bag-of-words inicial contenía 1477 características, lo que hacía que el entrenamiento fuera extremadamente lento. Para mitigar esta situación, implementamos varias estrategias:

Reducción del número de características:
Aplicamos un umbral más restrictivo, quedándonos solo con palabras que aparecen al menos 20 veces en el corpus, lo que redujo el número de características a 516. Esto simplificó el problema sin sacrificar demasiada información relevante.

Paralelización:
Utilizamos n_jobs=-1 en el GridSearchCV para ejecutar evaluaciones en paralelo en todos los núcleos de la CPU, reduciendo drásticamente el tiempo necesario para la búsqueda de hiperparámetros.

Menos pliegues en la validación cruzada:
Se disminuyó el número de particiones en la validación cruzada de cv=10 a cv=5, reduciendo la cantidad total de evaluaciones a la mitad. Aunque esto puede disminuir ligeramente la estabilidad de las estimaciones, es un compromiso razonable para ganar eficiencia.

Rango más acotado para C:
Limitamos el rango de valores de C a 5 valores (Cs = np.logspace(-3, 5, num=5, base=10.0)) en lugar de explorar decenas de valores. Esto permitió explorar opciones representativas sin prolongar innecesariamente el tiempo de búsqueda.

Resultados de los Kernels

Kernel Lineal

- Mejor parámetro C encontrado: 10.0
- Exactitud en el conjunto de test: 64.04%
- Número de soportes: 152 (73.78% de los datos de entrenamiento)
- Intervalo de confianza en validación cruzada para el mejor C:
- Precisión promedio: 58.28%
- Intervalo de confianza (95%): [50.86%, 65.71%]

Kernel Polinomial

- Mejor combinación de parámetros: C = 10000.0, Grado = 3
- Exactitud en el conjunto de test: 58.43%
- Número de soportes: 137 (66.50% de los datos de entrenamiento)
- Intervalo de confianza en validación cruzada para el mejor C:
- Precisión promedio: 55.34%
- Intervalo de confianza (95%): [54.49%, 56.19%]

Kernel RBF (Radial Basis Function)

- Mejor combinación de parámetros: C = 10000.0, Gamma = 1.0
- Exactitud en el conjunto de test: 61.80%
- Número de soportes: 178 (86.41% de los datos de entrenamiento)
- Intervalo de confianza en validación cruzada para el mejor C y Gamma:
- Precisión promedio: 52.92%
- Intervalo de confianza (95%): [52.05%, 53.78%]

Interpretación del Número de Vectores de Soporte

Kernel Lineal:
El kernel lineal presentó un número relativamente alto de vectores de soporte (73.78%), lo cual indica que las clases no están perfectamente separadas en el espacio de características. Esto es común en problemas de texto con alta dimensionalidad.

Kernel Polinomial:
El número de vectores de soporte fue menor que en el kernel lineal (66.50%), lo que podría indicar que el modelo es más eficiente en capturar relaciones no lineales. Sin embargo, su desempeño fue peor en el conjunto de test, probablemente debido a un sobreajuste.

Kernel RBF:
Este kernel tuvo el mayor número de vectores de soporte (86.41%), lo que refleja su flexibilidad y complejidad. Aunque mejoró respecto al polinomial, no logró superar al kernel lineal.

4.5. Meta-learning algorithms:

En este análisis se han comparado varios algoritmos de meta-aprendizaje, específicamente Majority Voting, Bagging, Random Forest y AdaBoost, con el objetivo de evaluar su rendimiento en la clasificación de los datos. A continuación, se explica el procedimiento realizado y los resultados obtenidos.

Procedimiento

1. Majority Voting (Hard y Soft): Se utilizó el VotingClassifier para combinar múltiples modelos de clasificación, en este caso, Naive Bayes, K-Nearest Neighbors (KNN) y Decision Tree. Se evaluaron dos enfoques de votación:
    - Votación Hard: Cada modelo vota por una clase y la clase con más votos es seleccionada como la predicción final.
    - Votación Soft: Similar a la votación hard, pero en lugar de contar los votos, se ponderan según las probabilidades de cada modelo. Con las ponderaciones que hicimos conseguimos obtener los mismos resultados que con hard.

2. Bagging: El BaggingClassifier se utilizó para mejorar el rendimiento del modelo base (un DecisionTreeClassifier) mediante la combinación de múltiples instancias de este modelo entrenados en diferentes subconjuntos de los datos. Se probaron varios valores para el parámetro n_estimators (número de clasificadores en el conjunto) y se evaluó el rendimiento utilizando validación cruzada. También se ajustó el parámetro max_features, que controla la cantidad de características seleccionadas para cada clasificador individual, y se comprobó cómo afectaba esto a la precisión del modelo.

3. Random Forest: RandomForestClassifier es un algoritmo similar al bagging, pero con la particularidad de que selecciona características aleatorias en cada árbol de decisión, lo que agrega más diversidad entre los clasificadores. Se probó con diferentes valores de n_estimators y se comparó su rendimiento con el de los otros algoritmos de meta-aprendizaje.

4. AdaBoost: AdaBoostClassifier es un algoritmo de boosting que ajusta el peso de los clasificadores en función de su rendimiento. Los clasificadores más débiles reciben mayor peso, y el algoritmo intenta corregir los errores de las iteraciones anteriores. Se probó con diferentes valores de n_estimators para analizar cómo el número de clasificadores afecta la precisión del modelo.

Resultados

- Majority Voting (Hard y Soft): En general, la votación hard y soft proporcionaron un rendimiento decente, pero no destacaron como las mejores opciones.

- Bagging: Este método, al combinar múltiples clasificadores, mostró un rendimiento robusto, especialmente cuando se aumentó el número de estimadores. Las evaluaciones indicaron que con más estimadores, el modelo se volvió ligeramente más preciso, y se demostró que el bagging es una estrategia eficaz para reducir el sobreajuste y mejorar la estabilidad del modelo.

- Random Forest: Este algoritmo, al igual que el bagging, mostró un buen rendimiento, y de hecho, fue el que presentó los mejores resultados en términos de precisión. La combinación de múltiples árboles de decisión con selección aleatoria de características generó un modelo más preciso y menos propenso a sobreajustar.

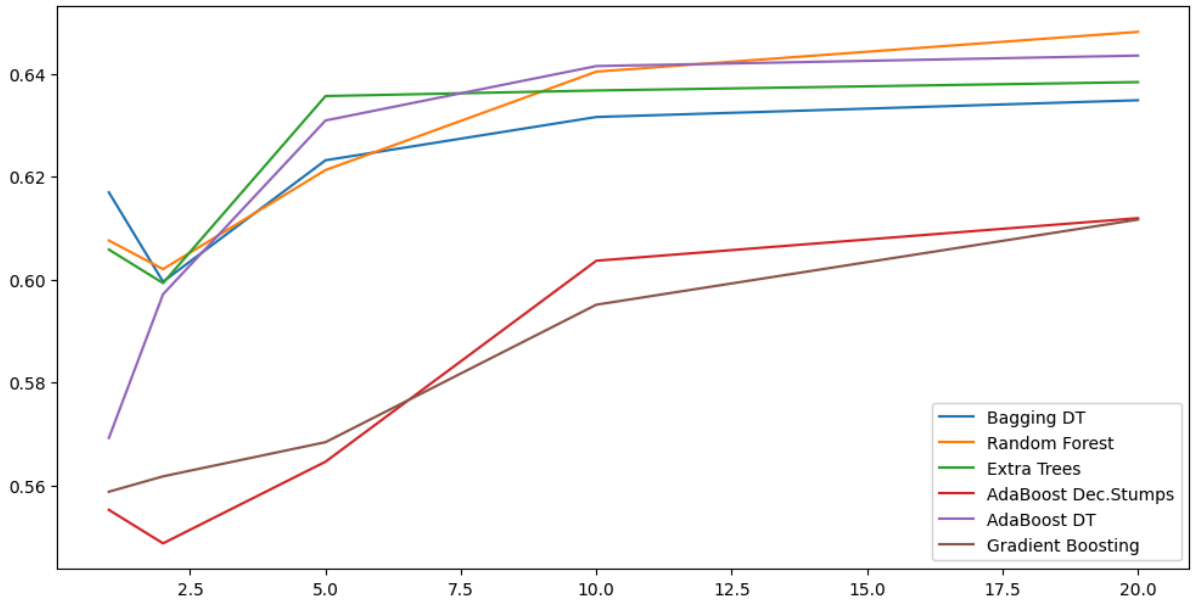
- AdaBoost: A pesar de ser un algoritmo de boosting poderoso, AdaBoost no logró el mismo nivel de rendimiento que el bagging y el random forest en este caso específico. Los resultados mostraron que con un número bajo de estimadores, AdaBoost tenía dificultades para obtener mejor precisión, y aunque se mejoró con más estimadores, seguía siendo menos preciso que bagging y random forest. Esto fue diferente a la hora de analizar AdaBoost DT que obtuvo mejores resultados que Bagging y que extra trees.

Conclusiones

De acuerdo con los resultados obtenidos, Random Forest resultó ser el mejor algoritmo, seguido de AdaBoost DT i Bagging, que también mostraron un buen rendimiento. Ambos superaron a AdaBoost Dec o AdaBoost Boosting , que tuvieron un rendimiento relativamente inferior.

Es posible que la combinación de árboles de decisión con selección aleatoria de características en Random Forest y el enfoque de bagging para reducir la varianza sean los factores clave detrás de su superioridad. En comparación,otros no fueron capaces de aprovechar tan bien el conjunto de datos en este escenario particular.

Aquí podemos ver una imagen de la comparativa en forma de grafica de rendimiento de los diferentes métodos utilizados:



Intervalo de confianza(randomForest 20 estimators)

Al evaluar el Random Forest con 20 estimadores, se calculó el intervalo de confianza del 95% para la precisión del modelo:

```
Python

(20, np.float64(0.6470508474576271), 0.6361441728721854, 0.6579575220430688)]
```

Accuracy Promedio: 0.647
Intervalo de Confianza (95%): (0.636, 0.658)
Esto indica que, con alta probabilidad, el rendimiento del modelo se encuentra en este rango al aplicarse a conjuntos de datos similares.

5. Comparaciones y conclusiones:

Cross Validation

Single Fold Validation: Has obtenido una precisión de 60.37%, lo que indica que el modelo tiene un rendimiento moderado en este único conjunto de datos. Este resultado, aunque parece positivo, puede ser optimista ya que solo se evalúa un subconjunto de los datos, lo cual no garantiza que el modelo generalice bien a nuevos datos.

K-Fold Cross-Validation: La precisión promedio es 59.20%. Aunque este resultado es ligeramente inferior al del Single Fold Validation, ofrece una evaluación más confiable porque se basa en varios subconjuntos, lo que permite comprobar cómo se comporta el modelo en diferentes situaciones. Esto puede indicar que el modelo no generaliza tan bien cuando se expone a diferentes particiones de los datos.

Cross Validation	Accuracy
Single-Fold CV	60.37%
K-Fold CV	59.20%

Learning Methods

Naïve Bayes es un método con una precisión del 72%, que funciona bien en escenarios donde las variables son independientes y siguen una distribución probabilística. Este método suele ser rápido y eficiente para tareas de clasificación, especialmente cuando los datos tienen una distribución condicional clara. Sin embargo, puede tener dificultades cuando las relaciones entre las variables son más complejas, ya que hace la suposición de independencia entre ellas, lo que puede no ser adecuado en ciertos casos.

KNN, con una precisión del 68%, es un método basado en la similitud entre los datos. Aunque es sencillo de implementar, su eficiencia se ve afectada cuando el conjunto de datos es grande o cuando las dimensiones son altas. Además, puede ser lento en la predicción, ya que requiere calcular las distancias entre las muestras para cada predicción, lo que lo hace menos adecuado para datasets grandes.

Decision Tree tiene una precisión del 67%. Este método es interpretativo y fácil de entender, lo que lo hace útil para problemas donde es importante comprender cómo se toman las decisiones. Sin embargo, es susceptible al sobreajuste, especialmente cuando los datos tienen muchas variaciones. Es importante equilibrar su profundidad para evitar el overfitting y garantizar que el modelo no sea demasiado específico para los datos de entrenamiento.

SVM (Support Vector Machine) obtiene una precisión del 58%. Aunque este modelo es muy potente para tareas de clasificación en problemas lineales y no lineales, su

rendimiento puede verse afectado en datasets con mucho ruido o cuando el conjunto de datos es muy grande. Su complejidad computacional también puede ser un inconveniente, especialmente en problemas a gran escala.

Meta-learning (Random Forest) alcanza una precisión del 71%. Esta técnica combina varios árboles de decisión para mejorar la estabilidad y reducir la variabilidad en las predicciones. Aunque es un enfoque robusto y a menudo más preciso que los árboles individuales, puede ser más complejo y costoso en términos de recursos computacionales. Sin embargo, su capacidad para manejar grandes volúmenes de datos y mejorar la precisión lo convierte en una opción muy valiosa.

Model/Classificador	Accuracy
Naïve Bayes	72%
KNN	68%
Decision Tree	67%
SVM	58%
Meta-learning (Random Forest)	71%

Naïve Bayes presenta un buen rendimiento general con una precisión del 72% para la clase "Mucho" y del 71% para la clase "Poco", con un recall más alto en la clase "Poco" (0.76), lo que indica que el modelo es eficaz en identificar esta clase. Sin embargo, su precisión es algo más baja en la clase "Mucho" (0.72), lo que podría sugerir cierta dificultad en la generalización de estas muestras, especialmente en casos donde las características no son totalmente independientes. En resumen, Naïve Bayes es eficaz para identificar la clase "Poco", pero muestra un rendimiento moderado con la clase "Mucho".

KNN muestra una precisión más alta para la clase "Mucho" (0.74), pero el recall de la clase "Poco" es significativamente más alto (0.83), lo que implica que el modelo tiene una mejor capacidad para identificar las muestras menos representadas, especialmente en un conjunto de datos desequilibrado. La baja precisión en la clase "Mucho" (0.65) sugiere que el modelo tiene dificultades para identificar correctamente las muestras de la clase más común. Aunque tiene un buen recall para "Poco", el modelo necesita mejorar la precisión en "Mucho".

Decision Tree presenta un rendimiento equilibrado con una precisión y recall muy similares entre las dos clases, con un f1-score de 0.68 en ambas. Esto sugiere que el modelo es consistente y no se ve influenciado por el desequilibrio entre las clases. Puede identificar las relaciones entre las características de manera eficaz, pero sin

mostrar un rendimiento superior en ninguna clase específica. Este modelo es robusto, pero no se destaca en comparación con otros métodos.

SVM presenta un rendimiento algo inferior, con una precisión de 0.68 para la clase "Mucho" y un recall de 0.40, lo que indica que tiene dificultades para identificar correctamente las muestras de la clase "Mucho". Sin embargo, su recall en la clase "Poco" es más alto (0.79), lo que sugiere que tiene una mejor capacidad para identificar las muestras menos frecuentes. La baja precisión y f1-score para la clase "Mucho" (0.51) indican que el modelo necesita optimizarse para mejorar su capacidad de generalización en esta clase.

Meta-learning con Random Forest obtiene una buena precisión para ambas clases ("Mucho" y "Poco"), con un recall algo superior en la clase "Poco". El modelo presenta un buen equilibrio entre ambas métricas, con un promedio de accuracy del 71%. Su capacidad para gestionar mejor las variaciones entre las clases, especialmente cuando se combinan varias técnicas de aprendizaje, lo convierte en un modelo robusto y eficiente para tareas de clasificación con desequilibrio entre las clases.

En general, los mejores métodos encontrados son Naïve Bayes por su capacidad de generalización en ambas clases y el Meta-learning con Random Forest, que muestra una precisión y recall consistentes. Los modelos de árboles de decisión también son robustos, pero su rendimiento no es tan destacado como el de los métodos de meta-aprendizaje o Naïve Bayes. Los resultados sugieren que los datos tienen ciertas relaciones lineales y no lineales, donde los modelos como Random Forest y Naïve Bayes pueden ofrecer una buena combinación de estabilidad y capacidad de generalización.

Model/Classificador	Class	precision	recall	f1-score
NaÏve Bayes	Mucho	0.72	0.67	0.69
	Poco	0.71	0.76	0.73
KNN	Mucho	0.74	0.52	0.61
	Poco	0.65	0.83	0.73
Decision Tree	Mucho	0.67	0.68	0.68
	Poco	0.68	0.68	0.68
SVM	Mucho	0.68	0.40	0.51
	Poco	0.54	0.79	0.64
Meta-learning (Random Forest)	Mucho	0.72	0.70	0.71
	Poco	0.71	0.73	0.72

Comparación de Métodos Utilizando Intervalos de Confianza
Naive Bayes:

- Intervalo de Confianza (95%): (0.696, 0.733)
- Es el modelo más estable con el intervalo más estrecho y un rendimiento consistentemente alto.

K-Nearest Neighbors (KNN):

- Intervalo de Confianza (95%): (0.662, 0.701)
- Menor estabilidad que Naïve Bayes, pero con un desempeño aceptable.

Árbol de Decisión:

- Intervalo de Confianza (95%): (0.657, 0.696)
- Similar a KNN, pero con menor precisión

SVM (Kernel Lineal):

- Intervalo de Confianza (95%): (0.509, 0.657)
- Ofreció un desempeño razonable, pero su intervalo más amplio refleja menor estabilidad en comparación con los otros modelos.

Random Forest:

- Intervalo de Confianza (95%): (0.636, 0.658)
- Más confiable que AdaBoost, pero menos que Naïve Bayes..

Naïve Bayes presenta el intervalo de confianza más estrecho, destacándose como el modelo más fiable. Random Forest es robusto, pero con menor precisión y estabilidad en comparación.

Model/Classificador	Intervalos de Confianza
NaÏve Bayes	(0.696, 0.733)
KNN	(0.662, 0.701)
Decision Tree	(0.657, 0.696)
SVM	(0.509, 0.657)
Meta-learning (Random Forest)	(0.636, 0.658)