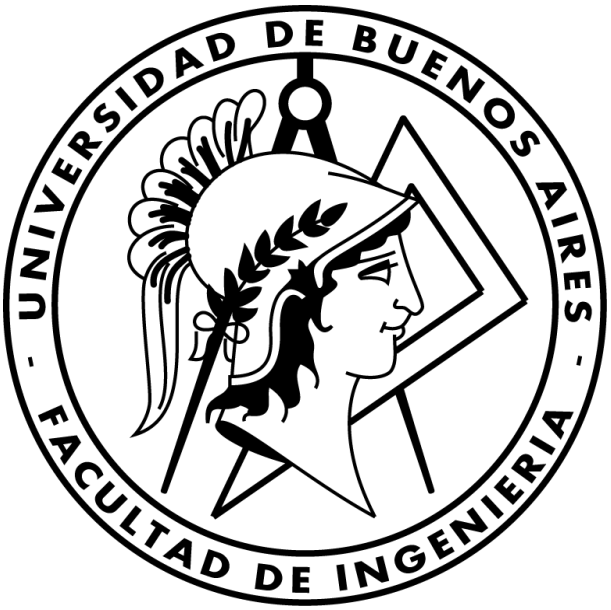
Universidad de Buenos Aires
Facultad de Ingeniería



Ciencia de Datos - TA047
Trabajo Práctico 2 - Cátedra Rodriguez
2do Cuatrimestre 2025

Grupo 07

Integrantes:

- Leandro Elias Brizuela
  - **Padrón:** 109842
  - **Mail:** lebrizuela@fi.uba.ar
- José Rafael Patty Morales
  - **Padrón:** 109843
  - **Mail:** jpatty@fi.uba.ar
- Jesabel Pugliese
  - **Padrón:** 110860
  - Mail: jpugliese@fi.uba.ar
- Candela Matelica
  - **Padrón:** 110641
  - Mail: cmatelica@fi.uba.ar

Introducción

El objetivo de este trabajo práctico es hacer la predicción más precisa posible acerca de si una crítica cinematográfica dada en español es **positiva** o **negativa**. Esto es un problema de clasificación binaria ya que buscamos categorizar datos en categorías finitas y conocidas. Para esto se nos brindó un set de entrenamiento `train.csv` y un set de validación `test.csv`. A continuación se hará una breve descripción del conjunto de datos que se utilizará para el entrenamiento de los modelos:

`train.csv` es un dataset que contiene 50.000 filas y 3 columnas, que estas últimas son:

- `ID`: Es el identificador único de una crítica cinematográfica.
  - ➢ **Tipo:** Categórica numérica ordinal. Aunque está representada con números, no tienen un significado cuantitativo, sirven para identificar y ordenar las críticas.
  - ➢ Tipo de dato: $int64$

- `review_es`: Es el contenido textual de la crítica cinematográfica.
  - ➢ Tipo: Datos de texto no estructurado. Son párrafos de texto libre.
  - ➢ Tipo de dato: $object$

- `sentimiento`: Nos dice el sentimiento de la crítica. Esta variable puede tomar valores:
  - ○ $positivo = el\ sentimiento\ es\ positivo.$
  - ○ $negativo = El\ sentimiento\ es\ negativo.$
  - ➢ Tipo: Categórica texto nominal. Puede tomar dos valores posibles que no tienen un orden entre ellos.
  - ➢ Tipo de dato: $object$

Preprocesamiento

En esta etapa se realizó un análisis de balanceo del dataset y una evaluación de la presencia de valores faltantes y atípicos en las variables relevantes.

Al examinar la variable objetivo `sentimiento`, se observó que presenta dos categorías: *positivo* y *negativo*, cada una con 25.000 registros.
Esto indica que el conjunto de datos se encuentra completamente balanceado, con una distribución del $50\% - 50\%$ entre ambas clases.

Dicha proporción entre las categorías asegura que los modelos no tiendan a favorecer una clase sobre la otra durante el entrenamiento, contribuyendo a una mejor generalización de los resultados.

Por otro lado, se verificó que ninguna de las variables consideradas en este análisis, tanto en el conjunto de entrenamiento (train) como en el de prueba (test), presentan valores nulos o vacíos.

Esto garantiza que los datos se encuentran completos y consistentes, por lo que no fue necesario aplicar técnicas de imputación ni eliminar registros por datos faltantes.

El análisis realizado permitió concluir que los datasets utilizados (train y test) están balanceados y libres de valores nulos, constituyendo una base sólida para las etapas posteriores de procesamiento textual, vectorización y modelado predictivo.

Limpieza de los datos:

Ya se mencionó que en el dataset no se encontró presencia de datos nulos. Sin embargo, se encontró un porcentaje del 3.6% de reseñas que se encontraban escritas totalmente en otro idioma, y como se trataba de un porcentaje muy bajo, decidimos eliminarlas.

Luego de la eliminación de esos datos, el dataset nos quedó con 24.131 reseñas de sentimiento *positivo*, y 24.048 de sentimiento *negativo*. Como la cantidad de reseñas para cada clase resultó muy parecida, supusimos que este pequeño desbalance no iba a afectar la predicción de los modelos.

Extracción de características:

El objetivo de la extracción de características fue construir un vocabulario representativo que mantuviera información semántica relevante sin introducir ruido.
Por ello, se aplicaron los siguientes criterios y técnicas:

- **Tokenización:** separamos el texto en unidades básicas (*tokens*) de palabras. En particular, utilizamos NLTK (*Natural Language Toolkit*).
    - Como NLTK falla al tokenizar palabras con el signo “¿”, los eliminamos con *regular expressions*.
- **Frecuencia mínima y máxima de aparición:** eliminamos palabras poco frecuentes (aparición en menos de 10 reseñas) y muy frecuentes (aparición en más del 90% de reseñas), ya que consideramos que no aportarían a la generalización.
- **Presencia de números:** no conservamos números ya que consideramos que introducirían mucho ruido, al ser reseñas de películas habría gran presencia de fechas, duración de la películas, etc. No realizamos ningún análisis extra

para aquellas palabras que por error contuviesen algún número, pues consideramos estas ocurrencias como errores de tipeo poco frecuentes que el propio procesamiento de TF-IDF no iba a considerar.

- **Presencia de mayúsculas:** no conservamos las mayúsculas y convertimos todo el texto a minúsculas. La presencia de palabras totalmente en mayúsculas era poco frecuente y no aportaba información suficiente como para justificar un vocabulario duplicado. Unificando el texto en minúsculas se reduce el ruido y la dimensionalidad.
- **Presencia de tildes en las palabras:** no conservamos tildes. La eliminación de tildes simplifica el preprocesamiento y reduce la variabilidad del texto sin generar una pérdida semántica significativa.
- **Presencia de la letra "ñ":** conservamos la "ñ", ya que su pérdida sí afecta de manera sustancial el significado de muchas palabras
- **Presencia de emojis:** no conservamos emojis en las reseñas ya que los análisis mostraron que su frecuencia es muy baja ($< 2\%$ para emojis con caracteres ASCII y 0.01% para emojis Unicode), por lo que su influencia estadística sería insignificante frente al resto del vocabulario.
- **Negation Propagation:** Se utilizó para lidiar con la negación, lo que hace es que entre un "no" y el siguiente signo de puntuación, se cambian las palabras concatenando "NO_" al principio. Con esto mejoramos la interpretación del sentimiento en frases con negaciones.
- **Normalización de palabras con letras repetidas:** Normalizamos palabras con letras repetidas (por ejemplo, convertir "buenaaa" a "buena") para reducir el ruido y la dimensionalidad.
- **TF-IDF:** Calculamos la matriz de ocurrencias con TF-IDF (*Term frequency-inverse document frequency*), que no solo cuenta cuántas veces aparece una palabra en un documento (como hace *Bag of Words*), sino que además pondera su importancia considerando cuán frecuente es en el resto del dataset, reduciendo el peso de las palabras comunes y destacando las más informativas o distintivas de cada texto.
- **Stopwords:** Eliminamos *stop words*, ya que son palabras demasiado frecuentes en el idioma español que no nos aportará información para diferenciar el sentimiento y solo aumentaría la dimensionalidad del vocabulario.
- **N-Gramas:** Observamos la frecuencia de ciertas combinaciones de palabras para extraer tokens que representaban hasta 3-gramas. Estos nos permiten recorrer o formar estas columnas pero con combinaciones de dos o más tokens.
- **Lematización:** llevamos las palabras del dataset a su forma canónica o esencial.
- **Embeddings:** Es una técnica que traduce palabras a listas de números (llamadas vectores). Estos vectores no tienen números al azar, estos colocan las palabras en un "mapa matemático" multidimensional. En este mapa, las

palabras que tienen significados similares están “físicamente cerca” una de la otra.

- Nota: Solo se utilizó para el modelo de red neuronal recurrente.

Extracción de características exclusivas de la red neuronal con BERT:
Para nuestra red neuronal, que trabaja en conjunto con BERT, se decidió aplicar una tokenización, donde cada palabra de nuestro vocabulario pasó a tener un ID con el cual se identificaba, dicho ID ya viene predefinido por el modelo tokenizador `bert-base-case`.
A continuación, a cada secuencia se le agregarán tokens especiales, que son solicitados para el buen funcionamiento de BERT, que son `CLS` (al inicio) y `SEP` (al final).
Además, todas nuestras frases deberán tener la misma longitud, esto quiere decir que aquellas frases que superen la longitud establecida serán truncadas, y aquellas que no la alcancen serán rellenadas con ceros.
Por último, se agrega una `attention_mask`, para que el modelo solo preste atención a lo que se encuentre encerrado entre los dos tekens especiales, que es donde tenemos la información relevante. De este modo, el modelo no presta atención a los ceros agregados anteriormente.

- Nota: Lo nombrado en esta sección es el único preprocesamiento que se aplicó para el entrenamiento de la red neuronal.

Cuadro de resultados

Modelo	F1 Test	Presicion Test	Recall Test	Accuracy Test	Kaggle
Bayes Naive	0.88	0.88	0.89	0.88	0.75
XGBoost	0.87	0.87	0.88	0.87	0.72
Random Forest	0.87	0.87	0.88	0.87	0.74
Red Neuronal	0.9	0.9	0.9	0.9	0.81
Ensamble	0.91	0.91	0.91	0.91	0.76

Descripción de Modelos

Métrica a maximizar:

Se decidió maximizar la métrica de **macro F1-Score** debido a las siguientes ventajas:

1. Si nuestro modelo falla en clasificar una de las clases (aunque sea perfecto en la otra), el Macro F1-Score bajará notablemente.
2. Para obtener un puntaje alto, obligamos al modelo a ser bueno clasificando en ambas las categorías por igual, no solo en una (exige consistencia).
3. Calcula el éxito de cada clase por separado antes de promediar. Esto asegura que una clase "fácil" no infle artificialmente el puntaje de una clase "difícil" (hace una evaluación independiente para cada clase).

Nota: En redes neuronales, no se pudo maximizar una de estas métricas de manera explícita, ya que no cumplen con el requisito de tener una pendiente constante para saber hacia dónde moverse en cada paso del entrenamiento.

Para el entrenamiento de los modelos se utilizó un vectorizador TF-IDF para la representación del texto considerando unigramas, bigramas y trigramas, pasando antes, por la función de preprocesamiento de los datos donde se aplicaron las técnicas como: Negation Propagation, eliminación de stopwords, lematización, etc.

XGBoost:

Inicialmente, se procedió a determinar una cantidad óptima de árboles para el modelo, para esto se probaron tres valores distintos y se seleccionó aquel que minimiza el error, evitando al mismo tiempo valores demasiado grandes que puedan extender el tiempo de entrenamiento. Posteriormente, se buscó una cantidad óptima de folds, para la validación cruzada. Ambos procesos de optimización de hiperparámetros se realizaron con Grid Search, debido a su reducida cantidad de combinaciones.
Para la búsqueda del resto de hiperparámetros se utilizó Random Search debido a su eficiencia en la búsqueda dentro de un espacio amplio de combinaciones.

Los parámetros a mejorar fueron:

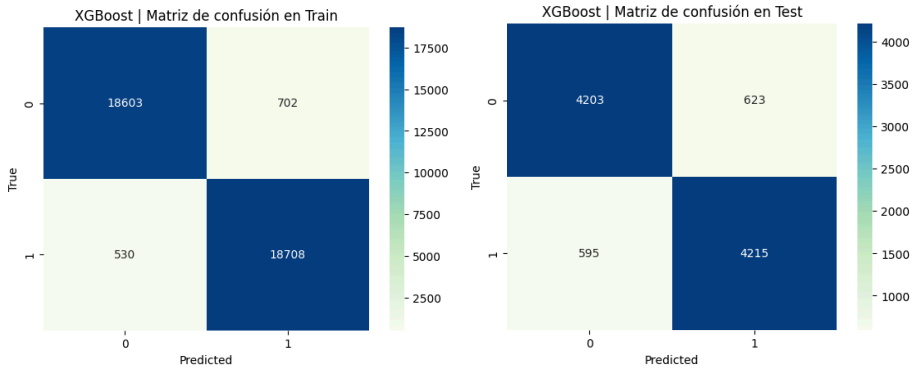
- `n_estimators`: Es el número de etapas de refuerzo de gradiente a realizar
- `subsample`: Es la fracción de muestras que se utilizará para ajustar los aprendices base individuales.

- `reg_lambda:` Penaliza los coeficientes grandes, pero no los lleva a cero, solo los achica.
- `reg_alpha:` Penaliza los coeficientes grandes forzando algunos a ser exactamente cero.
- `max_depth:` Es la profundidad máxima de los estimadores de regresión individual.
- `learning_rate:` Reduce la contribución de cada árbol.
- `colsample_bytree:` Controla qué porcentaje de las columnas se seleccionan aleatoriamente para construir cada árbol.

Parámetros del modelo:

```
{'n_estimators': 900,
'subsample': 0.6,
'reg_lambda': 4,
'reg_alpha': 0.4,
'max_depth': 6,
'learning_rate': 0.1,
'colsample_bytree': 0.6}
```

Matriz de confusión en Train y en Test:



Performance:

	F1 Test	Presicion Test	Recall Test	Accuracy Test
Train	0.97	0.96	0.97	0.97
Test	0.87	0.87	0.88	0.87

Random Forest:

Inicialmente, se procedió a determinar una cantidad óptima de árboles para el modelo, para esto se probaron dos valores distintos y se seleccionó aquel que minimiza el error, evitando al mismo tiempo valores demasiado grandes que puedan extender el tiempo de entrenamiento. Posteriormente, se buscó una cantidad óptima de folds, para la validación cruzada. Ambos procesos de optimización de hiperparámetros se realizaron con Grid Search, debido a su reducida cantidad de combinaciones.
Para la búsqueda del resto de hiperparámetros se utilizó Random Search debido a su eficiencia en la búsqueda dentro de un espacio amplio de combinaciones.

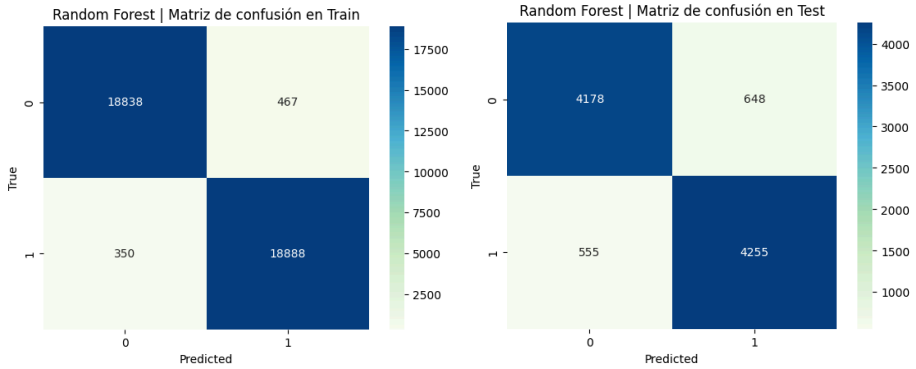
Los parámetros a mejorar fueron:

- `n_estimators`: Es el número de árboles que va a tener el bosque.
- `max_depth`: Es la profundidad máxima que puede crecer el árbol.
- `min_samples_split`: Es el número mínimo de muestras necesarias para dividir un nodo interno.
- `min_samples_leaf`: Es el número mínimo de muestras que debe haber en un nodo hoja.
- `max_features`: Es la cantidad de características a tener en cuenta al buscar la mejor división.
- `bootstrap`: Si se utiliza muestras bootstrap al construir árboles. Si es falso, se utiliza todo el conjunto de datos para construir cada árbol.

Parámetros del modelo:

```
{'n_estimators': 200,
'max_depth': None,
'min_samples_split': 2,
'min_samples_leaf': 2,
'max_features': log2,
'bootstrap': False}
```

Matriz de confusión en Train y en Test:



Performance:

	F1 Test	Presicion Test	Recall Test	Accuracy Test
Train	0.98	0.98	0.98	0.98
Test	0.88	0.87	0.88	0.88

Naïve Bayes:

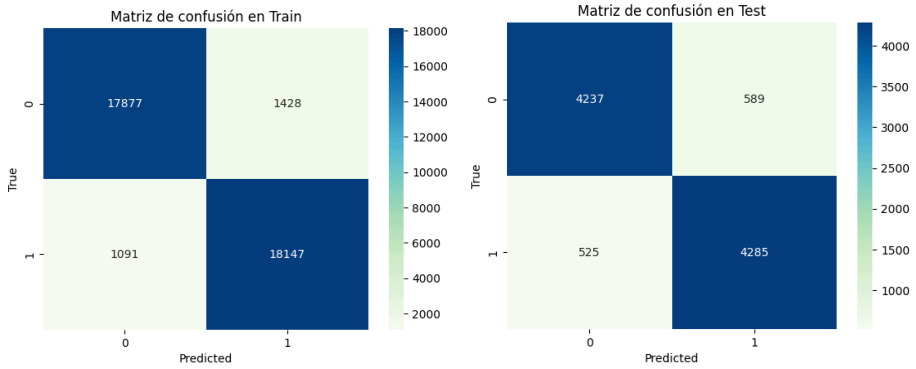
Se realizó una búsqueda de hiperparámetros, utilizando Grid Search con validación cruzada de 5-folds aprovechando que son pocos parámetros y así poder encontrar los valores óptimos, los parámetros a mejorar fueron:

- `alpha:` Parámetro de Smoothing.
- `fit_prior:` Nos dice si se deben aprender las probabilidades previas de las clases. Controla cómo el modelo juzga la probabilidad inicial de cada clase.

Parámetros del modelo:

```
{'alpha': 0.1,
'fit_prior': False}
```

Matriz de confusión en Train y en Test:



Performance:

	F1 Test	Presicion Test	Recall Test	Accuracy Test
Train	0.93	0.93	0.94	0.93
Test	0.88	0.88	0.89	0.88

Linear Support Vector Machines (LSVM):

Se seleccionó el modelo `LinearSVC` debido a que, tras el proceso de vectorización, los datos textuales generan un espacio de alta dimensionalidad, en donde pueden tender a estar linealmente separables.

- Nota: Este modelo fue elegido para ser usado en el modelo de ensamblaje híbrido.

Para la búsqueda de los hiperparámetros se calculó la cantidad de folds óptima y se utilizó validación cruzada con Random Search debido a su eficiencia en la búsqueda dentro de un espacio amplio de combinaciones.

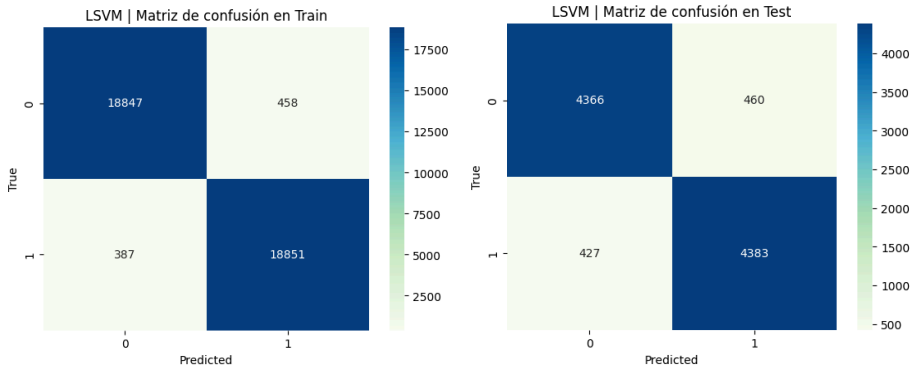
Los parámetros a mejorar fueron:

- `C`: Parámetro de regularización, controla qué tan estricto es el modelo con los errores de clasificación en el entrenamiento.
- `loss`: Función de pérdida, controla cómo se penalizan los errores al entrenar.

Parámetros del modelo:

```
{'C': 0.3,
'loss': 'squared_hinge'}
```

Matriz de confusión en Train y en Test:



Performance:

	F1 Test	Presicion Test	Recall Test	Accuracy Test
Train	0.98	0.98	0.98	0.98
Test	0.91	0.91	0.91	0.91

Ensamble híbrido:

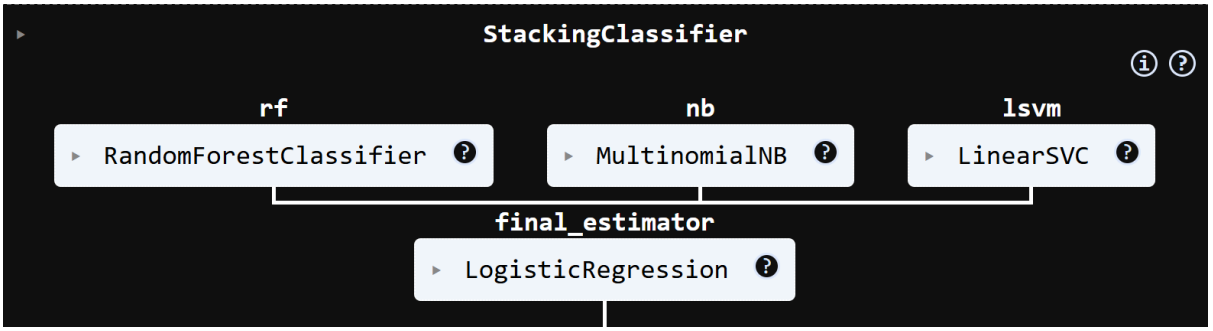
Decidimos probar dos modelos de ensamblaje híbrido: `Voting` y `Stacking`, utilizando tres de los modelos previamente entrenados (`Random Forest`, `Naïve Bayes` y `Linear Support Vector Machines`).

Hard Voting:

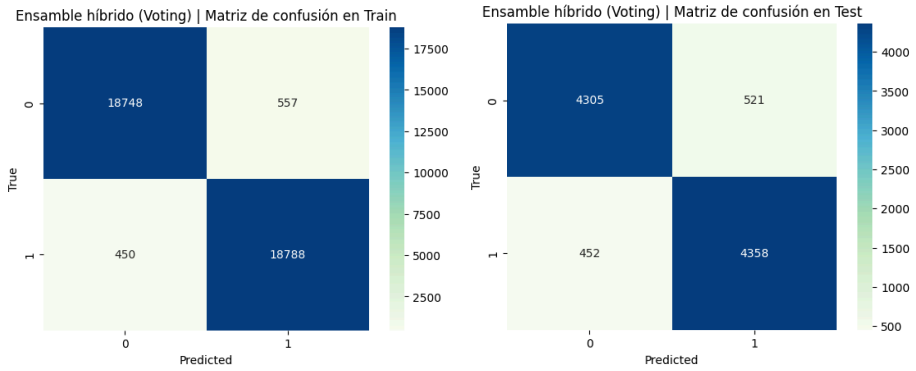
Este modelo de ensamble híbrido es simple y computacionalmente liviano, pero presenta limitaciones: no pondera la calidad individual de los modelos y no permite aprender relaciones adicionales entre ellos. Por este motivo, si bien funcionó de manera aceptable, no ofreció mejoras significativas respecto de los modelos individuales.

Stacking:

En este caso, utilizamos como meta-modelo una Regresión Logística, la cual se encargó de combinar las predicciones de los modelos base. Este enfoque permitió capturar relaciones adicionales entre las salidas de cada modelo y aprovechar mejor la información que aportaba cada uno. En nuestras pruebas, el Stacking obtuvo un rendimiento superior al Hard Voting: alcanzó el mejor F1-Score, y presentó menor varianza entre iteraciones de cross-validation. Por esto decidimos continuar con esta estrategia.



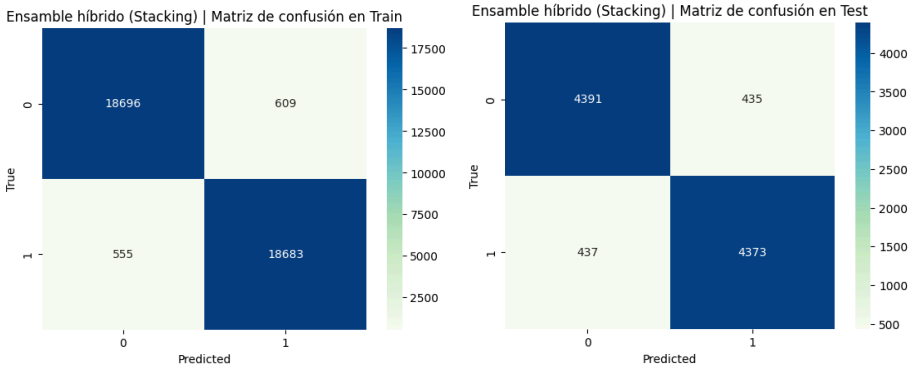
Matriz de confusión en Train y en Test (Hard Voting):



Performance:

	F1 Test	Presicion Test	Recall Test	Accuracy Test
Train	0.97	0.97	0.98	0.97
Test	0.90	0.89	0.91	0.90

Matriz de confusión en Train y en Test (Stacking):



Performance:

	F1 Test	Presicion Test	Recall Test	Accuracy Test
Train	0.97	0.97	0.97	0.97
Test	0.91	0.91	0.91	0.91

Elección del modelo:

Podemos observar que dieron mejores métricas en test para el ensamble híbrido con **Stacking**, por lo que se decidió quedarnos con este modelo.

Red Neuronal Recurrente:

Inicialmente, implementamos una red neuronal recurrente, ya que nos pareció una buena opción teniendo en cuenta de que es un tipo de red neuronal especialmente diseñada para gestionar secuencias de datos, como series temporales, texto, etc. A diferencia de otras redes neuronales, las RNN tienen conexiones que se revierten sobre sí misma, lo que les proporciona una forma de “memoria” que les ayuda a “recordar” las entradas anteriores en la secuencia. Esta estructura única las hace muy buena para tareas donde la dinámica temporal y el orden de las entradas son importantes.
En este caso, se utilizó la variante conocida LSTM (Long Short-Term Memory) las cuales son capaces de aprender dependencias a largo plazo en los datos, que lo que las RNN tradicionales a menudo no logran capturar debido a problemas como desvanecimiento de gradiente. Para este problema, en donde analizamos el sentimiento de reseñas de películas, comprender el texto es crucial y muchas veces

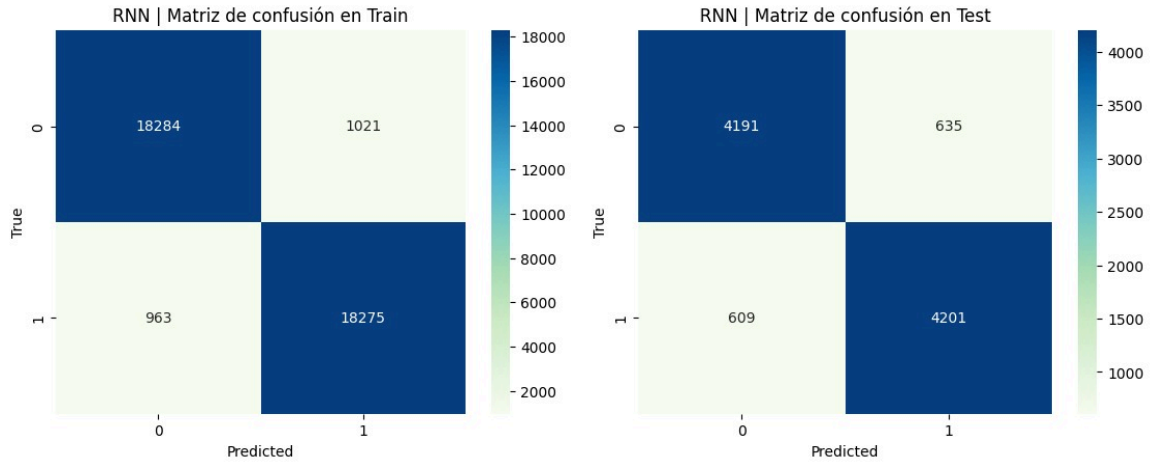
el sentimiento de estas se basan en palabras recientes, sino que puede verse influenciado por palabras que aparecen mucho antes en el texto.

Uso de embeddings pre-entrenados:
Para el entrenamiento de esta red se utilizaron embeddings pre-entrenados, más precisamente los embeddings pre-entrenados por FasText que tienen una dimensionalidad 300.
Para integrar esto al modelo, se realizó un matching entre nuestro vocabulario y estos embeddings externos: se creó una matriz de pesos donde cada palabra de nuestro dataset heredó el vector pre-entrenado correspondiente si este se encontraba disponible.

Arquitectura de la red implementada:
Primero tenemos una capa de Embeddings que transforma palabras codificadas con números enteros en vectores numéricos, donde utilizamos padding para estandarizar la longitud de entradas. Estos embeddings se procesan con una LSTM bidireccional para extraer patrones y relaciones contextuales, luego las últimas capas ocultas se combinan y pasan por capas densas de activación ReLu y Dropout.
Entrones:

- En la capa de entrada tenemos 300 neuronas, para cada palabra.
- En la capa recurrente tenemos $126 + 126$ neuronas, debido a la bidireccionalidad.
- Una capa densa oculta de 64 neuronas.
- En la capa de salida tenemos 1 neurona para la desición binaria.

Matriz de confusión en Train y en Test:



Performance:

	F1 Test	Presicion Test	Recall Test	Accuracy Test
Train	0.95	0.95	0.95	0.95
Test	0.87	0.87	0.87	0.87

En Kaggle, este modelo tuvo un puntaje de 0.71 aproximadamente, lo que no nos resultó muy poco convincente por lo que decidimos explorar otro modelo de redes neuronales.

Red Neuronal con BERT:

Antes de describir la arquitectura de nuestra red neuronal implementada, presentaremos la parte principal de esta.

¿Qué es BERT?
BERT o Bidirectional Encoder Representations from Transformers es un modelo de Aprendizaje Automático pre-entrenado con el objetivo de mejorar la comprensión contextual del texto sin etiquetar en una amplia gama de tareas, aprendiendo a predecir el texto que podría aparecer antes y después de otro texto (bidireccional).

Para la implementación se utilizó el modelo de BERT proporcionado por la plataforma Hugging Face, precisamente la versión en español desarrollada por la Universidad Católica de Chile (**dccuchile/bert-base-spanish-wwm-cased**).

Arquitectura de la red neuronal implementada:
La arquitectura consta del modelo BERT pre-entrenado mencionado anteriormente y a este se le agrega capa de neuronas. Esta capa de neuronas tiene la misma cantidad de neuronas de entrada que el número de neuronas de salida del modelo BERT (esta capa de neuronas se conecta encima BERT), o sea 768. Además, esta capa tendrá sólo 2 neuronas de salida, ya que sólo tenemos dos clases (`positivo` o `negativo`).

Hiper-parámetros:

- `N_CLASSES`: Es la cantidad de clases que puede predecir nuestro modelo. Como este es un problema de clasificación binaria (el modelo va a clasificar si una reseña tiene un sentimiento `positivo` o `negativo`), su valor es 2.

- `MAX_LEN`: Representa la máxima cantidad de palabras por reseña utilizadas para el entrenamiento del modelo.
- `BATCH_SIZE`: Representa la cantidad de reseñas por paquete que se le mandan en una pasada al modelo para su entrenamiento.
- `EPOCHS`: Representa la cantidad de iteraciones de entrenamiento, es la cantidad de veces que se pasará el dataset en entrenamiento en la red.
- `L_RATE`: Representa la tasa de aprendizaje.

La búsqueda de estos hiper-parámetros fue hecha a mano, fuimos probando con diferentes valores en cada parámetro (excepto en `N_CLASSES`) quedándonos con lo que dieron el mejor desempeño, estos fueron:

- `N_CLASSES = 2`
- `MAX_LEN = 200`
- `BATCH_SIZE = 12`
- `EPOCHS = 5`
- `L_RATE = 2e-5`

Método de regularización utilizado para el entrenamiento:
Se utilizó un **Dropout** para evitar co-dependencias en las conexiones de la red. La idea es “apagar” aleatoriamente las conexiones durante el entrenamiento. Se utiliza para generalizar mejor.

Optimizador:
El optimizador que se eligió para la red neuronal fue **AdamW**, que es una variante del optimizador **Adam**, donde se mejora la capacidad de generalización al corregir la forma en que se aplica la regularización de pesos.
Además, podemos destacar una mejor generalización de los datos comparada con Adam.

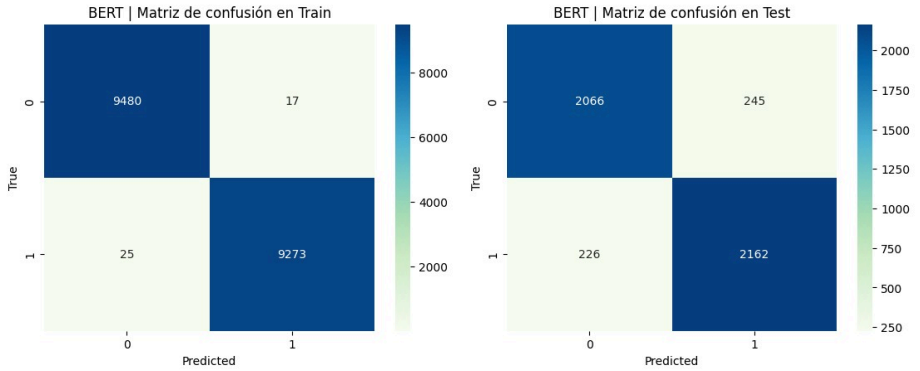
Se utilizó la librería `PyTorch` para:

- Implementar una clase `ReseniasDataset` que va a servir para preparar y empaquetar los datos para que puedan ser procesados por el modelo BERT pre-entrenado.
- Orquestar la carga de los datos utilizando un `DataLoader` con el cual divide el dataset en paquetes más pequeños y después cargarlos en paralelo utilizando cuatro procesos para acelerar la carga.
- Implementar una clase `BERTClassifier` la cuál se compone de una capa que contiene el modelo BERT pre-entrenado, seguida por una capa “`Dropout`” que es seguida por una capa “`Linear`” (que representa la capa de neuronas) la cuál nos devolverá el resultado de la clasificación binaria.

Debemos recordar que el modelo BERT, nos devuelve dos resultados, el primero es la representación completa y contextual de cada token de nuestra frase y el segundo resultado es el resumen de toda la frase condensada en un solo vector, lo cual nos va a permitir hacer el análisis de sentimientos. El segundo resultado es el que nos interesa para nuestra red neuronal para que la misma lo procese y entienda si pertenece a cierta clase o no.

Decidimos implementar este tipo de red neuronal utilizando BERT debido a su capacidad de procesamiento bidireccional (aprende a predecir el texto que podría aparecer antes y después de otro texto), que permite una mejor comprensión del contexto de las reseñas. Además que este modelo pre-entrenado es muy usado para análisis de sentimientos.

Matriz de confusión en Train y en Test:



Performance:

	F1 Test	Presicion Test	Recall Test	Accuracy Test
Train	1.0	1.0	1.0	1.0
Test	0.9	0.9	0.9	0.9

Conclusiones generales

Tenemos que destacar la gran importancia de la exploración de los datos, ya que gracias a esta, pudimos notar la presencia de palabras en otro idioma y eso nos llevó a encontrar reseñas que estaban completamente escritas en idiomas distintos al español, lo que ensuciaba nuestro dataset. Una parte que ayudó mucho al rendimiento de los modelos fue la de preprocesamiento porque a medida que íbamos añadiendo nuevas tareas de preprocesamiento vimos cómo iban mejorando (aunque sea ligeramente) las métricas de los modelos.

Después de realizar toda la etapa de exploración de datos, preprocesamiento con todas sus tareas y entrenamiento todos los modelos pudimos observar que el que mejor performance logró con el conjunto de prueba fué el modelo de ensamblaje híbrido, el cuál logró un puntaje de 0.91 en todas las métricas. Sin embargo, en Kaggle el modelo que mayor performance logró fue la red neuronal con BERT. Esto puede indicar que hubo un mayor sobreajuste en el modelo de ensamblaje híbrido.

En cuanto al modelo más rápido y sencillo de implementar fue Naïve Bayes, esto se debe a que no requiere de un cálculo complejo (como sí lo requieren modelos como los de redes neuronales), ya que el mismo encara el problema desde un lado estadístico. A pesar de su simplicidad, obtuvo métricas bastante aceptables, es decir, que en términos de complejidad y tiempo de entrenamiento, resulta un modelo bastante útil para este tipo de problema.

El modelo de redes neuronales combinado con BERT, fue el que obtuvo un mejor desempeño general (en test y Kaggle). Esto se debe a que se trabajó en conjunto con la red pre-entrenada BERT, más una arquitectura de redes neuronales. Si bien fue el que mejores resultados obtuvo, su costo de entrenamiento era bastante elevado. Por este motivo, consideramos que aún existe un rango de mejora, ya sea optimizando el proceso de entrenamiento o ajustando los parámetros para lograr un mejor equilibrio entre rendimiento y eficiencia computacional.

Como se mencionó anteriormente, el costo computacional no era la única limitación, sino que también se debe destacar que para el desarrollo de dichos modelos se utilizó la plataforma gratuita de **Google Colab**, lo cual impuso una cota superior a los recursos disponibles. La capacidad limitada de memoria RAM y uso de la GPU de este entorno restringido condiciona el entrenamiento de nuestros modelos. Ya que al momento de entrenar los modelos, muchos de estos precisaban de una gran cantidad de espacio o en su respectivo caso, tardaban mucho tiempo en entrenarse, lo que producía que se agote el tiempo de uso de la GPU. En consecuencia, el proceso de entrenar y comparar diferentes modelos se tornaba lento y dificultoso.

En cuanto a la red neuronal recurrente, consideramos que el modelo no alcanzó el desempeño esperado debido a que habían bastantes palabras que no tenían un embedding asignado y no se tomaron en cuenta. Posiblemente creando embeddings propios para esas palabras el rendimiento hubiera sido mejor, sin embargo, por falta de tiempo no pudimos explorar esa alternativa.

En conclusión, con un mejor entorno de desarrollo, dichas barreras no hubieran existido o su impacto hubiera sido menor. Esto nos habría permitido experimentar con una mayor variedad de hiperparámetros en los diferentes modelos, provocando posiblemente una mejora en los resultados obtenidos.

Tareas realizadas

Integrante	Principales Tareas realizadas	Promedio Semanal (hs)
Leandro Elias Brizuela	- Preparación del entorno de Colab - Entrenamiento de XGBoost - Entrenamiento Naive Bayes - Entrenamiento Red Neuronal - Elaboración del Informe	8 hrs
José Rafael Patty Morales	- Preparación del entorno de Colab - Entrenamiento de XGBoost - Entrenamiento Naive Bayes - Entrenamiento Red Neuronal - Elaboración del Informe	8 hrs
Jesabel Pugliese	- Extracción de características - Entrenamiento de Random Forest - Entrenamiento de Ensamble Híbrido - Elaboración del Informe	6 hrs
Candela Matelica	- Extracción de características - Entrenamiento de Random Forest - Entrenamiento de Ensamble Híbrido - Elaboración del Informe	6 hrs