



CIENCIA DE DATOS
CÁTEDRA RODRÍGUEZ

Trabajo Práctico 2

2C 2025

2 de diciembre de 2025

Nombre	Padrón	Mail
Tomas Caporaletti	108598	tcaporaletti@fi.uba.ar
Helen Elizabeth Chen	110195	hchen@fi.uba.ar
Blas Sebastián Chuc	110253	bchuc@fi.uba.ar
Franco Agustin Rodriguez	108799	frodriguez@fi.uba.ar

Índice

1. Introducción	3
1.1. Descripción del problema	3
1.2. Descripción del dataset	3
1.3. Modificaciones realizadas sobre el dataset	4
2. Bayes Naïve	4
2.1. Exploración de técnicas de preprocesamiento	4
2.2. Hipótesis o supuestos tomados	5
2.3. Entrenamiento del modelo	5
2.4. Análisis de resultados	5
3. Random Forest	6
3.1. Detección y filtrado de reseñas no escritas en español	6
3.2. Técnicas de preprocesamiento	7
3.3. Hipótesis o supuestos tomados	7
3.4. Entrenamiento	7
3.4.1. Entrenamiento del vectorizador TF-IDF	7
3.4.2. Entrenamiento del Random Forest	8
3.5. Análisis de resultados	9
4. XGBoost	10
4.1. Preprocesamiento y vectorización	11
4.2. Entrenamiento del modelo base	11
4.3. Búsqueda de hiperparámetros	11
4.4. Entrenamiento final y predicciones en Kaggle	12
5. Red Neuronal	12
5.1. Preprocesamiento	12
5.2. Elección de arquitectura	13
5.3. Mejor modelo	13
5.4. Aclaraciones sobre el modelo	14
6. Ensamble	14
7. Resultados	16
7.1. Cuadro de resultados	16
7.2. Modelo seleccionado como mejor predictor	16
8. Conclusiones Generales	16
9. Tareas realizadas	17

1. Introducción

En este trabajo utilizaremos una colección de críticas cinematográficas en español con el objetivo de identificar si cada una expresa una opinión positiva o negativa. Para ello, se aplicarán técnicas de procesamiento de lenguaje natural y distintos modelos de clasificación.

1.1. Descripción del problema

El problema consiste en clasificar cada crítica según el tipo de opinión que expresa. Como el texto no puede procesarse directamente, debe preprocesarse y transformarse en una representación numérica mediante técnicas como bag of words u otros métodos de vectorización. Sobre estas representaciones, los modelos de clasificación aprenden los patrones que permiten distinguir entre críticas positivas y negativas.

1.2. Descripción del dataset

Utilizamos dos datasets compuestos por reseñas cinematográficas en español, provistos por la cátedra: `train.csv` y `test.csv`. Cada uno presenta características propias y se emplea con fines diferentes, como se detalla a continuación.

Conjunto de train

El dataset `train.csv` contiene 50 000 registros, cada uno corresponde a una reseña. Está compuesto por tres columnas: `ID`, un identificador numérico único para cada instancia; `review_es`, que incluye el texto completo de la reseña; y `sentimiento`, que indica si la opinión expresada es positiva o negativa. Todas las columnas presentan valores no nulos, y el archivo tiene un tamaño aproximado de 1.1 MB.

El dataset presenta un balance total en la variable `sentimiento`, ya que la mitad de las reseñas están etiquetadas como positivas y la otra mitad como negativas.

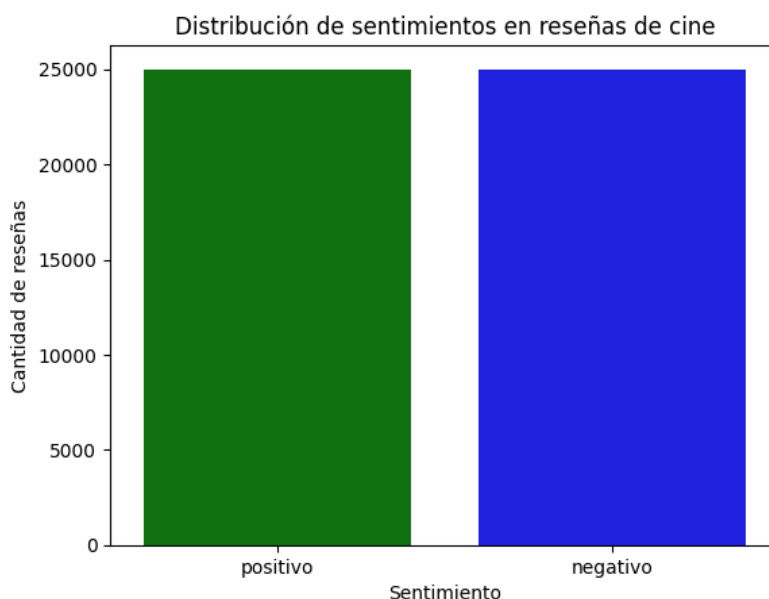


Figura 1: Distribución de la variable sentimiento

Este es el dataset que utilizaremos para el entrenamiento de los distintos modelos y para el cálculo de sus métricas, lo que permitirá compararlos entre sí en términos de rendimiento.

Conjunto de test

El conjunto `test.csv` contiene 8 599 registros y está compuesto por dos columnas: `ID`, un identificador numérico único para cada reseña, y `review_es`, que contiene el texto de la reseña. A diferencia del conjunto de entrenamiento, `test.csv` no incluye la variable `sentimiento`. Todas las entradas presentan valores no nulos, y el archivo tiene un tamaño aproximado de 134.5 KB.

Tras entrenar los modelos, este dataset se empleó para generar sus predicciones, las cuales fueron utilizadas para participar en la competencia de Kaggle del curso.

1.3. Modificaciones realizadas sobre el dataset

En el dataset `train` se transformaron los valores de la variable `sentimiento`, asignando el valor 1 a las reseñas positivas y 0 a las negativas. De esta manera, la variable quedó representada como una categoría binaria numérica, adecuada para su utilización en los modelos de clasificación.

2. Bayes Naïve

Para el desarrollo del modelo de clasificación, primero se dividió el dataset de `train` con una proporción de 80 % para entrenamiento y 20 % para validación.

Se aplicó el parámetro `stratify=y` para garantizar que la distribución de clases (positivas/negativas) se mantuviera equilibrada en ambos subconjuntos, evitando sesgos en el aprendizaje.

2.1. Exploración de técnicas de preprocesamiento

Decidimos explorar dos estrategias secuenciales de preprocesamiento, buscando mejorar la representación semántica del texto:

Modelo 1

- **Limpieza básica:** Conversión a minúsculas, eliminación de etiquetas HTML y eliminación de caracteres no alfabéticos.
- **Normalización con Lematización:** Se utilizó `WordNetLemmatizer`.
 - *Limitación detectada:* Este lematizador está diseñado principalmente para inglés. Al aplicarlo sobre texto en español, no redujo correctamente las palabras a su raíz, funcionando efectivamente solo como un tokenizador básico.
- **Stopwords:** Se eliminaron todas las *stopwords* de la lista estándar. Esto provocó la pérdida de contexto en frases negativas (ej: “no me gusta” pasaba a ser “gusta”).

Modelo 2

Implementamos un preprocesamiento más robusto para corregir las falencias del anterior:

- **Normalización con Stemming:** Se reemplazó el lematizador por `SnowballStemmer('spanish')`. Esto permitió reducir eficazmente las variaciones verbales y de género a su raíz común (ej: “gustó”, “gustaba” → “gust”), reduciendo la dimensionalidad del vocabulario y así mejorando la capacidad de generalización de Naive Bayes.
- **Tratamiento de Stopwords (personalizado):** Se modificó la lista para excluir y conservar explícitamente las negaciones y modificadores de intensidad (ej: “no”, “ni”, “sin”, “pero”, “muy”, “nunca”). Esto es crítico para el Análisis de Sentimiento, ya que eliminar un “no” invertiría el significado de una frase negativa (ej: “no me gusta” pasaría a ser “gusta”).

2.2. Hipótesis o supuestos tomados

Para la elección del procesamiento del modelo, nos basamos en los siguientes supuestos:

- **Independencia de palabras (por Naive Bayes):** Asumimos que cada palabra aporta evidencia de manera independiente, sin importar su relación con las demás.
- **Relevancia de la frecuencia (TF-IDF):** Las palabras que aparecen frecuentemente en un documento pero raramente en el corpus general (TF-IDF alto) son las que mejor discriminan el sentimiento de la reseña.
- **Ruido en palabras únicas:** Asumimos que las palabras que aparecen en una sola reseña (`min_df`) son probablemente errores tipográficos o nombres propios irrelevantes que introducen ruido.
- **Importancia del Contexto Local (N-gramas):** En el Modelo 2, partimos de la hipótesis de que el sentimiento no reside solo en palabras aisladas, sino en secuencias cortas (ej: “no recomiendo”). Por eso ampliamos la búsqueda a bigramas y trigramas.

2.3. Entrenamiento del modelo

Utilizamos un *Pipeline* que integra la vectorización (`TfidfVectorizer`) y el clasificador (`MultinomialNB`).

Para la optimización de hiperparámetros utilizamos `GridSearchCV`, haciendo una validación cruzada (`cv=5`) para optimizar los siguientes parámetros:

Hiperparámetro	Modelo 1	Modelo 2	Explicación
<code>ngram_range</code>	(1,1), (1,2)	(1,1), (1,2), (1,3)	El Modelo 2 explora hasta trigramas para capturar frases complejas de negación.
<code>min_df</code>	1, 2	2	Se filtra el ruido eliminando palabras que aparecen en menos de 2 reseñas.
<code>max_df</code>	N/A	0.5, 0.9	El Modelo 2 ignora palabras extremadamente frecuentes que no aportan información.
<code>binary</code>	N/A	True, False	El Modelo 2 evalúa si importa la frecuencia de la palabra o solo su presencia.
<code>sublinear_tf</code>	N/A	True	El Modelo 2 suaviza logarítmicamente las frecuencias para evitar que palabras repetitivas dominen.
<code>alpha</code> (NB)	0.1, 0.5, 1.0	0.1, 0.5, 1.0	Parámetro de suavizado de Laplace. Evita asignar probabilidad cero a palabras no vistas.

Cuadro 1: Comparación de hiperparámetros probados

2.4. Análisis de resultados

A continuación, se comparan las métricas obtenidas:

- **Modelo 1 (Inicial):**
 - Mejor Score CV: 0.8709

- F1-Score Validación: 0.8803
- Mejores Params: `alpha=0.1, min_df=1, ngram=(1,2)`
- **Modelo 2 (Optimizado):**
 - Mejor Score CV: **0.8768**
 - F1-Score Validación: **0.8810**
 - Mejores Params: `alpha=0.1, ngram=(1,3), max_df=0.9, sublinear_tf=True`

Conclusiones

- **Mejora Metodológica:** Se observa una mejora en el rendimiento del Modelo 2. Aunque la diferencia numérica en el F1-Score no es masiva (~ 0.006 en CV), metodológicamente el Modelo 2 es superior. Esto se debe a una mejor comprensión de la estructura del lenguaje español gracias a la implementación de *stemming* específico y la conservación estratégica de las negaciones.
- **Impacto de los N-gramas:** La selección automática de `ngram_range=(1, 3)` en el Modelo 2 confirma que el análisis de sentimiento se beneficia del contexto. Leer secuencias de hasta tres palabras (trigramas) permite al modelo capturar matices y frases negativas que se pierden al analizar palabras aisladas.
- **Análisis de Generalización y Sobreajuste:** En ambos modelos, la similitud entre el *Score de Validación* y el *Mejor Score CV* indica que la validación cruzada fue fiable y estimó correctamente el rendimiento real del modelo.

Sin embargo, al contrastar esto con el rendimiento casi perfecto en el conjunto de entrenamiento, se detecta un claro **sobreajuste**. Existe una brecha de ~ 11 puntos, lo que indica que el modelo ha memorizado ruido del set de entrenamiento debido a su alta complejidad.

Si bien el Modelo 2 presenta una varianza alta (sobreajuste) que podría reducirse en futuras iteraciones aumentando la regularización, se selecciona como el **mejor modelo** entre ambos. Está construido sobre cimientos sólidos que procesan correctamente la semántica del español, mientras que el Modelo 1 presentaba fallas estructurales en el preprocesamiento. Por lo tanto, el Modelo 2 ofrece la base más confiable y correcta teóricamente.

3. Random Forest

En este apartado se presenta el modelo Random Forest, el cuál se basa en el entrenamiento de múltiples árboles de decisión mediante bagging. El modelo será aplicado sobre las representaciones vectorizadas de las reseñas para evaluar su desempeño en la tarea de clasificación del sentimiento.

3.1. Detección y filtrado de reseñas no escritas en español

Para garantizar la coherencia lingüística del conjunto de train, se realizó un análisis del idioma presente en cada reseña utilizando la biblioteca `langdetect`. Este procedimiento permitió identificar automáticamente la lengua predominante en cada texto. Los resultados mostraron que aproximadamente 1,800 reseñas estaban redactadas en inglés, lo que representa menos del 4 % del total del dataset.

Dado que el enfoque del trabajo se centra exclusivamente en el análisis de sentimiento en español, y considerando que esta fracción minoritaria no aportaba información relevante al objetivo planteado, se decidió excluir dichas reseñas. Este filtro permite evitar ruido lingüístico en la etapa de vectorización y entrenamiento, reduciendo posibles sesgos y mejorando la consistencia del modelo.

3.2. Técnicas de preprocesamiento

Para este modelo se aplicó la misma técnica de preprocesamiento de texto utilizada en el modelo 2 de Bayes Naïve, detallada en la Sección 2.1. Esto incluye el uso del SnowballStemmer en español y una lista de stopwords personalizada que conserva negaciones y modificadores de intensidad. La decisión de emplear este mismo esquema se debe a que ofrece una representación más compacta y consistente del texto, reduciendo la variabilidad morfológica sin perder información semántica relevante para el análisis de sentimiento.

3.3. Hipótesis o supuestos tomados

La decisión de utilizar estas técnicas de procesamiento de texto se fundamenta en los siguientes supuestos:

- **Reducción del espacio de features para mejorar la estabilidad del modelo:** Random Forest funciona mejor cuando las características no son excesivamente dispersas o ruidosas. Al aplicar stemming y un filtrado cuidadoso de stopwords, se asume que el espacio vectorial resultante será más compacto, lo que facilita que los árboles encuentren divisiones informativas sin sobreajustar a términos poco frecuentes.
- **Conservar negaciones e intensificadores mejora la discriminación del sentimiento en los splits:** Se presupone que términos como “no”, “nunca”, “muy”, “demasiado” tienen una importancia clave para la clasificación de sentimiento, y su conservación aporta señales fuertes que los árboles en el bosque pueden usar para generar divisiones con mayor ganancia.
- **El modelo no requiere features especialmente normalizadas o dependientes del orden:** Dado que los árboles no dependen de la escala de las variables ni del orden de los tokens, se asume que un preprocesamiento basado en bag of words o TF-IDF funciona de manera adecuada, sin necesidad de transformaciones más complejas.

3.4. Entrenamiento

Para la etapa de entrenamiento del modelo, se dividió el dataset de train con una proporción de 80 % para entrenamiento y 20 % para validación.

Se aplicó el parámetro `stratify=y` para garantizar que la distribución de clases (positivas/negativas) se mantuviera equilibrada en ambos subconjuntos, evitando sesgos en el aprendizaje.

3.4.1. Entrenamiento del vectorizador TF-IDF

Para este entrenamiento se construyó un Pipeline compuesto por dos etapas: la vectorización del texto mediante `TfidfVectorizer` y el modelo `RandomForestClassifier` con sus parámetros por defecto. El objetivo fue analizar cómo distintas configuraciones del espacio vectorial impactan en el rendimiento del modelo, manteniendo fijo el clasificador.

La optimización se realizó utilizando `GridSearchCV`, con validación cruzada (`cv=3`) y la métrica F1-macro como criterio de selección. El grid abarcó exclusivamente los hiperparámetros del vectorizador, entre ellos:

Resultados para el mejor `TfidfVectorizer` obtenido:

- **Mejor score CV:** 0.8423
- **Mejores parámetros:** `{"binary": False, "sublinear_tf": True, "ngram_range": (1, 3), "max_df": 0.5, "min_df": 5}`

Hiperparámetro	Valores
ngram_range	(1,1), (1,2), (1,3)
min_df	2, 5
max_df	0.5, 0.7, 0.9
binary	True, False
sublinear_tf	True, False

Cuadro 2: Combinaciones de hiperparámetros probados TfidfVectorizer

3.4.2. Entrenamiento del Random Forest

En este segundo entrenamiento se utilizó un Pipeline donde la etapa de vectorización se fijó mediante el TfidfVectorizer previamente entrenado con los mejores hiperparámetros obtenidos en el análisis anterior. De esta forma, el espacio vectorial permanece estable, permitiendo concentrar el proceso de optimización únicamente en el comportamiento del RandomForestClassifier.

Se empleó nuevamente GridSearchCV con validación cruzada (cv=3) y la métrica F1-macro. El grid evaluado incluyó los siguientes hiperparámetros:

Hiperparámetro	Valores
criterion	'gini'
n_estimators	100, 200, 300
max_depth	5, 10, 15
min_samples_split	5, 10
min_samples_leaf	2, 4, 6
max_features	'sqrt'

Cuadro 3: Combinaciones de hiperparámetros probados Random Forest

Mejores parámetros obtenidos: {'criterion': 'gini', 'max_depth': 15, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'min_samples_split': 10, 'n_estimators': 300}

3.5. Análisis de resultados

Métricas en train y test

Class / Metric	Precision	Recall	F1-score	Support
TRAIN				
Negativo	0.922	0.829	0.873	19306
Positivo	0.844	0.930	0.885	19240
Accuracy	0.879		(on 38546 samples)	
Macro avg	0.883	0.879	0.879	38546
Weighted avg	0.883	0.879	0.879	38546
TEST				
Negativo	0.856	0.794	0.824	4827
Positivo	0.808	0.866	0.836	4810
Accuracy	0.830		(on 9637 samples)	
Macro avg	0.832	0.830	0.830	9637
Weighted avg	0.832	0.830	0.830	9637

Cuadro 4: Métricas Random Forest TRAIN - TEST

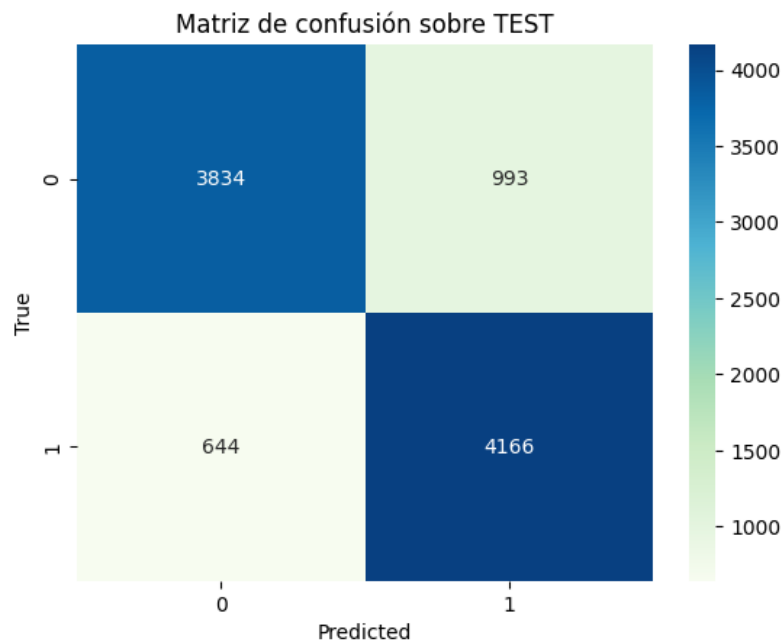


Figura 2: Matriz de confusión TEST

El modelo aprende bien en entrenamiento, hay un buen balance entre clases. La clase positiva tiene mayor recall, lo que indica que el modelo detecta bien las críticas positivas, aunque a costa de una menor precisión (más falsos positivos). La clase negativa muestra lo contrario, alta precisión pero menor recall.

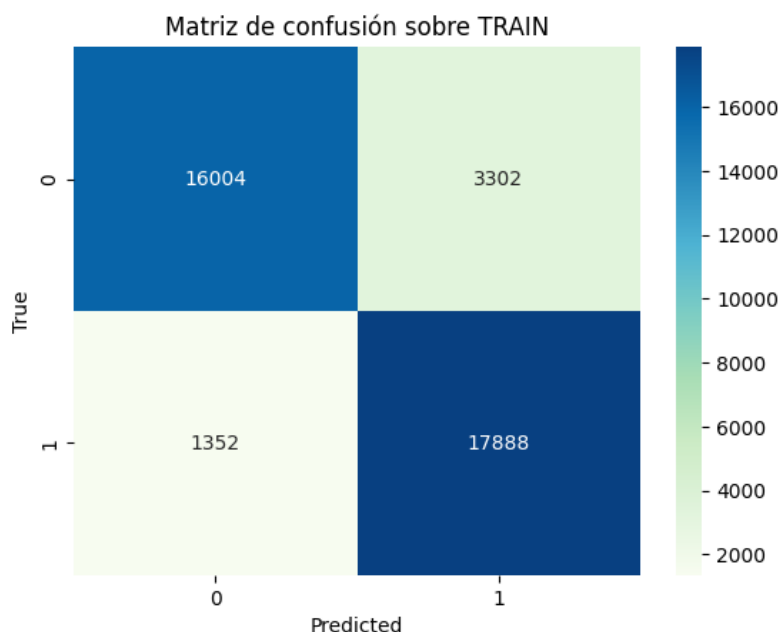


Figura 3: Matriz de confusión TRAIN

Hay una caída moderada en todas las métricas respecto al entrenamiento, lo que sugiere que hubo un ligero sobreajuste. El modelo sigue generalizando razonablemente bien. La clase positiva mantiene buen recall, lo que puede ser útil si tu objetivo es captar la mayor cantidad de críticas positivas, aunque con algo de ruido.

Conclusiones sobre los resultados obtenidos

El modelo muestra un rendimiento general sólido, con una precisión global del 83% en el conjunto de test. Lo que nos indica una capacidad adecuada para distinguir entre críticas positivas y negativas en datos no vistos.

Se observa un ligero sobreajuste, evidenciado por la caída en las métricas de test. Sin embargo, consideramos que esta diferencia no compromete la capacidad de generalización del modelo.

El modelo tiende a priorizar el recall en la clase positiva, lo que implica que logra identificar la mayoría de las críticas favorables, aunque con una menor precisión. Este comportamiento puede ser deseable si el objetivo es maximizar la detección de reseñas positivas, por ejemplo, para destacar contenido en plataformas de recomendación.

En contraste a esto último, la clase negativa presenta mayor precisión, lo que sugiere que cuando el modelo predice una crítica como negativa, lo hace con alta confiabilidad.

4. XGBoost

XGBoost es un algoritmo basado en boosting de árboles de decisión que se destaca por su capacidad para capturar relaciones no lineales y patrones complejos en datos de alta dimensionalidad. En nuestro caso, lo utilizamos para la clasificación binaria del sentimiento a partir de reseñas en español ya preprocesadas.

A diferencia de los modelos anteriores, XGBoost requiere una representación numérica densa y consistente del texto. Por este motivo, se evaluaron dos metodologías de vectorización: **TF-IDF** y **Bag-of-Words (BOW)**. La totalidad del pipeline de XGBoost —preprocesamiento, vectorización y búsqueda de hiperparámetros— se implementó en el notebook correspondiente al modelo.

4.1. Preprocesamiento y vectorización

Se utilizó el mismo preprocesamiento profundo desarrollado en la primera etapa del trabajo: limpieza mediante expresiones regulares, tokenización con spaCy en español y lematización de cada token.

A partir de ese texto lematizado se construyeron dos espacios vectoriales:

- **TF-IDF con unigramas y bigramas**, filtrando términos muy infrecuentes ($\text{min_df} = 10$) y términos demasiado comunes ($\text{max_df} = 0.9$).
- **Bag-of-Words (BOW)** con los mismos filtros de frecuencia, lo que permite comparar directamente ambas representaciones.

Ambas matrices resultantes son de muy alta dimensionalidad, por lo que XGBoost representa una buena elección al manejar de forma eficiente espacios dispersos.

4.2. Entrenamiento del modelo base

Antes de realizar una optimización más exhaustiva, se entrenó un **modelo base** con hiperparámetros razonables. El objetivo de esta etapa fue:

- validar la coherencia del pipeline texto \rightarrow vectorización \rightarrow modelo;
- obtener una primera referencia de desempeño;
- evaluar tiempos de entrenamiento para decidir el tamaño de la búsqueda de hiperparámetros.

En el conjunto de validación, el modelo base obtuvo:

- **F1 = 0,844** con TF-IDF;
- **F1 = 0,833** con BOW.

La cercanía entre ambos resultados indica que el preprocesamiento semántico fue exitoso y que XGBoost es robusto ante distintas representaciones vectoriales.

4.3. Búsqueda de hiperparámetros

Para evitar entrenamientos de varias horas, se limitó la búsqueda aleatoria a:

- **5 combinaciones** de hiperparámetros elegidas al azar ($\text{n_iter} = 5$);
- **validación cruzada de 4 folds** ($\text{cv} = 4$).

Esto implica entrenar un total de 20 modelos por tipo de vectorizador. Se exploraron hiperparámetros clásicos de XGBoost: número de árboles, profundidad máxima, tasa de aprendizaje, regularización L1/L2, submuestreo de filas y columnas y parámetro γ de complejidad.

Los mejores valores obtenidos fueron:

- **TF-IDF**: F1 (CV) $\approx 0,847$;
- **BOW**: F1 (CV) $\approx 0,839$.

En ambos casos, la mejora respecto al modelo base fue marginal (del orden de 0,003 puntos en TF-IDF), lo cual sugiere que los parámetros por defecto ya eran razonables y que la representación lematizada facilita la generalización.

4.4. Entrenamiento final y predicciones en Kaggle

Con los mejores hiperparámetros identificados, se reentrenó el modelo usando **todo el conjunto de entrenamiento** y se vectorizaron nuevamente tanto *train* como *test*.

Los resultados finales en Kaggle fueron:

- **XGBoost TF-IDF**: score $\approx 0,70365$;
- **XGBoost BOW**: score $\approx 0,71039$.

Aunque ambos modelos superan el rendimiento de Random Forest, no alcanzan el desempeño del ensamble.

Los archivos de submission generados automáticamente corresponden a:

- `submission_xgboost_tfidf.csv`,
- `submission_xgboost_bow.csv`.

Los modelos entrenados se almacenaron como:

- `modelo_xgboost_tfidf.joblib`,
- `modelo_xgboost_bow.joblib`.

5. Red Neuronal

Implementamos un modelo de Red Neuronal que trabajará para procesar secuencias de texto, preprocesado y limpiado, con el fin de, mediante la sucesión de épocas, alteración de parámetros y el aumento o disminución de neuronas, obtener un resultado probabilístico a cerca de cuan positivo es una crítica de cinematográfica.

5.1. Preprocesamiento

En esta sección definiremos todos aquellos procedimientos realizados antes de ejecutar el entrenamiento de la red en cuestión.

Lo primero que realizamos fué la creación de nuestro vocabulario, esto en base a una normalización del texto bajo las normas de la librería **Unicode** y de la librería **regex**, las cuales nos permitió eliminar acentos, signos de admiración o pregunta y todo aquel caracter que no sea alfanumérico, tambien concatenaciones caracteres que tienen distribuciones repetitivas, como múltiples vocales iguales (las cuales son utilizadas para hacer énfasis en palabras particulares o comentarios efucivos), puntos suspensivos, más de un espacio entre palabras, etc. y por último reducimos todos los caracteres a minúscula. Con todas las críticas normalizadas bajo las mismas reglas, nos quedaremos con aquellos tokens (definimos token como una palabra normalizada) que tengan al menos 2 apariciones y a esto lo llamaremos **vocabulario**.

Definimos un índice para cada uno de los tokens distintos, dejando el valor 0 y el 1 reservados para padding (0) y palabras desconocidas (1), este último será aplicable para el procesamiento del vocabulario del archivo de test, el cual podría contener tokens que no existen dentro de nuestro vocabulario, ya que no existen dentro del archivo de entrenamiento.

También necesitamos un **corpus**, el cual consiste en transformar las críticas (conjunto de palabras) en vectores (la representación de las mismas palabras, convertidas en sus respectivos valores numéricos).

Con el **corpus** y una nueva variable, que llamaremos **MAX LEN**, asociado al percentil 95 del total de las longitudes de las criticas, estandarizaremos el largo de cada una de las mismas, rellenando (padding) o truncando si se excede del valor **MAX LEN**.

Una vez estandarizados los tamaños de las críticas, el siguiente paso es definir nuestra red neuronal y entrenarla.

5.2. Elección de arquitectura

Una vez realizado el preprocesamiento de nuestros datos de entrenamiento, definimos nuestra red neuronal, dividiéndola en 3 partes:

- **Embedding:** Esta capa es fundamental y debe existir debido a que su función es agrupar palabras similares en grupos homogéneos y que dichos grupos sean lo más heterogéneos entre sí, unos de otros.
- **LSTM:** Dado que procesaremos secuencias, es pertinente utilizar una RNN, por lo tanto nos decantamos por el uso de este tipo respecto a una GRU, ya que es ideal para procesar con mayor precisión, a costa de mayor utilización de recursos. En caso de haber tenido recursos o tiempo más acotados y/o reseñas menos complejas o más reducidas, la elección hubiera cambiado.
- **Dense:** Finalmente, nos encontramos con la capa necesaria para poder conectar todo lo previamente calculado, indispensable para obtener un resultado concreto, luego del todo el procesamiento previo de las anteriores capas.

No menos importante, en medio de estos 3 bloques de capas bien definidos, incorporamos la utilización de capas de "dropout", con el fin de "apagar" neuronas aleatoriamente con cierta probabilidad, para evitar el overfitting dentro del modelo, y ayudar a la generalización.

5.3. Mejor modelo

Nuestros mejores resultados se encontraron, luego de reiteradas combinaciones de parámetros, aplicadas a arquitecturas como la anterior mencionada. Dentro de nuestro modelo:

```
1 embedding_dim = 128
2 LSTM_UNITS = 64
3
4 model = keras.Sequential([
5     layers.Embedding(input_dim=vocab_size,
6                      output_dim=embedding_dim,
7                      input_length=MAX_LEN,
8                      mask_zero=True),
9
10    layers.Dropout(0.4),
11
12    layers.Bidirectional(layers.LSTM(LSTM_UNITS, return_sequences=False)),
13
14    layers.Dropout(0.4),
15
16    layers.Dense(32, activation='relu'),
17    layers.Dense(1, activation='sigmoid')
18 ])
```

Podemos observar la alta probabilidad (0.4) de "apagar" neuronas en ambas capas intermedias de dropout, este fue el valor utilizado, debido a que aumentarlo no mejoraba los resultados a lo largo de las épocas (incluso redujo los resultados dentro del leaderboard de Kaggle), y reducirlo aumentó la velocidad a la que se llegaba al overfitting entre las épocas.

La utilización de LSTM es dentro de una capa bidireccional, ya que esto nos permite procesar no solo de la primera a la última palabra, sino al revés también, ya que dentro del idioma español es muy común la ubicación de adjetivos que alteran el significado o importancia de un sustantivo, por delante de él (por ejemplo: "Fue una película muy mala" y "Fue una muy mala película", ambas tienen exactamente la misma efucividad y sentimiento, pero el adjetivo que afecta se encuentra por delante en la primera y por detrás en la segunda).

Por último, aplicamos 2 capas densas, con diferentes funciones de activación y diferentes tamaños de salidas, esto es con el fin de tener en primer lugar una capa densa que nos de cierto nivel de abstracción, gracias a la transformación la cantidad de inputs presentes (64) a un nuevo output de solo 32 valores, sobre los datos otorgados por la capa LSTM y con una función de activación ReLU, distinta a la Sigmoidea de la 2da capa para combinar la clasificación de ambas capas en lugar de avocarnos a una única metodología de activación para nuestra capa totalmente conectada. Por último y necesaria, la capa densa con 1 único output para converger todo lo procesado previamente en una única salida, probabilística, que utilizaremos para denotar de forma binaria si fué una crítica positiva o negativa en base a un umbral situado en 0.5.

5.4. Aclaraciones sobre el modelo

Algunas aclaraciones necesarias sobre nuestra implementación son:

- La utilización de una métrica custom (`F1_custom`), creada debido a que la clasificación nativa de Keras nos dió problemas con la dimensionalidad de los resultados.
- Nuestros primeros intentos fueron realizados con un modelo más sencillo que el expuesto previamente, compuesto únicamente por 3 capas:
 - Embedding: con parámetros mucho más reducidos, en comparación con los usados en nuestro mejor modelo).
 - Bidireccional GRU: utilizado para un entrenamiento más veloz y para tener una primera apreciación acerca del funcionamiento y entrenamiento de una red neuronal.
 - Densa: una única capa que tiene como output un solo valor que será el resultado. No se encuentra presente nada con respecto a la abstracción.
- Se realizaron intentos con entrenamientos en partes (con `checkpoints`), debido a la elevada cantidad de épocas que queríamos entrenar, pero finalmente, incrementar en exceso dicha cantidad, solo nos dio resultados peores. Estos resultados se atribuyen a un overfitting por parte del modelo.
- Se utilizó "early stop" para monitorear el progreso de entrenamiento de nuestro modelo, y destacó su uso cuando fueron entrenadas más de 6 épocas, ya que dependiendo los parámetros impuestos en las capas de la red, nos era muy útil para evitar entrenar épocas de más innecesariamente.

6. Ensamble

El enunciado requería la construcción de un **modelo de ensamble utilizando al menos tres clasificadores**, con el fin de evaluar si la combinación de modelos diversos mejora el rendimiento individual. Para cumplir este objetivo, se utilizaron tres modelos preentrenados por el grupo:

- Naïve Bayes optimizado;
- Random Forest optimizado;
- XGBoost optimizado con TF-IDF.

Todos los modelos fueron cargados directamente desde Google Drive sin reentrenamiento. El ensamble se implementó en una notebook separada, reutilizando exactamente los pesos y vectorizadores entrenados previamente.

Homogeneización del preprocesamiento

Dado que los modelos fueron entrenados con vectorizadores distintos, surgió un desafío metodológico importante: *¿cómo crear un ensemble si cada modelo requiere una representación diferente del texto?*

La solución adoptada fue mantener intacto el pipeline de cada modelo y hacer que cada uno procese el mismo texto dentro de su propio espacio vectorial:

- Naïve Bayes y Random Forest trabajan sobre texto preprocesado con stemming y TF-IDF;
- XGBoost trabaja sobre texto lematizado con spaCy y vectorizado mediante TF-IDF con unigramas y bigramas.

De esta forma se respeta exactamente el entrenamiento original y se evita introducir inconsistencias entre representaciones.

Soft voting (promedio de probabilidades)

Para combinar las predicciones finales se eligió un esquema de **soft voting**, donde cada modelo aporta su probabilidad estimada de que una reseña sea positiva. La probabilidad final se obtiene como el promedio simple de las probabilidades individuales:

$$\hat{y} = \mathbb{I} \left(\frac{p_{NB} + p_{RF} + p_{XGB}}{3} > 0,5 \right),$$

donde p_{NB} , p_{RF} y p_{XGB} son las probabilidades de clase positiva estimadas por Naïve Bayes, Random Forest y XGBoost respectivamente, y $\mathbb{I}(\cdot)$ es la función indicadora.

Este esquema tiene varias ventajas:

- pondera modelos más seguros (probabilidades cercanas a 0 o 1),
- suaviza errores individuales,
- combina modelos conceptualmente diferentes (lineales, basados en árboles y boosting).

Resultados del ensemble

El ensemble se evaluó sobre el mismo conjunto de validación utilizado para los modelos individuales y posteriormente se generó un archivo de submission para Kaggle. El resultado final fue:

- **Ensamble Soft Voting:** score $\approx 0,75296$ en Kaggle.

Este valor supera claramente a los modelos individuales:

- mejora de aproximadamente +0,049 puntos respecto a XGBoost TF-IDF
- mejora de aproximadamente +0,033 puntos respecto a Random Forest TF-IDF
- mejora de aproximadamente +0,003 puntos respecto a Naive Bayes TF-IDF

Estos resultados confirman empíricamente que combinar modelos heterogéneos puede mejorar la performance, incluso cuando cada modelo utiliza una representación distinta del texto.

Resumen comparativo

La Tabla 5 resume los puntajes obtenidos en Kaggle para los modelos considerados.

Modelo	Score en Kaggle
XGBoost TF-IDF	0.70365
Random Forest TF-IDF	0.72040
Naive Bayes TF-IDF	0.74970
Ensamble (soft voting)	0.75296

Cuadro 5: Comparación de desempeño entre los modelos considerados y el ensemble final.

7. Resultados

7.1. Cuadro de resultados

Con el propósito de evaluar de manera comparativa el desempeño de los modelos entrenados, se elaboró un cuadro de resultados que reúne las principales métricas obtenidas en el conjunto de test, calculadas en función de las reseñas clasificadas como positivas por cada modelo y expresadas con 4 decimales significativos.

Además, se incorpora el desempeño obtenido en Kaggle, donde la plataforma evalúa únicamente el 50 % de las predicciones enviadas y calcula la métrica F1-score sobre ese subconjunto.

A continuación, se presenta el cuadro con la información recopilada.

Modelo	F1-Test	Precisión Test	Recall Test	Accuracy Test	Kaggle
Bayes Naive	0.8816	0.8772	0.8860	0.8810	0.7497
Random Forest	0.8360	0.8080	0.8660	0.8300	0.7204
XgBoost	0.8700	0.8400	0.9100	0.8700	0.7037
Red Neuronal	0.8677	0.9270	0.8222	0.8786	0.7492
Ensamble	0.9100	0.8800	0.9300	0.9000	0.7530

7.2. Modelo seleccionado como mejor predictor

8. Conclusiones Generales

¿Fue útil realizar un análisis exploratorio de los datos?

El análisis exploratorio permitió identificar aspectos relevantes del corpus y guiar decisiones de limpieza, como el filtrado de palabras en inglés aplicado en el modelo Random Forest. Si bien estas acciones contribuyeron a mejorar la coherencia lingüística del conjunto de entrenamiento, su impacto en las métricas de clasificación fue limitado, por lo que puede concluirse que el análisis exploratorio fue útil pero no determinante para el rendimiento final de los modelos.

¿Las tareas de preprocesamiento ayudaron a mejorar la performance de los modelos?

Sí, definitivamente. Las tareas de preprocesamiento fueron determinantes para transformar un modelo genérico en uno robusto, mejorando significativamente su performance. Si bien en cada modelo realizamos diferentes técnicas de preprocesamiento, entre las intervenciones que más impacto tuvieron destacan:

Stopwords: Al conservar las negaciones (ej: "no", "sin"), permitimos que el modelo distinga el sentido real de frases como "no recomiendo", corrigiendo el error de los modelos iniciales que invertían el sentimiento.

Limpieza y Estandarización: La conversión a minúsculas junto con la eliminación y caracteres no alfabéticos redujo drásticamente el ruido del dataset. Esto evitó la duplicidad de términos y permitió que el algoritmo se enfocara puramente en el contenido textual relevante.

¿Cuál de todos los modelos obtuvo el mejor desempeño en TEST?

El modelo que mejor desempeño obtuvo en Test (en 4 de las 5 métricas calculadas, solo superada por la Red Neuronal en "Precision") fue el de Ensamble, esto se puede deber a que contrarrestó el underfitting existente en el modelo de Random Forest (el cual obtuvo las peores métricas en 4 de las 5 existentes).

¿Cuál de todos los modelos obtuvo el mejor desempeño en Kaggle?

El modelo que tuvo un mejor desempeño en Kaggle fue el Ensamble, alcanzando un score de 0.75296 puntos.

¿Cuál fue el modelo más sencillo de entrenar y más rápido?

El modelo más sencillo y rápido de entrenar fue el de Naive Bayes.

Suponemos que es porque su arquitectura se basa en el cálculo de probabilidades a partir de conteos. Esto lo convierte en el modelo con menor costo computacional entre todas las alternativas evaluadas.

¿Es útil en relación al desempeño obtenido?

A pesar de ser el modelo más simple y rápido, Naive Bayes resultó sorprendentemente competitivo. Su puntaje público en Kaggle (0.74970) quedó prácticamente al mismo nivel que la red neuronal (0.74924) y muy cerca del ensamble (0.75296), a pesar de que estos modelos son significativamente más complejos y costosos de entrenar.

Esto demuestra que la simplicidad de Naive Bayes no implica necesariamente un bajo rendimiento para nuestro caso. Por lo tanto, considerando su velocidad, facilidad de implementación y desempeño, Naive Bayes constituye una opción altamente útil y eficiente dentro del conjunto de modelos evaluados.

¿Cree que es posible usar su mejor modelo de forma productiva?

Sí, el mejor modelo podría utilizarse de forma productiva, ya que sus métricas reflejan una buena capacidad para clasificar reseñas en español. Sin embargo, es importante considerar que fue entrenado mayormente (o exclusivamente, como en el caso de Random Forest) sobre textos en este idioma, por lo que en un entorno real podría enfrentar limitaciones si recibe reseñas con ruido, mezcla de lenguas o directamente en otro idioma. En tales casos, la clasificación perdería fiabilidad, ya que el modelo no dispone de representaciones adecuadas para vocabulario ajeno al entrenamiento.

¿Cómo podría mejorar los resultados?

Aunque es viable su uso productivo en el dominio definido como mencionamos anteriormente, se recomienda incorporar mecanismos de detección y filtrado de idioma para garantizar resultados robustos.

9. Tareas realizadas

A continuación se resumen las tareas asumidas por cada integrante del equipo y el tiempo promedio semanal invertido en estas.

Integrante	Tarea	Prom. Hs Semana
Franco Rodriguez	Trabajó en el modelo Random Forest	3
Blas Chuc	Trabajó en el modelo de Red Neuronal	5
Helen Chen	Trabajó en el Bayes Naïve	3
Tomas Caporaletti	Trabajó en el modelo XGBoost y el ensamble	3

Cuadro 6: Distribución de tareas y carga semanal promedio