Práctica 2: Clasificación de discurso de odio en Twitter/X

UACM

Universidad Autónoma de la Ciudad de México

Nada humano me es ajeno

Alumno: Garcia Mercado Edwar Ezequiel

Matrícula: 18-003-1925

Profesor: Sabino Miranda

Materia: Redes Neuronales

1. Introducción

La detección de discurso de odio en redes sociales se ha convertido en un reto importante para las plataformas digitales. En **Twitter/X**, donde los mensajes son breves y se difunden rápidamente, identificar contenido discriminatorio contra grupos como inmigrantes y mujeres requiere soluciones automatizadas que puedan procesar grandes volúmenes de datos.

Esta práctica se basa en el **dataset HatEval 2019**, una iniciativa internacional que reunió tweets en español e inglés específicamente anotados para detectar discurso de odio. El conjunto de datos incluye **5,000 ejemplos en español** y **10,000 en inglés**, cada uno etiquetado como *"contiene discurso de odio"* o *"no contiene discurso de odio"*.

El objetivo principal es **implementar una red neuronal desde cero** para abordar este problema de clasificación. Pero más allá de simplemente construir el modelo, nos interesa entender cómo las decisiones que tomamos afectan su rendimiento:

- ¿Qué tipo de preprocesamiento de texto funciona mejor?
- ¿Las representaciones **TF** o **TF-IDF** son más efectivas?
- ¿Cómo influye la arquitectura de la red en los resultados?
- ¿Qué parámetros de entrenamiento optimizan el aprendizaje?

Estas preguntas son clave no solo para este proyecto, sino para comprender los fundamentos del **aprendizaje profundo aplicado al procesamiento de lenguaje natural**.
A través de experimentos sistemáticos con diferentes configuraciones, podremos identificar las **mejores prácticas** para este tipo de problemas.

2. Desarrollo

2.1 Implementación del Perceptrón Multicapa

La implementación de la red neuronal se realizó mediante la clase MLP, desarrollada completamente desde cero para esta práctica. Esta clase incorpora todas las funcionalidades necesarias para entrenar y evaluar un Perceptrón Multicapa.

Características Principales de la Implementación

Arquitectura:

- **Entradas**: Variable, según la dimensionalidad del dataset
- **Capa oculta**: Configurable en número de neuronas
- **Capa de salida**: 1 neurona con función sigmoide para clasificación binaria
- **Función de activación**: Sigmoide en todas las capas
- **Función de error**: Error Cuadrático Medio (MSE)

Mecanismos de Aprendizaje:

- **Forward Propagation**: Calcula las salidas de cada capa aplicando la función sigmoide a la combinación lineal de entradas y pesos
- **Backward Propagation**: Implementa el algoritmo de descenso de gradiente para actualizar los pesos basándose en el error calculado
- **Inicialización de Pesos**:
    - Soporta Xavier y Normal, seleccionable con el parámetro inicializacion
    - Los sesgos se inicializan en cero

Hiperparámetros de entrenamiento:

- **Número de épocas**: definido por el parámetro epochs
- **Tasa de aprendizaje** (learning_rate): configurable, afecta la velocidad de actualización de pesos
- **Batch size** (batch_size): define los minibatches para entrenamiento estocástico
- **Random seed** (random_state): asegura reproducibilidad de los experimentos

2.2 Preprocesamiento de Texto

Se implementó un sistema modular de preprocesamiento que permite experimentar con tres niveles progresivos de limpieza de texto, tanto para español como para inglés.

Funciones de Preprocesamiento Implementadas

1. only_normalizar_texto()

- Eliminación de URLs, menciones y caracteres especiales
- Normalización de caracteres Unicode y eliminación de acentos
- Reducción de caracteres duplicados (máximo 2 consecutivos)
- Conversión a minúsculas
- Eliminación de espacios redundantes

2. normalizar_txt_sin_StopWords()

- Aplica todas las normalizaciones del nivel básico
- Eliminación de stopwords específicas del idioma
- **Ejemplo de transformación**:
  "El usuario publicó un tweet de odio" → "usuario publicó tweet odio"

3. normalizar_txt_sin_StopWords_mas_stemming()

- Aplica normalización y eliminación de stopwords
- Reducción de palabras a su raíz mediante Snowball Stemmer
- **Ejemplo de transformación**:
  "publicó tweets odiosos" → "public tweet odi"

Sistema de Selección Flexible

```
def mi_preprocesamiento(tipo_procesamiento):
    procesadores = {
        'only_normalizar_texto': only_normalizar_texto,
        'normalizar_txt_sin_StopWords': normalizar_txt_sin_StopWords,
        'normalizar_txt_sin_StopWords_mas_stemming':
normalizar_txt_sin_StopWords_mas_stemming
    }
    return procesadores[tipo_procesamiento]
```

Este diseño permitió evaluar sistemáticamente el impacto de cada nivel de preprocesamiento en el rendimiento final del modelo, manteniendo consistencia entre los experimentos en ambos idiomas.

2.3 Representaciones Vectoriales de Términos

Para capturar diferentes niveles de información lingüística en los textos preprocesados, se implementaron tres esquemas de representación de términos.

Esquemas de N-gramas

1. Unigramas (1-gram)

- Secuencias individuales de palabras que capturan vocabulario básico y términos clave
  Ejemplo: "odio", "mujeres", "inmigrantes"

2. Bigramas (2-gram)

- Pares consecutivos de palabras que capturan frases y contextos locales
  Ejemplo: "discurso odio", "contra mujeres", "odio inmigrantes"

3. Unigramas + Bigramas (1-gram + 2-gram)

- Combinación que captura tanto términos individuales como relaciones entre palabras adyacentes
  Ejemplo: "odio", "mujeres", "discurso odio", "contra mujeres"

2.4 Pesado de Términos: TF y TF-IDF

1. TF (Term Frequency)

- Calcula la frecuencia absoluta de cada término dentro de cada documento (tweet)
- Captura la importancia local de las palabras dentro de cada mensaje individual
- **Fórmula**:

$$tf(t,d) = frecuencia\ de\ t\ en\ el\ documento\ d$$

- **Ventaja**: Simple y rápido de computar
- **Limitación**: Puede dar mucho peso a términos muy frecuentes pero poco discriminativos

2. TF-IDF (Term Frequency – Inverse Document Frequency)

- Pondera la frecuencia de cada término considerando su distribución global en todo el corpus.
- Reduce el peso de términos comunes y aumenta el de términos raros pero discriminativos.
- **Fórmula**:

$$tfidf(t,d) = tf(t,d)\ x\ log\ (N\ /\ df(t)\ )$$

  - Donde N es el número total de documentos
  - df(t) es el número de documentos que contienen el término t

- **Ventaja**: Mejor capacidad para identificar términos característicos de cada clase

2.5 Estrategia de Muestreo Experimental

Diseño General

Se implementó una estrategia de generación secuencial e independiente para cada arquitectura de red neuronal, modificando manualmente el parámetro de neuronas en cada ejecución:

```
200 EXPERIMENTOS TOTALES
├── EJECUCIÓN 1: 64 neuronas
│   └── "combinaciones_64_neuronas.csv" (20 experimentos)
├── EJECUCIÓN 2: 128 neuronas
│   └── "combinaciones_128_neuronas.csv" (20 experimentos)
├── EJECUCIÓN 3: 256 neuronas
│   └── "combinaciones_256_neuronas.csv" (20 experimentos)
├── EJECUCIÓN 4: 512 neuronas
```

```
│    └── "combinaciones_512_neuronas.csv" (20 experimentos)
└── EJECUCIÓN 5: 1024 neuronas
    └── "combinaciones_1024_neuronas.csv" (20 experimentos)
```

Espacio de Parámetros por Configuración

Cada archivo contiene 20 combinaciones aleatorias que exploran:

Tabla 1. Hiperparametros utilizados para generar las combinaciones

Parámetro	Valores	Combinaciones
Inicialización	Normal, Xavier	2
Pesado	TF, TF-IDF	2
Representación	(1,1), (2,2), (1,2)	3
Preprocesamiento	3 técnicas	3
Learning Rate	0.01, 0.1, 0.5	3
Batch Size	16, 32, 64	3
Épocas	100	1

Espacio total por arquitectura: 2 × 2 × 3 × 3 × 3 × 3 = **324 combinaciones posibles**
Muestra seleccionada: 20 combinaciones aleatorias

2.6 Estrategia de Aleatorización

- Semillas únicas para cada configuración de neuronas
- Muestreo aleatorio simple del espacio completo de parámetros
- Archivos independientes para facilitar el procesamiento
- Reproducibilidad garantizada mediante semillas fijas

Ventajas del Enfoque Modular

1. **Control granular:** Cada arquitectura tiene su propio conjunto de experimentos
2. **Organización clara:** Archivos separados por tipo de red neuronal
3. **Ejecución flexible**: Posibilidad de correr configuraciones específicas
4. **Análisis independiente:** Evaluación separada por complejidad de modelo
5. **Reproductibilidad:** Fácil verificación y replicación por arquitectura

Aplicación en Experimentos Finales

Cada dataset utilizó los 5 archivos de configuración:

Dataset 1: 5 arquitecturas × 20 experimentos = **100 experimentos**
Dataset 2: 5 arquitecturas × 20 experimentos = **100 experimentos**

Total: **200 experimentos**

2.7 Optimizaciones Computacionales Implementadas

Limitación de Features a 10,000

Decisión de Diseño Preventiva:
Dada la naturaleza exploratoria de los **200 experimentos** y la necesidad de **ejecutar los experimentos y realizar el reporte**, se implementó una limitación de **10,000 features máximo**, basada en fundamentos técnicos sólidos.

Fundamentos Técnicos

```
vectorizador = Vectorizador(
    max_features=10000,
    ngram_range=config['representacion'],
    preprocessor=mi_preprocesamiento(config['preprocesamiento'])
)
```

Justificación del Límite

1. Prevención de Dimensionalidad Explosiva:

Las representaciones n-gram pueden generar más de 70,000 features, resultando en requerimientos computacionales excesivos para los 200 experimentos.

2. Ley de Rendimientos Decrecientes en NLP(Procesamiento de Lenguaje Natural):

- Los términos más frecuentes suelen contener la mayor parte de la información relevante.
- Las features raras tienden a aportar más ruido que señal.

3. Fundamentos en Literatura Científica:
La relación entre el número de ejemplos (muestras) y el número de características (dimensiones) es fundamental para la generalización del modelo. El problema de la “maldición de la dimensionalidad” surge cuando el número de características es excesivamente mayor que el número de muestras de entrenamiento [1]. En estos casos de

alta dimensionalidad, los datos se vuelven esparsos, ya que el volumen del espacio crece exponencialmente, y el modelo se vuelve extremadamente propenso a sobreajustarse (overfitting) [2].

La selección de características aborda este problema directamente al identificar y conservar solo las variables más relevantes, eliminando las irrelevantes y redundantes [3]. Al reducir la dimensionalidad, no solo se disminuye la carga computacional, sino que se incrementa la eficiencia del aprendizaje, la precisión predictiva y se reduce la complejidad de los resultados del modelo [4]. Esto se debe a que, con un espacio de características más reducido y relevante, el algoritmo puede aprender las relaciones subyacentes más importantes sin adaptarse al ruido. Esta estrategia es fundamental para reducir el riesgo de sobreajuste y asegurar que el modelo generalice de manera efectiva, especialmente en conjuntos de datos con un número limitado de ejemplos [3].

Estrategia de Muestreo de Features

- Se seleccionan los **10,000 términos más frecuentes**.
- Esto captura el **vocabulario central**.
- Excluye términos raros que pueden causar *overfitting*.
- Se alinea con el **principio de parsimonia** en aprendizaje automático.

Mecanismo de Parada Temprana (Early Stopping)

Complemento a la Optimización:
Para incrementar la eficiencia durante el entrenamiento se implementó un mecanismo de parada temprana:

- **Detención inteligente:** cuando no hay mejora en 10 épocas.
- **Prevención de overfitting:** evita la memorización del dataset.
- **Ahorro computacional:** reduce las épocas innecesarias.

La integración de estas optimizaciones constituye una **aproximación de ingeniería responsable**, que prioriza la **viabilidad experimental** sin comprometer el rigor científico, respaldada por prácticas reconocidas en la investigación de *machine learning* aplicado.

2.9 Entrenamiento y evaluación

2.9.1 Dataset en Español

1. Descripción general de los experimentos

Para el dataset en español se realizaron **100 experimentos en total**, distribuidos en **20 ejecuciones por cada configuración del número de neuronas en la capa oculta** (64, 128, 256, 512 y 1024). En cada experimento, los **hiperparámetros restantes** como la tasa de aprendizaje, el tipo de inicialización de pesos, la representación de términos y las técnicas de procesamiento del texto se **seleccionaron de manera aleatoria**, con el fin de explorar diferentes combinaciones posibles y observar su impacto en el rendimiento del modelo. El conjunto de datos se dividió en **4,500 ejemplos para entrenamiento y 500 para prueba**, garantizando una evaluación equilibrada del modelo. Posteriormente, de cada grupo de 20 ejecuciones se **seleccionó el mejor resultado** según las métricas de **Accuracy** y **F1-score**, con el propósito de analizar el desempeño más representativo para cada configuración de neuronas y determinar la arquitectura con el **mejor equilibrio entre rendimiento y costo computacional**.

2. Resultados obtenidos

Los **cinco mejores resultados** seleccionados, correspondientes al mejor desempeño de cada configuración del número de neuronas, se muestran en la siguiente tabla:

Tabla 2. Las mejores Combinaciones por número de neuronas

Neuronas	Inicialización	Learning rate	Representación	Procesamiento	Accuracy	F1-score	Tiempo (s)
64	Xavier	0.01	TF (1,2)	sin StopWords	0.788	0.785	217
128	Xavier	0.01	TF (1,2)	normalizado	0.81	0.807	542
256	Xavier	0.01	TF (2,2)	normalizado	0.77	0.764	657
512	Normal	0.1	TF (1,1)	normalizado	0.734	0.729	1507
1024	Normal	0.1	TF-IDF (1,2)	sin StopWords + stemming	0.74	0.735	1355

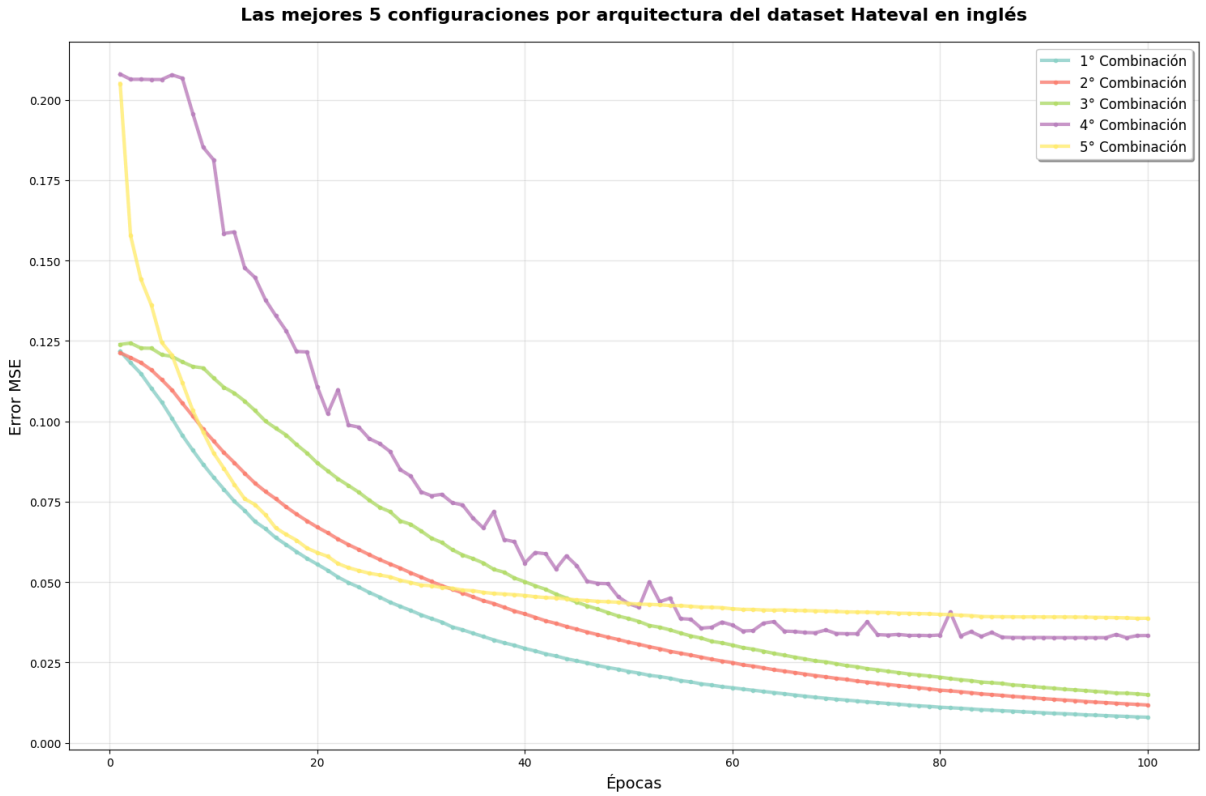

Figura 1. Evolución del Error (MSE) durante el entrenamiento para las 5 mejores configuraciones del modelo en el dataset en español. Se observa que las configuraciones con arquitecturas de 64 y 128 neuronas (líneas azul y naranja) presentan una convergencia más rápida y estable, alcanzando los menores valores de error final. Las curvas muestran que el modelo óptimo (128 neuronas) mantiene una tendencia descendente constante, mientras que configuraciones más complejas (512 y 1024 neuronas) exhiben mayor volatilidad en su aprendizaje.

Tabla 3. Los 5 mejores experimentos (dataset en español)

Combi	Neur.	Inicializ.	LR	Batch	Pesado	N-grama	Preprocesamiento	Error	Tiempo (s)	Accuracy	F1-score
1	128	Xavier	0.01	16	TF	(1,2)	Solo normalización	0.0079	542.6	**0.810**	**0.807**
2	64	Xavier	0.01	16	TF	(1,2)	Normalizar texto sin StopWords	0.0117	**217.0**	0.788	0.785
3	256	Xavier	0.01	16	TF	(2,2)	Solo normalización	0.0149	657.3	0.770	0.764
4	1024	Normal	0.1	32	TF-IDF	(1,2)	Sin StopWords + Stemming	0.0333	1355.4	0.740	0.735
5	512	Normal	0.1	16	TF	(1,1)	Solo normalización	0.0387	1507.5	0.734	0.729

4. Interpretación y análisis

- **Mejor desempeño global:** La red con **128 neuronas** obtuvo el mejor resultado (Accuracy = 0.81, F1 = 0.8067), mostrando un equilibrio adecuado entre rendimiento y costo computacional.

- **Efecto del número de neuronas:** Aumentar el número de neuronas no implicó una mejora constante en el rendimiento, a partir de 256 neuronas, el modelo presentó **una ligera disminución en las métricas** y un **incremento considerable en el tiempo de entrenamiento**.

- **Inicialización Xavier:** En los tres primeros casos (64, 128 y 256 neuronas), la inicialización Xavier permitió una **convergencia más estable** y mejores resultados que la inicialización normal empleada en las configuraciones de 512 y 1024 neuronas.

- **Preprocesamiento del texto:** Las mejores métricas se alcanzaron cuando el texto fue **normalizado** y, en algunos casos, se eliminaron las **stopwords**, lo que sugiere que la limpieza del texto ayuda al modelo a concentrarse en los términos más informativos.

- **Representación de términos:** La representación **TF con unigramas+bigramas (1,2)** se mostró más efectiva que las configuraciones basadas en unigramas o TF-IDF, lo que indica que la combinación de palabras consecutivas mejora la captura de contexto semántico en español.

2.9.2 Análisis general de resultados Dataset Español (100 experimentos)

Durante la experimentación se realizaron **100 ejecuciones** variando arquitecturas, tasas de aprendizaje, funciones de inicialización, representaciones de términos, y diferentes tipos de procesamiento textual. A continuación se presenta un análisis detallado de las principales variables que influyeron en el rendimiento del modelo.

1. Inicialización de Pesos

En el total de 100 experimentos se emplearon dos estrategias de inicialización: **Normal** (56%) y **Xavier** (44%). Aunque Xavier obtuvo los **mejores resultados individuales**, la inicialización **Normal** fue la más utilizada y alcanzó un mayor número de configuraciones con buen desempeño general. Xavier tendió a destacar en arquitecturas pequeñas o medianas (64 y 128 neuronas), mientras que Normal mostró un comportamiento más estable en arquitecturas grandes (256, 512 y 1024 neuronas).

La inicialización Normal fue dominante en frecuencia y estabilidad, pero Xavier alcanzó el mejor rendimiento global (F1 = 0.8067). Se recomienda Xavier para modelos de baja a media complejidad y Normal para configuraciones más grandes.

2. Representación de Términos

Se utilizaron dos esquemas principales: **TF** y **TF-IDF**. En los resultados se observó que **TF presentó un comportamiento más robusto**, logrando mayores valores de F1-score promedio y una mejor estabilidad en la mayoría de los experimentos.
TF-IDF, si bien es conceptualmente más informativo, mostró una mayor sensibilidad a los hiperparámetros del modelo.

La representación TF fue la más efectiva y estable para este dataset. El uso de TF-IDF no ofreció ventajas significativas y, en algunos casos, aumentó la variabilidad de los resultados.

3. Procesamiento del Texto

Se evaluaron tres estrategias principales:

Los resultados mostraron que el **procesamiento más simple (only_normalizar_texto)** obtuvo los mejores valores promedio de F1-score. En cambio, las versiones con eliminación de stopwords y stemming redujeron ligeramente el desempeño, probablemente por la pérdida de información semántica relevante.

Tabla 4. Impacto del preprocesamiento de texto en el F1-score para el dataset en español

Procesamiento	F1 promedio
only_normalizar_texto	0.78–0.81
normalizar_txt_sin_StopWords	0.75–0.77
normalizar_txt_sin_StopWords_mas_stemming	0.68–0.70

La estrategia de normalización simple fue la más adecuada. Los procesos de eliminación de palabras o raíces excesivos disminuyeron la capacidad del modelo para captar el contexto lingüístico.

4. Rango de N-Gramas

Se emplearon principalmente tres rangos de representación: **(1,1)**, **(1,2)** y **(2,2)**. Los experimentos confirmaron que el rango **(1,2)** (unigramas y bigramas) ofreció el mejor equilibrio entre contexto y generalización. El rango (2,2), al excluir unigramas, redujo el rendimiento, mientras que (1,1) fue más estable pero menos expresivo.

Tabla 5. Comparación del rendimiento (F1-score) según el rango de n-gramas en español

N-grama	F1 promedio
(1,1)	~0.73
(1,2)	~0.78–0.81
(2,2)	~0.70

La representación (1,2) fue la más efectiva para este conjunto, proporcionando un contexto textual suficiente sin comprometer la generalización del modelo.

5. Learning Rate y Batch Size

Se probaron distintas combinaciones de tasa de aprendizaje y tamaño de lote. Los valores más efectivos fueron **learning rate = 0.01** y **batch size entre 16 y 32**. Las configuraciones con tasas más altas (0.1 o 0.5) mostraron comportamientos inestables y menor rendimiento, indicando posibles problemas de convergencia o sobreajuste.

Tabla 6. Influencia del Learning Rate y Batch Size en el rendimiento del modelo (F1-score)

Learning Rate	Batch Size	F1 promedio
0.01	16–32	~0.78–0.81
0.1	16	~0.73
0.5	16–32	<0.65

Las mejores configuraciones surgieron con LR=0.01 y batches pequeños. Estas condiciones permitieron un aprendizaje controlado y una propagación de gradientes estable.

6. Tiempo de Entrenamiento y Eficiencia Computacional

El tiempo de entrenamiento aumentó exponencialmente con el número de neuronas y la complejidad de la representación textual. Los modelos con 128 neuronas ofrecieron el **mejor rendimiento por unidad de tiempo**, mientras que las redes de 512 y 1024 neuronas incrementaron el costo computacional sin mejoras relevantes en F1.

Tabla 7. Relación entre arquitectura de red, tiempo de entrenamiento y rendimiento (F1-score)

Arquitectura	Tiempo (s)	F1 promedio
64–128	200–500	Alta (~0.80)
256	650–700	Media (~0.76)
512–1024	>1300	Baja (~0.73)

En términos de eficiencia, los modelos medianos (128 neuronas, batch 16–32, LR=0.01) son los más recomendables, ofreciendo alta precisión con bajo costo computacional.

7. Conclusión

El conjunto de 100 experimentos evidencia que los mejores resultados se alcanzan al equilibrar la complejidad del modelo con una inicialización y parametrización adecuadas. Las configuraciones que combinan **inicialización Xavier**, **representación TF (1,2)**, **preprocesamiento simple**, **learning rate 0.01**, y **batch size pequeño (16–32)** alcanzaron los mayores valores de desempeño. En particular, el modelo con **128 neuronas** logró el **mejor F1-score (0.8067)** y **accuracy (0.81)**, confirmando que una arquitectura intermedia y bien calibrada produce la mejor generalización en este dominio textual.

2.9.4 Experimento complementario para validar la limitación de features

1. Configuración Base del Modelo

Todos los experimentos se realizaron usando la misma configuración base, la cual había mostrado un buen equilibrio entre rendimiento y eficiencia:

- **Neuronas:** 128
- **Inicialización:** Xavier
- **Normalización:** Ninguna
- **Learning rate:** 0.01
- **Batch size:** 16
- **Ponderación de términos:** TF
- **Representación de términos: Unigramas+bigramas** (1, 2)
- **Procesamiento de texto:** Normalización básica del texto

Esta configuración se eligió porque, en pruebas anteriores, ofreció buenos resultados sin exigir demasiado tiempo de entrenamiento ni recursos de cómputo.

2. Resultados Experimentales: Cambiando Épocas y Features

Tabla 8. Comparación de rendimiento y tiempo de entrenamiento según épocas y número de características

Épocas	Features	Tiempo (s)	Accuracy	F1-Score
100	10,000	542.6	0.810	0.807
100	71,171	3,975.9	0.810	0.807
500	10,000	1,875.8	0.808	0.804
500	71,171	16,460.1	0.814	0.811

3. Análisis de los Resultados

Los resultados confirman que **limitar el número de features a 10,000** es una **decisión muy acertada**, ya que mantiene el rendimiento del modelo mientras reduce drásticamente el tiempo de entrenamiento.

Cuando se entrenó con 100 épocas, tanto el modelo reducido (10,000 features) como el modelo completo (71,171 features) lograron exactamente la misma precisión y F1-score

(0.810 y 0.807). Sin embargo, el tiempo de ejecución fue **casi siete veces menor** en el modelo reducido (542.6 s vs. 3,975.9 s). En otras palabras, se logró **la misma calidad de predicción con un 86% menos de tiempo**.

Por otro lado, **aumentar el número de épocas** no mejoró los resultados. De hecho, con 10,000 features, pasar de 100 a 500 épocas no trajo mejoras y hasta se observó una ligera baja en las métricas. En el modelo con todas las features, el aumento sí generó una mejora mínima (de 0.810 a 0.814 en accuracy), pero el tiempo de entrenamiento se multiplicó por cuatro.

El caso más extremo 500 épocas y todas las features obtuvo el mejor resultado (Accuracy 0.814, F1 0.811), pero con un **tiempo de entrenamiento más de 30 veces mayor** que el modelo con 10,000 features y 100 épocas (16,460.1 s vs. 542.6 s). Esta diferencia muestra claramente que **el costo computacional no se justifica** por la pequeña ganancia en rendimiento.

4. Conclusión

Los experimentos dejan claro que **usar 10,000 features es la mejor opción** para mantener un equilibrio entre rendimiento y eficiencia. Con esta configuración, el modelo logra **prácticamente el mismo desempeño** que la versión completa, pero con un **tiempo de entrenamiento muchísimo menor**.

Reducir la cantidad de features no solo mejora la velocidad, sino que también hace que el modelo sea más fácil de implementar y escalar en entornos reales, donde los recursos y el tiempo suelen ser limitados.

Las pequeñas diferencias en las métricas (entre 0.3% y 0.4%) **no compensan el enorme aumento en tiempo de cómputo**. Por eso, entrenar con **10,000 features y 100 épocas** resulta la configuración más práctica y eficiente, logrando una **accuracy superior al 81%**, más que suficiente para la mayoría de aplicaciones.

Además, los resultados muestran que **entrenar más de 100 épocas no ofrece beneficios reales**, lo que confirma la solidez y eficiencia de esta configuración base.

2.9.5 Análisis de resultados Dataset en Inglés

1. Descripción general de los experimentos

Se utilizaron **las mismas configuraciones y rangos de hiperparámetros empleados en el dataset en español**, incluyendo combinaciones de tasa de aprendizaje, tipo de inicialización de pesos, representación de términos y técnicas de procesamiento de texto. **100 experimentos** distribuidos en **20 ejecuciones por cada configuración del número de neuronas en la capa oculta** (64, 128, 256, 512 y 1024).

El conjunto de datos se dividió en **9,000 ejemplos para entrenamiento y 1,000 para prueba**, garantizando una evaluación equilibrada del modelo.

De cada grupo de 20 ejecuciones se seleccionó el **mejor resultado según las métricas de Accuracy y F1-score macro**, con el fin de identificar la arquitectura más representativa y evaluar el equilibrio entre desempeño y costo computacional.

2. Resultados obtenidos

Los cinco mejores resultados, correspondientes al mejor desempeño alcanzado por cada configuración del número de neuronas, se resumen en la siguiente tabla:

Tabla 9. Las mejores Combinaciones por número de neuronas

Neuronas	Inicialización	Learning rate	Representación	Procesamiento	Accuracy	F1-score	Tiempo (s)
64	Xavier	0.1	TF-IDF (1,1)	only_normalizar_texto	0.717	0.713	200
128	Xavier	0.01	TF-IDF (1,2)	sin StopWords	0.732	0.730	205
256	Xavier	0.01	TF-IDF (1,2)	sin StopWords	0.727	0.723	707
512	Xavier	0.01	TF (1,1)	sin StopWords + stemming	0.708	0.702	1214
1024	Xavier	0.01	TF-IDF (1,2)	sin StopWords	0.727	0.721	6276

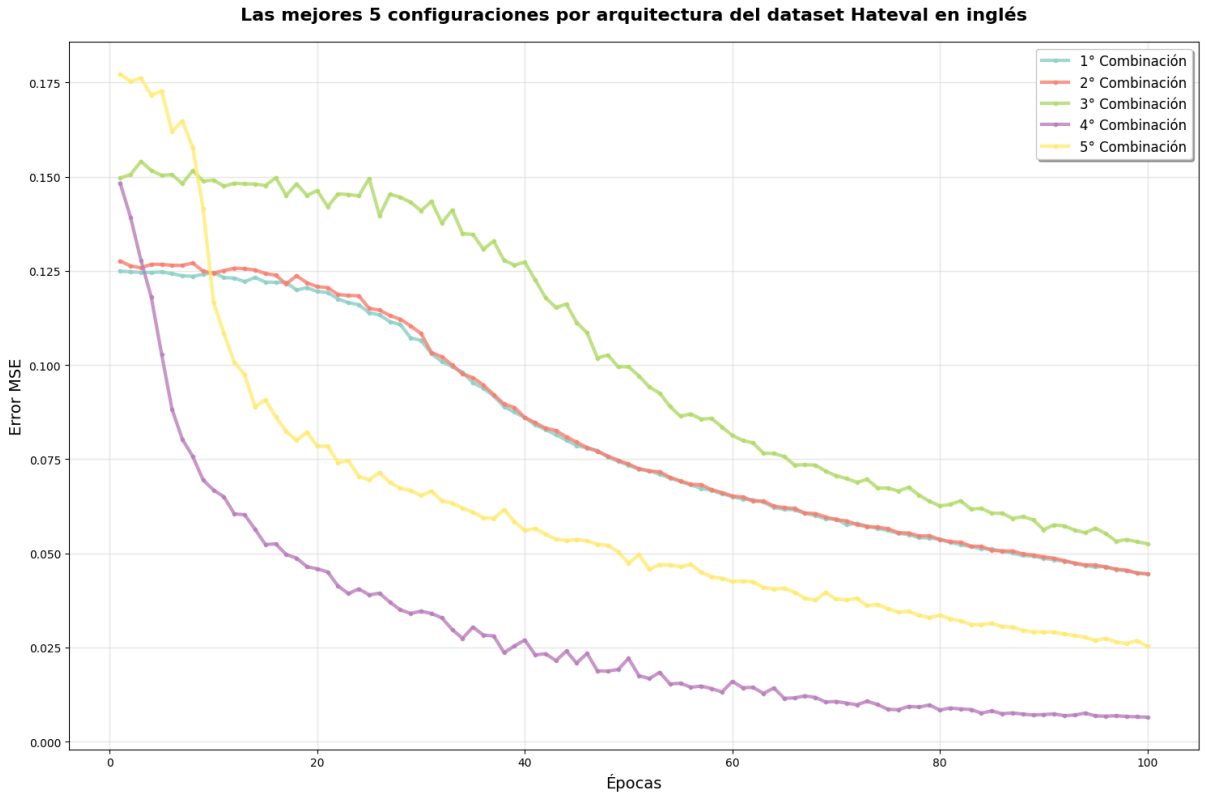

Figura 2: Evolución del Error (MSE) durante el entrenamiento para las 5 mejores configuraciones del modelo en el dataset en inglés. Se observa que las configuraciones con menor error final (azul y naranja) presentan una convergencia más estable y rápida, mientras que configuraciones con arquitecturas más grandes (verde) muestran mayor volatilidad. La combinación de 128 neuronas, inicialización Xavier y TF-IDF con unigramas+bigramas no solo logra el menor error, sino también la curva de aprendizaje más suave.

Tabla 10. Los 5 mejores experimentos (dataset en inglés)

Combi	Neur.	Inicializ.	LR	Batch	Pesado	N-grama	Preprocesamiento	Error	Tiempo (s)	Accuracy	F1-score
1	128	Xavier	0.01	64	TF-IDF	(1,2)	Sin StopWords	0.0445	205.3	0.732	**0.730**
2	256	Xavier	0.01	32	TF-IDF	(1,2)	Sin StopWords	0.0445	707.1	0.727	0.723
3	1024	Xavier	0.01	16	TF-IDF	(1,2)	Sin StopWords	0.0525	6275.5	0.727	0.721
4	64	Xavier	0.1	32	TF-IDF	(1,1)	Solo normalización	0.0064	**200.3**	0.717	0.713
5	512	Xavier	0.01	64	TF	(1,1)	Sin StopWords + Stemming	0.0253	1214.0	0.708	0.702

3. Interpretación y análisis

- **Mejor desempeño global:**
  La red con **128 neuronas** obtuvo el mejor resultado (Accuracy = 0.732, F1 = 0.730), logrando un equilibrio adecuado entre precisión, estabilidad y costo computacional. Esta arquitectura, al igual que en el dataset en español, muestra una tendencia óptima de generalización y eficiencia.
- **Efecto del número de neuronas:**
  El aumento del número de neuronas **no generó una mejora sostenida** en el rendimiento. A partir de 256 neuronas, las métricas tienden a estabilizarse o disminuir, mientras que el tiempo de entrenamiento se incrementa notablemente. Las redes más grandes (512 y 1024 neuronas) multiplicaron el tiempo de cómputo sin ofrecer mejoras sustanciales en las métricas de desempeño, repitiendo el mismo patrón observado en el dataset en español.
- **Inicialización Xavier:**
  En todos los casos se empleó la **inicialización Xavier**, que mostró un comportamiento estable y eficiente. En los experimentos globales (considerando las 100 ejecuciones), Xavier superó a la inicialización normal utilizada en el dataset en español, especialmente en configuraciones pequeñas y medianas. Esto sugiere que para el idioma inglés, donde la estructura del texto presenta menor **complejidad gramatical y léxica**, la propagación de gradientes proporcionada por Xavier resulta más adecuada.
- **Preprocesamiento del texto:** Los **mejores resultados** se lograron al normalizar el texto y eliminar *stopwords*, lo que permitió mantener una representación del significado más limpia y estable. Sin embargo, al aplicar *stemming* de forma adicional, el rendimiento disminuyó (F1 ≈ 0.70), evidenciando que una simplificación excesiva del vocabulario puede eliminar detalles útiles para la clasificación de textos en inglés.
- **Representación de términos:**
  El esquema **TF-IDF con unigramas+bigramas (1,2)** fue el más efectivo, mejorando la representación del contexto y permitiendo que el modelo distinguiera combinaciones de palabras con mayor poder semántico. Las configuraciones con unigramas (1,1) resultaron más limitadas, mientras que el uso de TF sin ponderación de frecuencia inversa fue menos consistente. Esto refuerza la idea de que en inglés, **TF-IDF** ofrece una ventaja clara al capturar términos más informativos y contextualmente relevantes.

2.9.6 Análisis general de resultados Dataset Inglés (100 experimentos)

Las configuraciones empleadas fueron **idénticas** a las utilizadas en el dataset en español, con el fin de mantener **condiciones comparables** y analizar el impacto del idioma sobre el desempeño del modelo. A continuación, se presenta el análisis detallado de las principales variables que influyeron en el rendimiento del modelo:

1. Inicialización de Pesos

En el total de 100 experimentos se emplearon dos estrategias de inicialización: **Normal (56%)** y **Xavier (44%)**. Aunque la inicialización **Normal** fue la más utilizada, los **mejores resultados individuales** provinieron de **Xavier**, que mostró un comportamiento más eficiente en redes pequeñas y medianas (64–256 neuronas). En contraste, la inicialización **Normal** mantuvo estabilidad en arquitecturas grandes, aunque con un rendimiento medio ligeramente inferior.
La inicialización **Normal** fue predominante en frecuencia y robustez, pero **Xavier** alcanzó los mejores F1 globales (0.73), por lo que se recomienda para arquitecturas de baja o media complejidad, donde facilita la convergencia estable y evita saturación de gradientes.

2. Representación de Términos

Se utilizaron dos esquemas principales: **TF** y **TF-IDF**. A diferencia del dataset en español, donde TF fue superior, en inglés la representación **TF-IDF** ofreció **mejor rendimiento global**, especialmente cuando se combinaron **unigramas+bigramas** y eliminación de stopwords. Esto sugiere que la ponderación por frecuencia inversa permite capturar mejor la relevancia léxica en textos con mayor variabilidad semántica como el inglés. La representación **TF-IDF** fue la más efectiva en inglés, mostrando ventajas en contextos de alta variabilidad léxica y expresiones idiomáticas.

3. Procesamiento del Texto

Se evaluaron las tres estrategias de preprocesamiento que también se evaluaron en el dataset de español. Los resultados muestran que el **preprocesamiento con eliminación de stopwords** fue el más consistente, logrando el mejor equilibrio entre reducción de ruido y retención de contexto. En cambio, el stemming introdujo una pérdida de precisión, probablemente al eliminar sufijos relevantes para la semántica del inglés.

Tabla 11. Impacto del preprocesamiento de texto en el F1-score para el dataset en inglés

Procesamiento	F1 promedio
only_normalizar_texto	0.70–0.72

normalizar_txt_sin_StopWords	**0.72–0.73**
normalizar_txt_sin_StopWords_mas_stemming	0.68–0.70

La eliminación de stopwords mejoró el rendimiento global. Sin embargo, aplicar stemming adicional redujo la capacidad del modelo para mantener relaciones léxicas precisas.

4. Rango de N-Gramas

Se emplearon tres configuraciones: **(1,1)**, **(1,2)** y **(2,2)**. Los resultados confirman que el rango **(1,2)** (unigramas + bigramas) ofreció el mejor rendimiento promedio, capturando relaciones locales sin comprometer la generalización. Las representaciones solo con bigramas (2,2) mostraron pérdida de información básica, mientras que (1,1) fue más estable pero menos expresiva.

Tabla 12: Comparación del rendimiento (F1-score) según el rango de n-gramas en inglés

N-grama	F1 promedio
(1,1)	~0.70
(1,2)	**~0.73**
(2,2)	~0.69

El rango **(1,2)** fue el más efectivo, reafirmando que incluir **unigramas+bigramas** enriquece la representación sin inducir sobreajuste.

5. Learning Rate y Batch Size

Se evaluaron distintas combinaciones de tasa de aprendizaje y tamaño de lote. El mejor rendimiento se obtuvo con **learning rate = 0.01** y **batch size = 16 o 64**, coincidiendo con la estabilidad observada en español. Tasas altas (0.1 o 0.5) produjeron sobreajuste o divergencia, mientras que batches muy pequeños (16) aumentaron la varianza de los gradientes. El uso de **LR=0.01** con batches medianos permitió una convergencia equilibrada y resultados más consistentes en todos los grupos de neuronas.

6. Tiempo de Entrenamiento y Eficiencia Computacional

El tiempo de entrenamiento creció exponencialmente con el número de neuronas y el uso de representaciones más complejas. Los modelos con **128 o 256 neuronas** lograron la **mejor relación entre precisión y tiempo**, mientras que las redes grandes (512 y 1024 neuronas) incrementaron drásticamente el costo computacional sin ganancias proporcionales en F1.

Tabla 13. Relación entre arquitectura de red, tiempo de entrenamiento y rendimiento (F1-score)

Arquitectura	Tiempo (s)	F1 promedio
64–128	200–250	**Alta (~0.73)**
256	700	Media (~0.72)
512–1024	>1200	Baja (~0.70)

Las arquitecturas medianas (128–256 neuronas) son las más eficientes, maximizando el F1 con un costo de entrenamiento razonable.

7. Conclusión

El conjunto de 100 experimentos evidencia que el **rendimiento del modelo en inglés** se mantiene **estable pero ligeramente inferior** al observado en español, con una mayor sensibilidad a los hiperparámetros de inicialización y vectorización.
El modelo con **128 neuronas, Xavier y TF-IDF (1,2)** alcanzó el **mejor F1-score = 0.73 y Accuracy = 0.732**, mostrando un equilibrio sólido entre precisión, estabilidad y eficiencia. Estos resultados confirman que, aunque el idioma inglés presenta mayor diversidad semántica, el pipeline experimental mantiene su robustez y capacidad de generalización bajo las mismas condiciones de diseño.

3. Conclusión final

El análisis de 200 experimentos sobre detección de discurso de odio permitió identificar patrones claros sobre cómo el idioma influye en el rendimiento de los modelos. Los resultados muestran que, aunque existen principios generales en el diseño de redes neuronales, la efectividad final depende en gran medida de las características lingüísticas propias de cada lengua.

En cuanto al rendimiento, el español mostró una ventaja importante, alcanzando un **F1-score de 0.807 frente al 0.730 del inglés**. Esta diferencia del 7.7% sugiere que la estructura del español, más regular y predecible, facilita al modelo reconocer de forma equilibrada los mensajes de odio y los que no lo son. Aun así, ambos idiomas coincidieron en una configuración óptima con **128 neuronas**, lo que confirma que los modelos de complejidad media logran el mejor equilibrio entre capacidad de generalización y eficiencia computacional.

Las diferencias en el preprocesamiento también fueron muy reveladoras. El español funcionó mejor con estrategias simples usando **TF** y una normalización básica, mientras que el inglés necesitó enfoques más avanzados como **TF-IDF** y eliminación de **stopwords** para manejar la mayor diversidad de su vocabulario. Esto demuestra que cada idioma requiere un procesamiento ajustado a sus propias características.

Por otro lado, **limitar las features a 10,000** fue una decisión muy acertada. Los experimentos demostraron que esta reducción mantenía el mismo nivel de F1-score (0.807), pero reducía el tiempo de entrenamiento en un **86%**. La mejora mínima de 0.004 al usar todas las features no justificó el aumento de 30 veces en el costo computacional, lo que confirma que en modelos de alta dimensionalidad, invertir más recursos no siempre significa obtener mejores resultados.

Esta experimentacion no sólo permitió construir un sistema eficaz para detectar discurso de odio, sino que también aportó aprendizajes valiosos sobre cómo diseñar soluciones de **NLP** que funcionen bien en distintos idiomas. El principal hallazgo es que el éxito en el aprendizaje profundo aplicado al lenguaje natural **no depende solo del modelo**, sino también de entender las particularidades lingüísticas y encontrar un equilibrio entre **precisión, eficiencia y viabilidad computacional**.

4. Referencias

[1] "Dimensionality Reduction in Machine Learning," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/dimensionality-reduction/

[2] "Dimensionality Reduction Techniques in Machine Learning," Encord Blog. [Online]. Available: https://encord.com/blog/dimentionality-reduction-techniques-machine-learning/

[3] N. Arrioja, "13 Técnicas de Reducción de Dimensionalidad," Medium, Aug. 30, 2023. [Online]. Available: https://medium.com/@nicolasarrioja/13-t%C3%A9cnicas-de-reducci%C3%B3n-de-dimensionalidad-b33b2340a060

[4] T. S. Ramzan et al., "A Review of Feature Selection Methods for Machine Learning-Based Disease Risk Prediction," Front. Inform., vol. 6, 2022, doi: 10.3389/fbinf.2022.927336. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC9580915/

5. Anexo

Código utilizado para poder llevar a cabo esta practica:

Código para generar combinaciones aleatorias

```
import itertools
import random
import pandas as pd

def generar_combinaciones_completas():
    """
    Genera todas las combinaciones posibles de parámetros
    """
    parametros = {
        'neuronas': [666],
        'inicializacion': ['Normal', 'Xavier'],
        'pesado': ['TF', 'TF-IDF'],
        'representacion': [(1,1), (2,2), (1,2)],
        'preprocesamiento': [
            'only_normalizar_texto',
            'normalizar_txt_sin_StopWords',
            'normalizar_txt_sin_StopWords_mas_stemming'
        ],
        'learning_rate': [0.01, 0.1, 0.5],
        'batch_size': [16, 32, 64],
        'epocas': [500]
    }

    # Generar todas las combinaciones
    claves = list(parametros.keys())
    valores = list(parametros.values())

    combinaciones = []
    for combinacion in itertools.product(*valores):
        config = dict(zip(claves, combinacion))
        combinaciones.append(config)

    return combinaciones

def seleccionar_combinaciones_aleatorias(combinaciones, n=20, semilla=42):
    """
    Selecciona n combinaciones aleatorias de la lista completa
    """
    random.seed(semilla)

    if n > len(combinaciones):
        n = len(combinaciones)
```

```
        print(f"Advertencia: n={n} es mayor que el total de combinaciones.
Seleccionando todas.")

    combinaciones_aleatorias = random.sample(combinaciones, n)
    return combinaciones_aleatorias

def guardar_combinaciones_csv(combinaciones,
archivo="combinaciones_experimentos.csv"):
    """
    Guarda las combinaciones en un archivo CSV
    """
    # Convertir a DataFrame
    df = pd.DataFrame(combinaciones)

    # Guardar en CSV
    df.to_csv(archivo, index=False, encoding='utf-8')
    print(f"Combinaciones guardadas en: {archivo}")
    print(f"Total de combinaciones guardadas: {len(combinaciones)}")

    return df


#
================================================================================
=
# FUNCIÓN PRINCIPAL - EJECUCIÓN COMPLETA
#
================================================================================
=

def generar_y_guardar_experimentos(n_combinaciones=20, semilla=42,
archivo_csv="combinaciones_experimentos.csv"):
    """
    Función principal que genera, selecciona y guarda combinaciones para
experimentos
    """
    print("GENERADOR DE COMBINACIONES PARA EXPERIMENTOS")
    print("=" * 60)

    # 1. Generar todas las combinaciones posibles
    print("Generando todas las combinaciones posibles...")
    todas_combinaciones = generar_combinaciones_completas()

    print(f"Total de combinaciones posibles: {len(todas_combinaciones)}")

    # 2. Seleccionar combinaciones aleatorias
    print(f"Seleccionando {n_combinaciones} combinaciones aleatorias...")
    combinaciones_seleccionadas = seleccionar_combinaciones_aleatorias(
```

```
        todas_combinaciones, n_combinaciones, semilla
    )

    print(f"{len(combinaciones_seleccionadas)} combinaciones seleccionadas")

    # 3. Guardar en archivo CSV
    guardar_combinaciones_csv(combinaciones_seleccionadas, archivo_csv)

    return combinaciones_seleccionadas

#
==========================================================================
=
# EJEMPLOS DE USO
#
==========================================================================
=

if __name__ == "__main__":
    # Opción 1: Generar y guardar nuevas combinaciones
    print("GENERAR NUEVAS COMBINACIONES")
    combinaciones = generar_y_guardar_experimentos(
        n_combinaciones=20,
        semilla=72,
        archivo_csv="combinaciones_experimentos.csv"
    )

Codigo para leer el dataset de formato .json
#FUNCION PARA LEER EL ARCIHVO DEL DATA SET
import json

def load_hateval_json(file_path):
    texts = []
    labels = []

    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            try:
                data = json.loads(line.strip())
                texts.append(data["text"])
                labels.append(int(data["klass"]))
            except Exception as e:
                print(f"Error leyendo línea: {e}")

    return texts, labels
```

Aquí solo cambiamos el archivo para inglés y para español para poder trabajar con los dataset.

```
#Paso 2 — Cargar los datos de entrenamiento y prueba
X_train_es, y_train_es = load_hateval_json("hateval_es_train_spanish.json")
X_test_es, y_test_es = load_hateval_json("hateval_es_test_spanish.json")
```

```
#Comprobar las cargas
print("ES train:", len(X_train_es))
print("ES test:", len(X_test_es))
```

Preprocesamiento de texto de español

```
# Agregar estos imports al inicio
import nltk
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer
import unicodedata
import re  # También necesitamos re para las expresiones regulares

# Descargar stopwords si es necesario
# nltk.download('stopwords')

# Inicializar recursos de NLP
stemmer = SnowballStemmer('spanish')
stop_words_es = set(stopwords.words('spanish'))

# Tus constantes para normalización
PUNCTUACTION = ";:,.\\-\"'/"
SYMBOLS = "()[]¿?¡!{}~<>|"
NUMBERS = "0123456789"
SKIP_SYMBOLS = set(PUNCTUACTION + SYMBOLS)

def only_normalizar_texto(input_str, punct=False, accents=False, num=False,
max_dup=2, lowercase=True):
    """
    Tu función de normalización (la que ya tenías)
    """
    nfkd_f = unicodedata.normalize('NFKD', input_str)
    n_str = []
    c_prev = ''
    cc_prev = 0
    for c in nfkd_f:
        if not num:
            if c in NUMBERS:
                continue
```

```
        if not punct:
            if c in SKIP_SYMBOLS:
                continue
        if not accents and unicodedata.combining(c):
            continue
        if c_prev == c:
            cc_prev += 1
            if cc_prev >= max_dup:
                continue
        else:
            cc_prev = 0
        n_str.append(c)
        c_prev = c
    texto = unicodedata.normalize('NFKD', "".join(n_str))
    texto = re.sub(r'(\s)+', r' ', texto.strip(), flags=re.IGNORECASE)

    if lowercase:
        texto = texto.lower()

    return texto

def normalizar_txt_sin_StopWords(texto):
    """
    Normaliza + elimina stopwords
    """
    texto_normalizado = only_normalizar_texto(
        texto, punct=False, accents=False, num=False, max_dup=2,
lowercase=True
    )

    if not texto_normalizado:
        return ""

    palabras = texto_normalizado.split()
    palabras_filtradas = [palabra for palabra in palabras if palabra not in
stop_words_es]

    return ' '.join(palabras_filtradas)

def normalizar_txt_sin_StopWords_mas_stemming(texto):
    """
    Normaliza + elimina stopwords + aplica stemming
    """
    texto_sin_stopwords = normalizar_txt_sin_StopWords(texto)

    if not texto_sin_stopwords:
        return ""
```

```
    palabras = texto_sin_stopwords.split()
    palabras_stemmed = [stemmer.stem(palabra) for palabra in palabras]

    return ' '.join(palabras_stemmed)

def mi_preprocesamiento(tipo_procesamiento):
    """
    Función dispatcher para seleccionar preprocesamiento
    """
    procesadores = {
        'only_normalizar_texto': lambda x: only_normalizar_texto(
            x, punct=False, accents=False, num=False, max_dup=2,
lowercase=True
        ),
        'normalizar_txt_sin_StopWords': normalizar_txt_sin_StopWords,
        'normalizar_txt_sin_StopWords_mas_stemming':
normalizar_txt_sin_StopWords_mas_stemming
    }

    return procesadores[tipo_procesamiento]
```

Preprocesamiento de texto para inglés

```
# Agregar estos imports al inicio
import nltk
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer
import unicodedata
import re

# Descargar stopwords si es necesario
nltk.download('stopwords')

# Inicializar recursos de NLP PARA INGLÉS
stemmer = SnowballStemmer('english')
stop_words_in = set(stopwords.words('english'))  #  Cambié a stop_words_in

# Tus constantes para normalización (se mantienen igual)
PUNCTUACTION = ";:,.\\-\"'/"
SYMBOLS = "()[]¿?¡!{}~<>|"
NUMBERS = "0123456789"
SKIP_SYMBOLS = set(PUNCTUACTION + SYMBOLS)

def only_normalizar_texto(input_str, punct=False, accents=False, num=False,
max_dup=2, lowercase=True):
    """
    Tu función de normalización (se mantiene IGUAL)
```

```
    """
    nfkd_f = unicodedata.normalize('NFKD', input_str)
    n_str = []
    c_prev = ''
    cc_prev = 0
    for c in nfkd_f:
        if not num:
            if c in NUMBERS:
                continue
        if not punct:
            if c in SKIP_SYMBOLS:
                continue
        if not accents and unicodedata.combining(c):
            continue
        if c_prev == c:
            cc_prev += 1
            if cc_prev >= max_dup:
                continue
        else:
            cc_prev = 0
        n_str.append(c)
        c_prev = c
    texto = unicodedata.normalize('NFKD', "".join(n_str))
    texto = re.sub(r'(\s)+', r' ', texto.strip(), flags=re.IGNORECASE)

    if lowercase:
        texto = texto.lower()

    return texto

def normalizar_txt_sin_StopWords(texto):
    """
    Normaliza + elimina stopwords EN INGLÉS
    """
    texto_normalizado = only_normalizar_texto(
        texto, punct=False, accents=False, num=False, max_dup=2,
lowercase=True
    )

    if not texto_normalizado:
        return ""

    palabras = texto_normalizado.split()
    palabras_filtradas = [palabra for palabra in palabras if palabra not in
stop_words_in]  #Cambié a stop_words_in

    return ' '.join(palabras_filtradas)
```

```
def normalizar_txt_sin_StopWords_mas_stemming(texto):
    """
    Normaliza + elimina stopwords + aplica stemming EN INGLÉS
    """
    texto_sin_stopwords = normalizar_txt_sin_StopWords(texto)

    if not texto_sin_stopwords:
        return ""

    palabras = texto_sin_stopwords.split()
    palabras_stemmed = [stemmer.stem(palabra) for palabra in palabras]

    return ' '.join(palabras_stemmed)

def mi_preprocesamiento(tipo_procesamiento):
    """
    Función dispatcher para seleccionar preprocesamiento (mismo nombre)
    """
    procesadores = {
        'only_normalizar_texto': lambda x: only_normalizar_texto(
            x, punct=False, accents=False, num=False, max_dup=2,
lowercase=True
        ),
        'normalizar_txt_sin_StopWords': normalizar_txt_sin_StopWords,
        'normalizar_txt_sin_StopWords_mas_stemming':
normalizar_txt_sin_StopWords_mas_stemming
    }
    return procesadores[tipo_procesamiento]
```

Aqui esta el codigo de la MLP desde 0

```
#CLASE MLP_TODO.PY Y FUNCIONES AUXILIARES
import numpy as np
import matplotlib.pyplot as plt
import random
import pandas as pd
from time import time

# ====================================================
# Funciones para activación y  su derivada
# ====================================================

# Función de activación sigmoide
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivada de la sigmoide
```

```
def sigmoid_derivative(x):
    # return sigmoid(x) * (1 - sigmoid(x))
    return x * (1 - x)

# ====================================================
# Funciones para manejo de la semilla
# ====================================================

# Establece la semilla para la generación de números aleatorios
def seed(random_state=33):
    np.random.seed(random_state)
    random.seed(random_state)

# ====================================================
# Funciones para inicialización y normalización
# ====================================================

# Inicialización Xavier
def xavier_initialization(input_size, output_size):
    return np.random.randn(input_size, output_size) * np.sqrt(1 /
input_size)
# Inicialización normal
def normal_initialization(input_size, output_size):
    return np.random.randn(input_size, output_size)
# Normalización Z-score
def zscore_normalization(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    X_norm = (X - mean) / std
    return X_norm

#Función para crear minibatches
def create_minibatches(X, y, batch_size):
    """
    Genera los lotes de datos (batchs) de acuerdo al parámetro batch_size de
forma aleatoria para el procesamiento.
    """
    n_samples = X.shape[0]
    indices = np.random.permutation(n_samples)  # Mezcla los índices
aleatoriamente
    X_shuffled, y_shuffled = X[indices], y[indices]  # Reordena X e y según
los índices aleatorios

    # Divide los datos en minibatches
    for X_batch, y_batch in zip(np.array_split(X_shuffled, np.ceil(n_samples
/ batch_size)),
                                np.array_split(y_shuffled, np.ceil(n_samples
/ batch_size))):
```

```
        yield X_batch, y_batch


class MLP_TODO:
    def __init__(self, num_entradas, num_neuronas_ocultas, num_salidas,
epochs, batch_size=128, learning_rate=0.2,
normalizacion="none",inicializacion="xavier",random_state=42):

        # ====================================================
        # Inicialización general del modelo
        # ====================================================


        seed(33)
        self.random_state = random_state

        # Definir la tasa de aprendizaje
        self.learning_rate = learning_rate
        # Definir el número de épocas
        self.epochs = epochs
        # Definir el tamaño del batch de procesamiento
        self.batch_size = batch_size
        # Definir el tipo de normalización
        self.normalizacion = normalizacion
        # Definir el tipo de inicialización
        self.inicializacion = inicializacion
        # definir las
        self.num_neuronas_ocultas = num_neuronas_ocultas

        # Inicialización de pesos y bias
        self.W1 = self.inicializar_pesos(num_entradas,
self.num_neuronas_ocultas) # Pesos entre capa de entrada y capa oculta
        self.b1 = np.zeros((1, self.num_neuronas_ocultas))   # Bias de la
capa oculta
        self.W2 =
self.inicializar_pesos(self.num_neuronas_ocultas,num_salidas)  # Pesos entre
capa oculta y capa de salida
        self.b2 = np.zeros((1, num_salidas)) # Bias de la capa de salida

        # Historial de errores
        self.errores_history = []
        # Historial de accuracy
        self.accuracy_history = []

    # ====================================================
    # Funciones para forward, backward, update, predict y train
    # ====================================================
```

```
    def forward(self, X):
        #implementar el forward pass
        #-----------------------------------------------
        # 1. Propagación hacia adelante (Forward pass)
        #-----------------------------------------------
        # Calcular la suma ponderada Z (z_c1) para la capa oculta
        self.X = X
        self.z_c1 = X@self.W1 + self.b1
        #Calcular la activación de la capa oculta usando la función sigmoide
        self.a_c1 = sigmoid(self.z_c1)  # Activación capa oculta
        #Calcular la suma ponderada Z (z_c2)  para la capa de salida
        self.z_c2  = self.a_c1 @ self.W2 + self.b2
        #Calcular la activación de la capa de salida usando la función
sigmoide
        y_pred = sigmoid(self.z_c2)  # Activación capa salida
        return y_pred


    def loss_function_MSE(self, y_pred, y):
        #-----------------------------------------------
        # 2. Cálculo del error con MSE
        #-----------------------------------------------
        #Calcular el error cuadrático medio (MSE)
        self.y_pred = y_pred
        self.y = y
        error = 0.5 * np.mean((y_pred - y) ** 2)
        return error


    def backward(self):
        #implementar el backward pass
        # calcular los gradientes para la arquitectura de la figura anterior
        #-----------------------------------------------
        # 3. Propagación hacia atrás (Backward pass)
        #-----------------------------------------------

        #-----------------------------------------------
        # Gradiente de la salida
        #-----------------------------------------------
        #Calcular la derivada del error con respecto a la salida y
        dE_dy_pred = (self.y_pred - self.y)  # Derivada del error respecto a
la predicción con  N ejemplos
        #Calcular la derivada de la activación de la salida con respecto a
z_c2
        d_y_pred_d_zc2 = sigmoid_derivative(self.y_pred)
        #Calcular delta de la capa de salida
        delta_c2 = dE_dy_pred * d_y_pred_d_zc2  # (N, 1)
```

```
        #---------------------------------------------
        # Gradiente en la capa oculta
        #---------------------------------------------
        # calcular la derivada de las suma ponderada respecto a las
activaciones de la capa 1
        d_zc2_d_a_c1 = self.W2
        #Propagar el error hacia la capa oculta, calcular deltas de la capa
1
        delta_c1 = delta_c2 @ d_zc2_d_a_c1.T * sigmoid_derivative(self.a_c1)

        #calcula el gradiente de la función de error respecto a los pesos de
la capa 2
        self.dE_dW2 = self.a_c1.T @ delta_c2
        self.dE_db2 = np.sum(delta_c2, axis=0, keepdims=True)
        self.dE_dW1 = self.X.T @ delta_c1
        self.dE_db1 = np.sum(delta_c1, axis=0, keepdims=True)


    def update(self):  # Ejecución de la actualización de paramámetros
        #implementar la actualización de los pesos y el bias
        #---------------------------------------------
        # Actualización de pesos de la capa de salida
        #---------------------------------------------
        #Actualizar los pesos y bias de la capa de salida
        self.W2 = self.W2 - self.dE_dW2 * self.learning_rate
        self.b2 = self.b2 - self.dE_db2 * self.learning_rate
        #---------------------------------------------
        # Actuailzación de pesos de la capa oculta
        #---------------------------------------------
        #calcula el gradiente de la función de error respecto a los pesos de
la capa 1
        self.W1 = self.W1 - self.dE_dW1 * self.learning_rate
        self.b1 = self.b1 - self.dE_db1 * self.learning_rate

    def predict(self, X):  # Predecir la categoría para datos nuevos
        # TODO: implementar la predicción
        y_pred = self.forward(X)
        # Obtener la clase para el clasificador binario
        y_pred = np.where(y_pred >= 0.5, 1, 0)
        return y_pred

    def train(self, X, Y, paciencia=10):
        """
        Entrenamiento con parada temprana
        """
        # Normalizar los datos
        X = self.normalize(X)
```

```
        # Variables para parada temprana
        mejor_error = float('inf')
        epocas_sin_mejora = 0

        for epoch in range(self.epochs):
            num_batch = 0
            epoch_error = 0

            # Procesar cada batch
            for X_batch, y_batch in create_minibatches(X, Y,
self.batch_size):
                y_pred = self.forward(X_batch)
                error = self.loss_function_MSE(y_pred, y_batch)
                epoch_error += error
                self.backward()
                self.update()
                num_batch += 1

            # Error promedio de la época
            error_promedio = epoch_error / num_batch
            self.errores_history.append(error_promedio)

            # Calcular Accuracy
            acc_epoch = self.accuracy(X, Y)
            self.accuracy_history.append(acc_epoch)

            # Verificar si hay mejora
            if error_promedio < mejor_error:
                mejor_error = error_promedio
                epocas_sin_mejora = 0
            else:
                epocas_sin_mejora += 1

            # Parar si no hay mejora después de 'paciencia' épocas
            if epocas_sin_mejora >= paciencia:
                print(f"Parada temprana en época {epoch}. Error:
{error_promedio:.6f}")
                break

            # Imprimir progreso cada 50 épocas
            if epoch % 50 == 0:
                print(f"Época {epoch:3d} | Error: {error_promedio:.6f} |
Accuracy: {acc_epoch:.4f}")

        print(f"Entrenamiento finalizado. Mejor error: {mejor_error:.6f}")

    def evaluar(self, X, y):
        """
```

```
        Evalúa el desempeño del modelo en un conjunto de prueba.
        Muestra todas las predicciones junto con las salidas esperadas en
columnas paralelas.
        Calcula la precisión total.
        """
        # Normalizar los datos de entrada con la misma técnica usada en
entrenamiento
        X = self.normalize(X)

        # Obtener las predicciones
        y_pred = self.predict(X)

        # Combinar esperadas y predicciones en columnas paralelas
        resultados = np.column_stack((y, y_pred))

        # Mostrar resultados
        print("\nEvaluación del modelo (esperada | predicha):")
        for idx, (esperada, predicha) in enumerate(resultados):
            print(f"{idx+1:02d}: {esperada} | {predicha}")

        # Calcular precisión global
        accuracy = np.mean(y_pred == y)
        print(f"\nPrecisión del modelo: {accuracy * 100:.2f}%")

        return accuracy



    # ======================================================
    # Funciones para inicialización, normalización y accuracy
    # ======================================================
    # Normalización de los datos
    def normalize(self, X):
        if self.normalizacion == "z-score":
            return zscore_normalization(X)  # 🔹 Llamada a la función
existente
        else:  # sin normalizar
            return X

    # Inicialización de los pesos
    def inicializar_pesos(self, tamaño_entrada, tamaño_salida):
        if self.inicializacion == "xavier":
            return xavier_initialization(tamaño_entrada, tamaño_salida)
        elif self.inicializacion == "normal":
            return normal_initialization(tamaño_entrada, tamaño_salida)
        else:
            raise ValueError("Tipo de inicialización no soportado")
    # Cálculo de accuracy
```

```
    def accuracy(self, X, y):
        y_pred = self.predict(X)
        acc = np.mean(y_pred == y)  # compara predicciones con valores
reales
        return acc

    # ====================================================
    # Funciones para graficar Error, Acurery y ambas
    # ====================================================

    # Gráfica del error despues del entrenamiento
    def plot_error(self):
        plt.figure(figsize=(8, 5))
        plt.plot(self.errores_history, label="Error MSE", linewidth=2)
        plt.xlabel("Épocas")
        plt.ylabel("Error cuadrático medio (MSE)")
        #Título dinámico con configuración
        plt.title(f"Entrenamiento MLP - Capacas ocultas:
{self.num_neuronas_ocultas}, Inicializacion: {self.inicializacion},
Normalización: {self.normalizacion}, "
                  f"LR: {self.learning_rate}, Batch_size:
{self.batch_size},Funcion de activacion: Sigmoid, Épocas: {self.epochs}")
        plt.legend()
        plt.grid(True)
        plt.show()
```

FUNCIONES PARA LEER COMBINACIONES Y GUARDAR RESULTADOS EN ARCHIVOS CSV

```
import pandas as pd
import ast
from sklearn.metrics import precision_score, recall_score, f1_score,
accuracy_score
from time import time

def leer_combinaciones_desde_csv(archivo_csv):
    """
    Lee las combinaciones desde un archivo CSV
    """
    df = pd.read_csv(archivo_csv)
    combinaciones = []

    for _, row in df.iterrows():
        config = {
            'neuronas': int(row['neuronas']),
            'inicializacion': row['inicializacion'],
            'pesado': row['pesado'],
            'representacion': ast.literal_eval(row['representacion']),
```

```
            'preprocesamiento': row['preprocesamiento'],
            'learning_rate': float(row['learning_rate']),
            'batch_size': int(row['batch_size']),
            'epocas': int(row['epocas'])
        }
        combinaciones.append(config)

    print(f"Leídas {len(combinaciones)} combinaciones desde {archivo_csv}")
    return combinaciones

def guardar_resultados_csv(resultados,
archivo_salida="resultados_experimentos.csv"):
    """Guarda los resultados en un archivo CSV"""
    datos_csv = []

    for resultado in resultados:
        config = resultado['config']
        fila = {
            'Neuronas': config['neuronas'],
            'Inicialización': config['inicializacion'],
            'Normalización': 'none',
            'Learning rate': config['learning_rate'],
            'Batch size': config['batch_size'],
            'Pesado_termino': config['pesado'],
            'Representacion_terminos': str(config['representacion']),
            'ProcesamientosTXT': config['preprocesamiento'],
            'Epocas_ejecutadas': resultado.get('epocas_ejecutadas',
config['epocas']),
            'Error final': resultado['final_loss'],
            'Tiempo (s)': resultado['training_time'],
            'Presicion(macro)': resultado['precision'],
            'Recall(macro)': resultado['recall'],
            'Accuracy(macro)': resultado['accuracy'],
            'F1-score(macro)': resultado['f1_score']
        }
        datos_csv.append(fila)

    # Crear DataFrame y guardar
    df_resultados = pd.DataFrame(datos_csv)
    df_resultados.to_csv(archivo_salida, index=False, encoding='utf-8')

    print(f"Resultados guardados en: {archivo_salida}")
    print(f"Total de experimentos guardados: {len(datos_csv)}")

    return df_resultados
```

FUNCIÓN PARA EJECUTAR UN EXPERIMENTO

```
#FUNCION PARA EJECUTAR EXPERIMETO COMPLETO
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
def ejecutar_experimento_completo(config, X_train, y_train, X_test, y_test):
    """
    Ejecuta un experimento completo con una configuración específica
    y retorna todas las métricas solicitadas
    """
    print(f"\n Ejecutando experimento:")
    print(f"   Preprocesamiento: {config['preprocesamiento']}")
    print(f"   Pesado: {config['pesado']}")
    print(f"   Representación: {config['representacion']}")
    print(f"   Neuronas: {config['neuronas']}, LR:
{config['learning_rate']}, Batch: {config['batch_size']}")

    # 1. Crear vectorizador según la configuración
    if config['pesado'] == 'TF':
        Vectorizador = CountVectorizer
    else:  # TF-IDF
        Vectorizador = TfidfVectorizer

    vectorizador = Vectorizador(
        analyzer="word",
        preprocessor=mi_preprocesamiento(config['preprocesamiento']),
        ngram_range=config['representacion'],

    )

    try:
        # 2. Vectorizar datos
        X_train_vec = vectorizador.fit_transform(X_train).toarray()
        X_test_vec = vectorizador.transform(X_test).toarray()

        print(f"   Dimensiones: {X_train_vec.shape} -> {X_test_vec.shape}")

        # 3. Preparar parámetros para MLP_TODO
        n_input = X_train_vec.shape[1]
        n_hidden = config['neuronas']
        n_output = 1

        # Mapear inicialización
        init_map = {'Normal': 'normal', 'Xavier': 'xavier'}

        # 4. Crear y entrenar modelo
        model = MLP_TODO(
            num_entradas=n_input,
            num_neuronas_ocultas=n_hidden,
```

```
            num_salidas=n_output,
            epochs=config['epocas'],
            batch_size=config['batch_size'],
            learning_rate=config['learning_rate'],
            normalizacion="none",
            inicializacion=init_map[config['inicializacion']],
            random_state=42
        )

        # 5. Entrenar modelo
        start_time = time()
        model.train(X_train_vec, np.array(y_train).reshape(-1, 1),
paciencia=10)
        training_time = time() - start_time

        # === AGREGAR ESTAS 2 LÍNEAS ===
        # Obtener épocas realmente ejecutadas
        epocas_ejecutadas = len(model.errores_history)


        # 6. Evaluar
        y_pred = model.predict(X_test_vec)

        # 7. Calcular todas las métricas solicitadas
        precision = precision_score(y_test, y_pred, average='macro',
zero_division=0)
        recall = recall_score(y_test, y_pred, average='macro',
zero_division=0)
        f1 = f1_score(y_test, y_pred, average='macro', zero_division=0)
        accuracy = accuracy_score(y_test, y_pred)
        final_loss = model.errores_history[-1] if model.errores_history else
0

        # 8. Preparar resultados
        results = {
            'precision': precision,
            'recall': recall,
            'f1_score': f1,
            'accuracy': accuracy,
            'final_loss': final_loss,
            'training_time': training_time,
            'epocas_ejecutadas': epocas_ejecutadas,
            'config': config
        }

        print(f"   Resultados - F1: {f1:.4f}, Accuracy: {accuracy:.4f},
Time: {training_time:.2f}s")
```

```
        return results

    except Exception as e:
        print(f"   Error en experimento: {e}")
        return None
```

Con esta parte fui haciendo ejecuciones de 20 experimentos por corrida pasando el archivo de combinaciones y guardaba los resultados en un archivo .csv para luego empezar a hacer el análisis.

```
# CODIGO PARA AUTOMATIZAR LOS EXPERIMENTOS
combinaciones =
leer_combinaciones_desde_csv("combinaciones_experimentos_64.csv")

if combinaciones:
    # Cambiar todas las épocas a 100
    for config in combinaciones:
        config['epocas'] = 100

    print(f"Iniciando {len(combinaciones)} experimentos automáticos")

    # Lista para guardar todos los resultados
    todos_resultados = []

    # Recorrer cada configuración
    for i, config in enumerate(combinaciones, 1):
        print(f"\n EXPERIMENTO {i}/{len(combinaciones)}")
        print(f"Configuración: {config}")

        # Ejecutar experimento
        resultado = ejecutar_experimento_completo(
            config,
            X_train_es,
            y_train_es,
            X_test_es,
            y_test_es
        )

        if resultado:
            # Agregar a la lista de resultados
            todos_resultados.append(resultado)

            # Guardar resultados después de cada experimento
            guardar_resultados_csv(todos_resultados,
"resultados_automaticos_1024.csv")
            print(f"Experimento {i} guardado - Épocas:
```

```
{resultado['epocas_ejecutadas']}")
        else:
            print(f"Experimento {i} falló")

    print(f"\n¡TODOS LOS EXPERIMENTOS COMPLETADOS!")
    print(f"Total: {len(todos_resultados)}/{len(combinaciones)} exitosos")
```