

# MILESTONE 1

## Rule-based Text Classification

### **Objetivo**

Desarrollar un sistema que clasifique comentarios en **ofensivos (1)** o **no ofensivos (0)** **utilizando reglas lingüísticas** extraídas del texto, sin usar aprendizaje automático todavía.

### **PASOS A SEGUIR**

#### 1. Descargar y cargar el dataset HateBR

- Fuente oficial: [HateBR en Hugging Face](#)
- Contiene dos columnas principales:
  - text: comentario en portugués
  - label: 0 (no ofensivo) o 1 (ofensivo)

**Objetivo:** cargar los datos y explorar cuántos ejemplos hay de cada clase.


 *Librerías recomendadas:* pandas, datasets (de HuggingFace).

---


#### 2. Análisis exploratorio (EDA – Exploratory Data Analysis)

Analiza las características principales del texto:

- Longitud media del comentario.
- Frecuencia de palabras más comunes (tokens).
- Palabras características de cada clase (ofensiva / no ofensiva).
- Uso de mayúsculas, emojis, puntuación, insultos o palabras emocionales.

 *Librerías recomendadas:*

nlTK, spacy, collections.Counter, matplotlib, wordcloud.

 **Objetivo:** identificar *patrones lingüísticos* que te ayuden a definir reglas.

Ejemplo: si un comentario contiene insultos → ofensivo.


---

#### 3. Procesamiento del lenguaje


Aplica técnicas de preprocesamiento para obtener conocimiento lingüístico del texto:

- **Tokenización**
- **Lematización / stemming**
- **Etiquetado POS (Part-of-Speech tagging)**

- **Extracción de entidades (NER)**
- **Análisis de sentimiento (opcional)**

 *Librerías sugeridas:*

spacy (modelo portugués: pt\_core\_news\_sm), nltk, textblob-pt o vaderSentiment.

 **Objetivo:** obtener representaciones que faciliten crear reglas lingüísticas.


---

#### 4. Definir reglas lingüísticas

Crea reglas **explícitas** basadas en el conocimiento obtenido.

Ejemplos de reglas:

- Si el texto contiene palabras de una **lista de insultos** → **ofensivo (1)**
- Si el texto tiene un **sentimiento positivo alto** → **no ofensivo (0)**
- Si el texto incluye **mucha puntuación fuerte (!, ??, mayúsculas)** → más probable ofensivo.
- Si el texto contiene **nombres propios + insultos** → **ofensivo**.

 Define cada regla con su **justificación** (por qué la creaste).

---

#### 5. Aplicar las reglas al dataset

Implementa una función en Python que recorra los textos y aplique las reglas, asignando una etiqueta 0 o 1 según corresponda.


Ejemplo:

```
def classify_comment(text):  
    if any(insult in text.lower() for insult in offensive_words):  
        return 1  
    elif text.isupper():  
        return 1  
    else:  
        return 0
```

---

#### 6. Evaluación del sistema

Compara las etiquetas generadas por tus reglas con las etiquetas reales (label).

 Métricas que debes calcular:


- **Accuracy**

- **Precision**
- **Recall**
- **F1-score**

 *Librería recomendada:* sklearn.metrics

Ejemplo:

```
from sklearn.metrics import classification_report
print(classification_report(y_true, y_pred))
```

 **Objetivo:** entender qué reglas funcionan mejor y cuáles fallan.

---

## 7. Análisis y conclusiones

Redacta un pequeño análisis:

- Qué tipos de reglas fueron más efectivas.
- Ejemplos de errores y por qué ocurrieron.
- Cómo podrías mejorar el sistema.

 **Mini-report (3 páginas)**

Debe incluir:

1. Descripción del dataset.
  2. Conocimiento lingüístico extraído.
  3. Reglas definidas y justificación.
  4. Resultados (métricas y observaciones).
  5. Conclusiones.
- 

## 8. Preparar presentación (5 + 5 min)

Crea unas diapositivas simples con:

- Objetivo del sistema.
- Reglas principales.
- Ejemplos de clasificación.
- Resultados y errores típicos.

## 🧠 Milestone 1 = Sistema basado en reglas (sin entrenamiento)

Efectivamente ✅

👉 **NO entrenas ningún modelo.**

👉 Lo que haces es **diseñar reglas lingüísticas** que intenten imitar el razonamiento humano para decidir si un comentario es ofensivo o no.

El enfoque es totalmente **determinístico**, no estadístico.

Por ejemplo:

Texto	Regla aplicada	Resultado
“Seu idiota, cala a boca!”	Contiene palabra ofensiva (“idiota”)	1 (Ofensivo)
“Bom dia, presidente!”	No contiene insultos ni negatividad	0 (No ofensivo)

Estas reglas las defines **tú**, basándote en lo que observes en el dataset.

---

## ⚙️ Cómo se evalúa si no hay modelo

Aunque no entrenes un modelo, **sí puedes medir el rendimiento** de tu sistema.

Tienes **dos conjuntos de etiquetas**:

1. ◆ `y_true` → las etiquetas reales del dataset (label en HateBR).
2. ◆ `y_pred` → las etiquetas que produce tu función de reglas.

Ejemplo conceptual:

# Etiquetas reales

```
y_true = [0, 1, 0, 1]
```

# Etiquetas generadas por tu función basada en reglas

```
y_pred = [0, 1, 0, 0]
```

Con esto, puedes calcular las métricas igual que con un modelo de ML:

```
from sklearn.metrics import classification_report
```

```
print(classification_report(y_true, y_pred))
```

Esto te dará:

	precision	recall	f1-score	support
0 (no ofensivo)	0.90	1.00	0.95	2000
1 (ofensivo)	0.85	0.60	0.70	800

---

## Qué significan estas métricas aquí

**Métrica**      **Qué indica en tu caso**

**Precision (1)**      Qué porcentaje de los comentarios que tu sistema marcó como ofensivos realmente lo eran.

**Recall (1)**      Qué porcentaje de los comentarios ofensivos reales tu sistema detectó.

**F1-score**      Media entre precisión y recall: mide el equilibrio.

**Accuracy**      Porcentaje total de aciertos.

Así puedes decir, por ejemplo:

“El sistema basado en reglas obtiene un F1-score de 0.72, siendo más eficaz detectando lenguaje no ofensivo que ofensivo.”

## Resumen general de tus prácticas y su relación con el proyecto


### Práctica 1 – Procesamiento básico del lenguaje

#### Contenidos:

- Limpieza de texto (eliminación de stopwords, signos de puntuación, etc.).
- Tokenización.
- Normalización (minúsculas, stemming/lematización).
- Frecuencia de palabras y n-gramas.

#### Relación con Milestone 1:

Altamente útil.

 Aquí tienes todas las **técnicas iniciales de preprocesamiento** y análisis que debes aplicar sobre el dataset HateBR antes de definir tus reglas.

Te servirá para:

- Conocer las palabras más frecuentes en textos ofensivos/no ofensivos.
- Crear listas de insultos o patrones lingüísticos.
- Analizar la estructura de los comentarios.

→ **Debes basarte principalmente en esta práctica para la primera parte del Milestone 1 (análisis de datos y exploración lingüística).**

---

## Práctica 2 – Análisis gramatical y extracción de información

### ✦ Contenidos:

- Etiquetado gramatical (POS tagging) con *spaCy*.
- Reconocimiento de entidades (NER).
- Análisis de dependencias.
- Relaciones entre palabras y sentimiento.

### ✅ Relación con Milestone 1:

Muy relevante.

👉 Te servirá para la segunda parte del Milestone 1, donde necesitas **extraer conocimiento lingüístico** (como verbos, nombres, entidades o polaridad emocional) para definir tus reglas de clasificación.

→ **Puedes usar esta práctica para crear reglas basadas en POS o sentimiento** (por ejemplo: “si hay adjetivos insultantes o verbos agresivos, clasificar como ofensivo”).

---

## 🤖 Práctica 3 – Clasificación con Machine Learning

### ✦ Contenidos:

- Vectorización del texto (TF-IDF, CountVectorizer).
- Entrenamiento de modelos (Naive Bayes, SVM, Logistic Regression).
- Evaluación con métricas.

### ❌ Relación con Milestone 1:

Poca o ninguna.

👉 Esta práctica corresponde exactamente al **Milestone 2 del proyecto**, donde usarás modelos de ML.

En el Milestone 1 no debes entrenar modelos, solo usar reglas.

→ **No la utilices por ahora, pero te servirá en la siguiente fase.**

---

## 🧩 Práctica 4 – Uso de modelos preentrenados y LLMs

### ✦ Contenidos:

- Uso de modelos tipo BERT o GPT.
- Prompt Engineering.
- Inferencia con modelos de lenguaje.

### ❌ Relación con Milestone 1:

No aplicable todavía.

👉 Esto será útil para el **Milestone 3**, donde aplicarás Prompt Engineering sobre modelos de hasta 8B parámetros.

→ Guárdala para el último Milestone.

## PARA TRABAJAR EN EL PROYECTO CON MI COMPAÑERO TENEMOS DOS OPCIONES:

### ✓ 1. La mejor opción: usar GitHub como repositorio compartido

#### Ventajas

- Ambos trabajáis sobre **los mismos archivos y versiones**.
- Podéis **sincronizar cambios** sin pisar el trabajo del otro.
- Se mantiene un **historial de modificaciones** (útil para revisiones).
- Permite incluir el código, dataset, informes y notebooks en un mismo proyecto.

#### Estructura recomendada del repositorio

PLN\_Proyecto/

```
|
|
├── data/          # Datasets (HateBR.csv, etc.)
├── notebooks/
|   ├── milestone1.ipynb
|   ├── milestone2.ipynb
|   └── milestone3.ipynb
├── src/           # Código Python auxiliar (si se separa del notebook)
├── reports/       # Mini informes y entregas parciales
└── README.md      # Descripción general del proyecto
```

#### Flujo de trabajo recomendado

1. **Uno crea el repositorio en GitHub** (público o privado).
2. El otro **hace un fork** o se añade como colaborador.
3. Ambos **clonan el repositorio** en su Colab o entorno local:

```
!git clone https://github.com/usuario/PLN_Proyecto.git
```

```
%cd PLN_Proyecto
```

4. Cada uno **trabaja en una rama distinta**, por ejemplo:
  - rama-fran
  - rama-compañero

```
!git checkout -b rama-fran
```

5. Cuando terminas tu parte, haces:

`!git add .`

`!git commit -m "Avance Milestone 1 - EDA y limpieza"`

`!git push origin rama-fran`

6. Tu compañero hace lo mismo en su rama.

7. Cuando ambos hayáis terminado, podéis **fusionar (merge)** las ramas en la rama principal (main o master) mediante un *pull request*.

## ✅ 2. Combinación profesional (muy recomendable para un TFG o proyecto universitario)

**Usar GitHub + Google Colab:**

- Guardáis el repositorio en GitHub.
- Cada notebook se abre en Colab usando la URL directa:
- [https://colab.research.google.com/github/usuario/PLN\\_Proyecto/blob/main/notebooks/milestone1.ipynb](https://colab.research.google.com/github/usuario/PLN_Proyecto/blob/main/notebooks/milestone1.ipynb)
- Colab permite editar directamente el notebook de GitHub.
- Cuando termináis, hacéis “Guardar una copia en GitHub” desde el menú *Archivo*.

Este enfoque combina:

- **Colaboración online** en tiempo real.
- **Control de versiones profesional**.
- **Acceso desde cualquier dispositivo**.

### 1. Por qué una sola rama es la mejor opción en vuestro caso

- Vuestro trabajo es **secuencial**, no paralelo.
- El compañero B necesita la versión final del código de A para continuar.
- No tiene sentido mantener dos ramas distintas si uno depende del otro.
- Mantener todo en la rama main simplifica la sincronización y la entrega final.

---

### 2. Flujo de trabajo recomendado (caso “trabajo en cadena”)

**Fase 1 – Tú (persona A)**

1. **Clonas** el repositorio una vez en Colab:
2. `!git clone https://github.com/usuario/PLN_Proyecto_TextClassification.git`



3. `%cd PLN_Proyecto_TextClassification`
4. Creas tu notebook, por ejemplo `milestone1_EDA.ipynb`.
5. Trabajas normalmente en Colab.
6. Cuando termines:
  - En Colab, haz **Archivo** → **Guardar una copia en GitHub**.
  - Selecciona la rama main.
  - Escribe un mensaje de commit descriptivo (por ejemplo *“Finalizado Milestone 1 – EDA y limpieza”*).

GitHub se actualiza automáticamente.

---

## Fase 2 – Tu compañero B

1. Antes de empezar, **descarga la versión más reciente** del repositorio.  
Si trabaja desde Colab, simplemente **vuelve a clonarlo** (o borra el anterior y repite):
  2. `!git clone https://github.com/usuario/PLN_Proyecto_TextClassification.git`
  3. `%cd PLN_Proyecto_TextClassification`
  4. Abre el notebook que tú terminaste, por ejemplo:
  5. `notebooks/milestone1_EDA.ipynb`
  6. Crea un **nuevo notebook** basado en él para continuar, por ejemplo:
  7. `notebooks/milestone2_ML.ipynb`
  8. Trabaja sobre esa base, añadiendo su parte del proyecto.
  9. Cuando termine, guarda de nuevo la copia en GitHub en la misma rama main.
- 

## 3. Qué debe hacer tu compañero exactamente antes de trabajar

Si ya clonó el repositorio previamente y tú hiciste cambios nuevos:

```
%cd PLN_Proyecto_TextClassification
```

```
!git pull origin main
```

Esto actualiza su copia local con los cambios que tú subiste.

Si está trabajando en Colab, basta con **volver a abrir el notebook desde la URL del repositorio**, y Colab cargará automáticamente la última versión del archivo.

---

## 4. Buenas prácticas recomendadas

Situación	Recomendación
Cada Milestone terminado	Guardar con un commit claro en main.
Antes de empezar una nueva sesión	Hacer git pull origin main o reabrir desde GitHub para tener la última versión.
Modificaciones simultáneas en el mismo archivo	Evitarlas; mejor que cada uno trabaje en un notebook diferente (milestone1, milestone2, etc.).
Comunicación	Indicar siempre cuándo se suben cambios al repositorio.
Documentación	Actualizar el README.md con los avances de cada Milestone.

---

## 5. Ventajas de este flujo

- Simple: solo una rama (main), sin merges ni conflictos.
- Seguro: cada commit conserva el historial.
- Reproducible: el compañero B siempre trabaja sobre la última versión estable.
- Compatible con entregas por hitos (cada commit puede corresponder a un Milestone).

---

## 6. Desventajas (y cómo mitigarlas)

Posible problema	Solución
Riesgo de sobrescribir trabajo si ambos editan el mismo notebook	Evitar trabajar simultáneamente en el mismo archivo.
No se puede probar código nuevo sin afectar al principal	En ese caso puntual, crear una rama temporal (rama-test) y luego eliminarla.

---

## Resumen rápido del flujo

1. Tú (A) haces Milestone 1 → lo subes a main.
2. Tu compañero (B) hace git pull o vuelve a abrir el notebook → empieza Milestone 2.
3. Cuando termina, guarda en GitHub en la misma rama.
4. Así sucesivamente hasta el Milestone 3.