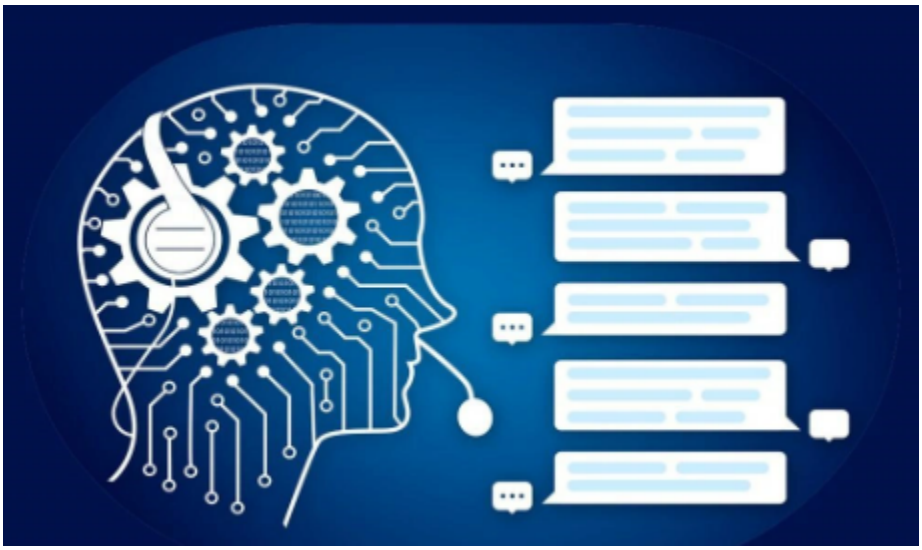
MEMORIA DEL PROYECTO FINAL: CLASIFICACIÓN DE TEXTOS OFENSIVOS

Procesamiento del Lenguaje Natural



Marco Antonio Carrión Soriano

macs0021@red.ujaen.es

Juan Bautista Muñoz Ruiz

jbmr0001@red.ujaen.es

Índice

1- Introducción

En este proyecto se ha trabajado en la implementación de un sistema de PLN clasificador binario de texto. En nuestro caso elegimos el problema centrado en el dominio de identificar si un texto es ofensivo haciendo uso del corpus MeOfendES. Por lo que este sistema está entrenado para categorizar un conjunto de palabras como ofensivo “OFF” o no ofensivo “NON”.

2- Diseño y Desarrollo

2.1- Lectura de Archivos

Disponemos de una carpeta de trabajo con todo el material para nuestro sistema como el corpus, el lexicón y demás archivos de salida. El lexicón usado fue lexicón SHARE de palabras ofensivas del grupo SINAI obtenido en el siguiente enlace.

Cabe destacar que también pensamos que sería buena idea usar el lexicón de emoticonos para el análisis de opiniones del mismo grupo de investigación, sin embargo no estaba disponible.

```
import os #Abrimos la carpeta
path = os.chdir("/content/drive/MyDrive/PLN/Proyecto")
os.getcwd()
```

- **Conjunto train y dev:** Con el uso de la biblioteca Glob se implementó la lectura automatizada de ficheros. Por otro lado, con la biblioteca Csv realizamos la apertura para lectura de los archivos tsv y su guardado en un vector con la información de cada columna llamado listaProcesada. Seguidamente, toda la información obtenida era encapsulada en un diccionario con cada apartado del comentario (id, texto, ofensividad, género y plataforma). Finalmente, obtendremos un vector de Comentarios train y otro de Comentarios dev

```
for filename in glob.glob('*.tsv'):  #Lectura de todos los archivos xlsx
    print(filename)
    if filename!="test.tsv":
      with open(filename,encoding='utf-8',errors='ignore') as file:
        tsv_file = csv.reader(file, delimiter="\t", quotechar='"')
        with open(filename+".txt", 'w') as f:
            totales=0
            for line in tsv_file:
                listaProcesada=line
                totales=totales+1

                if len(listaProcesada) == 5 and (listaProcesada[2] == 'OFF' or listaProcesada[2] == 'NON'):
                    listaProcesada[1]=tokenizarTexto(listaProcesada)
                    mapaLineas[filename]=listaProcesada
                    comentario = {
                        'id': listaProcesada[0],
                        'texto': listaProcesada[1],
                        'ofensividad': listaProcesada[2],
                        'genero': listaProcesada[3],
                        'plataforma': listaProcesada[4]
                    }

                    if(filename=="train.tsv"):
                      comentarios_training.append(comentario)
                    else:
                      comentarios_dev.append(comentario)
```

- **Lexicón (SHARE):** Este lexicón está compuesto por 5,888 unigramas, 2,447 bigramas y 1,790 trigramas ofensivos. Se realizó una lectura del archivo y se almacenó cada fila del archivo en un vector para propio para el lexicón. En otro vector paralelo asignamos las etiquetas para cada fila del lexicón “OFF”.

```
elementosLexicon=[]
etiquetasLexicon=[]
with open('Lexicón.txt', 'r') as archivo:
    for linea in archivo.readlines():
      etiquetasLexicon.append("OFF")
      elementosLexicon.append(tokenizar(linea))
      #elementosLexicon.append(linea)
```

- **Conjunto test:** Aquí se reutilizó la misma función para la lectura de los textos dev y train pero adaptándola para la lectura única del archivo “test.tsv” y con el formato de este:

```
filename="test.tsv"
```

```
with open(filename+".txt", 'w') as f:
    for line in tsv_file:
        listaProcesada=line
        totales=totales+1
        listaProcesada[1]=tokenizarTexto(listaProcesada)
        comentario = {
                'id': listaProcesada[0],
                'texto': listaProcesada[1],
                'ofensividad': "",
                'genero': listaProcesada[2],
                'plataforma': listaProcesada[3]
            }
```

Para finalizar, cabe destacar que todos los archivos leídos eran sometidos a la misma función de procesamiento de texto (tokenizar(texto)). Todas estas funciones serán explicadas en el siguiente punto.

2.2- Preprocesamiento

Para afrontar el problema propuesto se han planteado cuatro estrategias de preprocesamiento de texto:

- **Preprocesamiento 1.A (Procesamiento normal):** Mediante el uso de la biblioteca Spacy con la función nlp() y el modelo Spacy pre-entrenado "es_core_news_sm" nos hemos quedado con los tokens que son alfabéticos y no son stop words. Seguidamente, se han lematizado (reducción a la raíz) y pasado a minúscula.

```
import spacy.cli
spacy.cli.download("es_core_news_sm")
```

```
def tokenizarTexto(lista):
    texto=nlp(lista[1])
    tokens=[]
    textoProcesado=""
    for token in texto:
        if token.text.isalpha() and not token.is_stop: #Quitamos signos de puntuación y stop w
            #tokens.append(token.text.lower()) #la pasamos a minúscula
            tokens.append(token.lemma_.lower()) #Reducimos a la raiz cada palabra y la pasamos
    textoProcesado=" ".join(tokens)
    return textoProcesado
```

- **Preprocesamiento 1.B (Procesamiento 1.A sin lematización):** Se trata de una estrategia muy parecida a la anterior. Con el uso de la biblioteca Spacy con la función nlp() y el modelo Spacy pre-entrenado "es_core_news_sm" nos hemos quedado con los tokens que son alfabéticos y no son stop words. Sin embargo, aquí no hemos lematizado estos tokens. Únicamente los pasamos a minúscula.

```
def tokenizarTexto(lista):
    texto=nlp(lista[1])
    tokens=[]
    textoProcesado=""
    for token in texto:
        if token.text.isalpha() and not token.is_stop: #Quitamos signos de puntuación y stop
            tokens.append(token.text.lower()) #la pasamos a minúscula
            #tokens.append(token.lemma_.lower()) #Reducimos a la raiz cada palabra y la pasa
    textoProcesado=" ".join(tokens)
    return textoProcesado
```

- **Preprocesamiento 2 (Quitando palabras ofensivas de textos "NON"):** Hemos usado la misma base del preprocesamiento 1, pero en este caso hemos usado el

lexicón. La idea principal de esta estrategia era eliminar palabras ofensivas de los textos que fueran no ofensivos (NON). Por cada token obtenido de la función nlp() de spacy nos quedamos con los que son alfabéticos y no son stop words. Seguidamente, si la etiqueta de ese texto de entrenamiento era “NON” sólo guardamos los tokens que no aparezcan en el lexicón. En caso de ser ofensivo, se seguía el mismo proceder que en el procesamiento 1.A. Posteriormente, estos tokens serían lematizados y pasados a minúscula.

```
def tokenizarTexto(lista,filename):

    print(filename)
    texto=nlp(lista[1])
    tokens=[]
    textoProcesado=""
    for token in texto:
        if token.text.isalpha() and not token.is_stop: #Quitamos signos de puntuación y stop w

          ##############################QUITAMOS DE textos NON los insultos####################

            if lista[2]=="NON" and filename=="train.tsv":
              if token.lemma_.lower() not in elementosLexicon:
                tokens.append(token.lemma_.lower()) #Reducimos a la raiz cada palabra y la pas
              else:
                print("------Quitada de texto NON:",token.lemma_.lower(),filename)
            else:
              tokens.append(token.lemma_.lower()) #Reducimos a la raiz cada palabra y la pasam

          ##############################QUITAMOS DE textos NON los insultos####################

    textoProcesado=" ".join(tokens)
    return textoProcesado
```

- **Preprocesamiento 3 (Guardando los emojis):** Usando la misma base del procesamiento 1.A. Sin embargo, la característica diferenciadora es que en este procesado teníamos en cuenta los emojis del texto. El procesamiento está compuesto por los siguientes pasos:
    - Procesamiento de emojis: Hemos hecho el uso de la biblioteca Emoji para detección de estos. Toda palabra que fuera un emoji era convertida a formato textual Unicode. Posteriormente estos emojis en formato texto eran añadidos al conjunto final de tokens del texto.

- Procesamiento de texto: Esta parte es el mismo pre-procesamiento seguido en la estrategia 1.A. Nos quedamos con las palabras alfabéticas que no son stop words y posteriormente las lematizamos, pasamos a minúscula e insertamos en el conjunto final de tokens.

```
def tokenizarTexto(lista):
     ###################################Emojis#################
    emojis=[]
    for palabra in lista[1]:
      if emoji.is_emoji(palabra):
        emojis.append(emoji.demojize(palabra)) #lo pasamos a texto para que lo lea el modelo
        #print(emoji.demojize(palabra))
     ###################################Emojis#################
    texto=nlp(lista[1])
    tokens=[]
    textoProcesado=""
    for token in texto:
        if token.text.isalpha() and not token.is_stop: #Quitamos signos de puntuación y stop
            tokens.append(token.lemma_.lower()) #Reducimos a la raiz cada palabra y la pasam
    ###################################Emojis#################
    for emo in emojis:
      #print(emoji)
      tokens.append(emo)
    ###################################Emojis#################
    textoProcesado=" ".join(tokens)
    print(textoProcesado)
    return textoProcesado
```

Obtenemos un texto procesado que incluye tokens lematizados y emojis.

Después de implementar esos pre-procesamientos los sometimos a experimentación para quedarnos con el mejor, extrajimos las siguientes conclusiones:

- En la comparativa del procesamiento 1.A vs 1.B el que mejores resultados obtenía era el 1.A, por lo que este sería el procesamiento de texto a usar en la clasificación así como base para los procesamientos 2 y 3.
- El procesamiento 1.B era peor ya que al no lematizar las palabras reduciendolas a su raíz los modelos clasificadores estaban perdiendo instancias de las palabras al estar en diferentes formatos de tiempos verbales, géneros, etc...
- En la comparativa del 1.A. vs 2 vs 3, el mejor era el 1.A:

- En cuanto al procesamiento 2, el hecho de quitar las palabras ofensivas de los textos de entrenamiento no ofensivos hacía que frases que contienen palabras ofensivas pero con finalidad no ofensiva como “te quiero fea” se perdían por lo que se veía afectada la precisión de nuestros modelos.
- En cuanto al procesamiento 3. Simplemente obtuvimos peores resultados. Creemos que es porque los modelos utilizados en la posterior clasificación están optimizados para la clasificación de textos con elementos del lenguaje natural. Con lo que meter palabras en formato Unicode con los emojis perderían eficiencia los modelos.

Por otro lado también se probó la biblioteca Nltk para el preprocesamiento de los textos:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
```

```
# Preprocesar los comentarios de entrenamiento
def preprocess_text(comment):
    tokens = word_tokenize(comment.lower())
    filtered_tokens = [token for token in tokens if token.isalpha() and token not in stop_words]
    preprocessed_text = ' '.join(filtered_tokens)
    return preprocessed_text
```

Sin embargo los resultados obtenidos eran idénticos ya que las frases resultantes eran las mismas para los dos modelos.

3- Experimentación

3.1- Métricas utilizadas para la comparación de algoritmos

Para comparar los diferentes algoritmos que comentaremos a continuación, se han utilizado diferentes métricas, las cuales nos permiten saber qué algoritmo es superior a qué otro algoritmo. Esas métricas son las siguientes:

1. **Precisión:** La precisión se refiere a la proporción de verdaderos positivos (las predicciones correctas de la clase positiva) sobre el total de predicciones positivas realizadas por el modelo. Es decir, de todas las instancias que el modelo predijo

como positivas, ¿cuántas realmente lo eran? Se calcula como: Precisión = Verdaderos Positivos / (Verdaderos Positivos + Falsos Positivos)

2. **Recall:** El recall, también conocido como sensibilidad o tasa de verdaderos positivos, se refiere a la proporción de verdaderos positivos sobre el total de casos positivos reales en los datos. Es decir, de todas las instancias que eran realmente positivas, ¿cuántas pudo identificar el modelo? Se calcula como: Recall = Verdaderos Positivos / (Verdaderos Positivos + Falsos Negativos)
3. **F1-Score:** El F1 score es la media armónica de la precisión y el recall. Este valor será alto si tanto la precisión como el recall son altos. La media armónica tiene la propiedad de ser baja si cualquiera de sus componentes es baja, lo que significa que si cualquiera de la precisión o el recall es baja, el F1 score también será bajo. Esto es útil porque quieres que tu modelo tenga tanto una alta precisión como un alto recall. Se calcula como: F1 Score = 2 * (Precisión * Recall) / (Precisión + Recall)

Como podemos observar, el F1-Score depende tanto de la precisión como del Recall, por lo tanto es una métrica muy útil para saber cual método es superior a otro. Esta será la métrica que utilizaremos para comparar métodos entre sí.

Para mostrar dichas métricas utilizamos el módulo metrics de Sklearn, el cual muestra las métricas de la siguiente manera:

```
Precisión: 0.84
Macro-f1 0.7588908981314044
Recall 0.7297297297297297
              precision    recall  f1-score   support

         NON       0.85      0.96      0.90        74
         OFF       0.81      0.50      0.62        26

    accuracy                           0.84       100
   macro avg       0.83      0.73      0.76       100
weighted avg       0.84      0.84      0.83       100
```

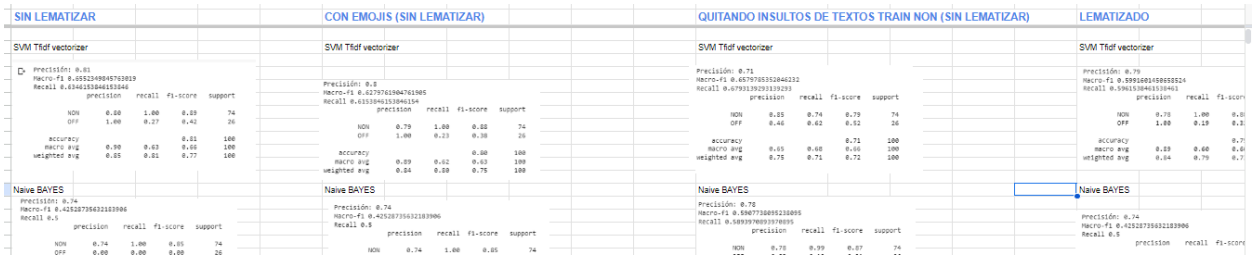
Además, estas métricas pueden ser globales o de clase, lo que significa que se pueden aplicar al total del modelo o a cada tipo de clase/etiqueta, lo que nos puede ayudar a ver la eficiencia del modelo a nivel general y la eficiencia del modelo para cada tipo de etiqueta,

dejando ver si el modelo tiene deficiencias a la hora de categorizar una etiqueta en concreto.

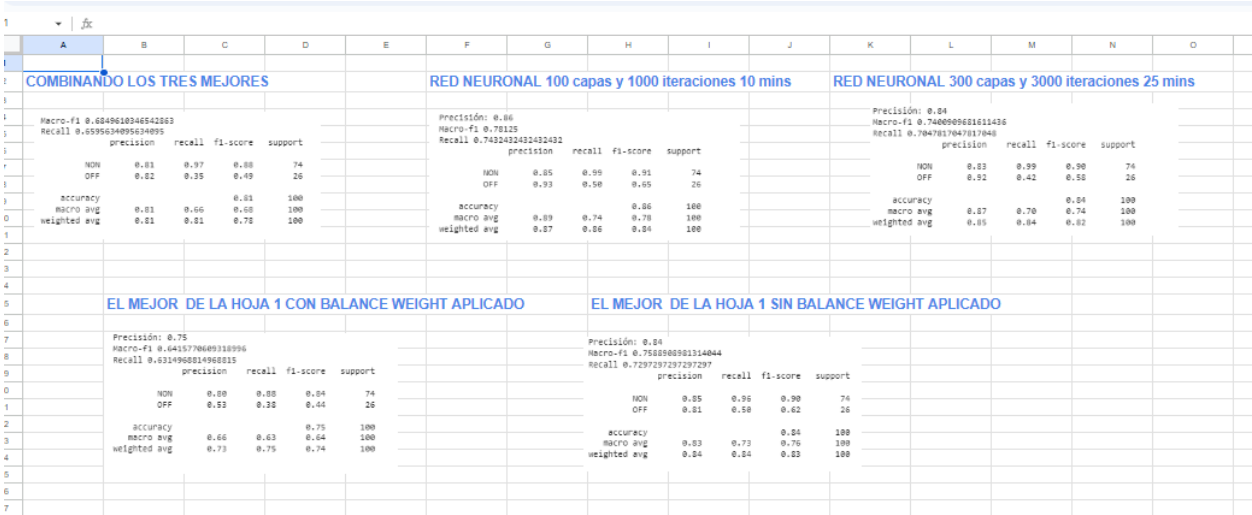
```
# Imprimir la precisión
print(f"Precisión: {precision}")
print(f"Macro-f1", f1_score(ofensividad_comentarios_dev, predictions, average='macro'))
print(f"Recall", recall_score(ofensividad_comentarios_dev, predictions, average='macro'))
report = classification_report(ofensividad_comentarios_dev, predictions)
print(report)
```

3.2- Comparación de los diferentes algoritmos

Se han ido recopilando algunas de las salidas de los diferentes pre-procesamientos, vectorizadores y modelos de cara a sus análisis en el siguiente archivo xlsx. En este documento contamos con dos hojas. En la primera, la comparativa inicial de todos los pre-procesamientos junto con todos los modelos y vectorizadores.



En la segunda hoja, una comparación más generalista de diferentes modelos:



4- Modelos Utilizados

4-1 Combinaciones de modelo y vectorizador

Para la selección de modelos se han investigado los diferentes tipos de modelos supervisados y los distintos vectorizadores que existen. Una vez recopilados estos tipos de modelos y vectorizadores, se realizó un algoritmo que ejecutaba todas las combinaciones modelo-vectorizador posibles y mostraba sus métricas. De esta forma, pudimos ver qué tipo de modelo supervisado y que vectorizador hacían la mejor combinación.

Los métodos utilizados fueron:

1. **Regresión logística:** Es un modelo de clasificación que calcula la probabilidad de que una instancia pertenezca a una clase. Si la probabilidad estimada es mayor a 50%, entonces el modelo predice que la instancia pertenece a esa clase y en caso contrario predice que la instancia no pertenece a esa clase en concreto.
2. **SVM:** Estos modelos buscan la línea que separa de forma óptima a dos clases, de modo que la distancia entre las instancias más cercanas de cada clase (los vectores de soporte) sea la máxima posible.
3. **K vecinos más cercanos:** Identifica las instancias que son más cercanas a una nueva instancia y asigna a la nueva instancia la clase más común entre esos K vecinos. Es decir, compara la nueva instancia con las instancias anteriores para saber a qué grupo pertenece.
4. **Árbol de decisión:** Genera un árbol de decisión en el que cada nodo es una característica de los datos de entrenamiento, es decir, una decisión y cada rama representa una respuesta, hasta llegar a los nodos hoja que representan las predicciones.
5. **Naive Bayes:** Realiza cada predicción basándose en la probabilidad. Cada predicción es independiente de la anterior, ya que estamos usando la variante Naive.

El que mejor resultado de estos métodos dió fue el 4. Árbol de decisión.

Los vectorizadores utilizados fueron:

1. **CountVectorizer**: Este método convierte una colección de documentos de texto en una matriz de conteos de tokens. Es decir, toma cada palabra única en todos los documentos y cuenta cuántas veces aparece en cada documento.
2. **TfidfVectorizer:** Convierte una colección de documentos de texto en una matriz de características, pero en lugar de simplemente contar las apariciones de cada palabra, divide estas cuentas por la frecuencia con la que aparecen las palabras en todos los comentarios. Esto significa que las palabras que aparecen en muchos comentarios diferentes tendrán un peso más bajo, mientras que las palabras que aparecen en pocos tendrán un peso más alto.
3. **HashVectorizer:** Al igual que los anteriores, HashingVectorizer convierte una colección de documentos de texto en una matriz de características. Sin embargo, en lugar de almacenar directamente las palabras y sus conteos, utiliza una técnica llamada "hashing" para convertir las palabras en números enteros. Su principal desventaja es que no puedes recuperar las palabras originales a partir de los números.

El que mejores resultados dió fue CountVectorizer.

La mejor combinación general fue el árbol de decisión - CountVectorizer.

```python
# Definir los métodos y vectorizadores a probar
methods = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'SVM': SVC(),
    'K-Nearest Neighbors': KNeighborsClassifier(),
    'Decision Tree': DecisionTreeClassifier(random_state=62),
    'Naive Bayes': GaussianNB(),
    # Agrega más métodos aquí
}

vectorizers = {
    'CountVectorizer': CountVectorizer(),
    'TfidfVectorizer': TfidfVectorizer(),
    # Agrega más vectorizadores aquí
}

# Entrenar y evaluar los modelos para cada combinación de método y vectorizador
resultados = []
for (method_name, method_model), (vectorizer_name, vectorizer_model) in product(methods.items(), vectorizers.items()):
    # Crear el vectorizador y transformar los comentarios de entrenamiento
    X_train = vectorizer_model.fit_transform(preprocessed_train_comments)

    # Entrenar el modelo con los datos de entrenamiento
    method_model.fit(X_train, train_labels)

    # Transformar los comentarios de prueba utilizando el vectorizador
    X_dev = vectorizer_model.transform(preprocessed_dev_comments)

    # Predecir las etiquetas de prueba utilizando el modelo entrenado
    y_pred = method_model.predict(X_dev)

    # Obtener el informe de clasificación para evaluar el modelo
    report = classification_report(dev_labels, y_pred, output_dict=True)
    resultados.append((method_name, vectorizer_name, report))
    print("Método: ", method_name, " Vectorizador: ", vectorizer_name, " === ", report)
```

4.2- Diferente procesamiento de los datos

Como hemos hablado anteriormente, se han probado diferentes procesamientos de los datos:

1. **Procesamiento normal:** Este es el método que usamos finalmente. En él, como se ha hablado anteriormente, se lematizan las palabras y se reducen a minúscula, además de eliminar las stopwords. Este método nos dió buenos resultados ya que consigue que una misma palabra ofensiva en diferentes derivaciones pueda ser reconocida por el modelo como la misma palabra, lo que ayuda a reconocer patrones.
2. **Procesamiento sin lematizar:** Este método se intentó basándose en la siguiente premisa, “Hay palabras que son ofensivas o no según qué palabra derivada sea del lexema principal”. Por ejemplo, “tontito” podría no considerarse ofensiva, si no un apelativo cariñoso, ya que está en diminutivo, mientras que “tonto” si que tendría más posibilidades de estar en un mensaje ofensivo. El problema de este tipo de

procesamiento es que el modelo no relaciona las palabras ofensivas del mismo lexema como similares, por lo que finalmente decidimos no utilizar este método.

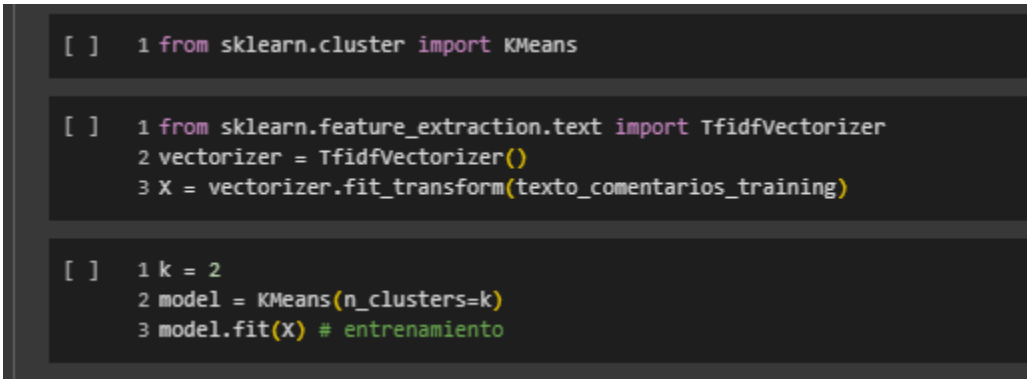
3. **Quitar palabras ofensivas de los textos NON:** Este procesamiento se basa en quitar las palabras ofensivas de los textos no ofensivos, con el fin de que se resalte más la diferencia entre mensajes ofensivos y no ofensivos. Para ello utilizamos el lexicón de palabras ofensivas. Si un elemento del lexicón está en un texto no ofensivo, este elemento se elimina del texto. Aunque los modelos dieron buen resultado con este preprocesamiento y posterior procesado, decidimos no utilizarlo, puesto que entendíamos que el modelo sería mucho menos permisivo con los textos no ofensivos, es decir, cualquier texto con una palabra que se considera ofensiva fuera de contexto sería catalogado como ofensivo, cuando en la realidad esto depende del contexto y de cómo se utiliza esa palabra que a priori parece ofensiva.
4. **Pre-procesamiento con emojis:** Como se ha explicado en las primeras páginas se trata del preprocesamiento normal pero ampliado con emoticonos como tokens.
5. **Añadir lexicón de palabras ofensivas a la muestra de palabras ofensivas:** Aunque este no es un preprocesamiento como tal, es un método que probamos sobre el conjunto de datos. Este método consistía en añadir al conjunto de datos catalogados como ofensivos en training, el conjunto de datos del lexicón, añadiendoles la etiqueta OFF. Este método no nos dio buenos resultados, suponemos que debido a que, al ser el lexicón palabras sueltas, el modelo no podía establecer relaciones dentro de la oración, puesto que la oración en sí no existía, lo cual empeoraba los resultados. Al tener sólo los datos ofensivos de training, que si son mensajes con relaciones entre palabras, el modelo aprendía de una forma más exacta.

4.3- Otros métodos utilizados

Una vez probamos todas las combinaciones Modelo - Vectorizador - Preprocesamiento y obtuvimos los resultados no nos quedamos ahí. Exploramos otras posibilidades, las más destacables son las siguientes:

1. **Modelo no supervisado:** Aunque suponíamos que no daría muy buenos resultados, probamos a realizar un modelo no supervisado, en el cual se dividían los

mensajes en clusters de palabras. En concreto nosotros dividimos los resultados en dos Clusters, para que el modelo los separase en ofensivos y no ofensivos. Aunque la precisión dió resultados aceptables, las demás métricas, tanto globales como de clase dieron un resultado bastante pobre.



2. **Red neuronal (MLP):** La red neuronal ha sido uno de los modelos que hemos entregado, concretamente una Multi Layer Perceptron. Fue elegida esta ya que suele ser usada para la clasificación. Esta red se puede configurar mediante parámetros para que se ajuste a las necesidades del problema. Estos parámetros son los siguientes:
    - **Número de capas ocultas:** Este parámetro como su nombre indica ajusta el número de capas ocultas, que son las capas que realizan las operaciones y ajustes en los valores para la obtención de resultados.
    - **Número de iteraciones:** Es el número de ejecuciones que realiza el algoritmo.

    Aunque el tiempo de ejecución es bastante extenso, los resultados pueden llegar a ser bastante buenos, como demuestran los datos obtenidos. En concreto, probamos varios valores de los parámetros, valores bajos, medios y altos para ambos parámetros, siendo el mejor resultado obtenido en valores 100 para las capas ocultas y 1000 para las iteraciones.

    Este método recibe datos de la misma manera que el resto de modelos.

```
[ ]  1 from sklearn.neural_network import MLPClassifier
     2 from sklearn.feature_extraction.text import CountVectorizer
     3
     4 vectorizer = CountVectorizer()
     5 X_train = vectorizer.fit_transform(texto_comentarios_training)
     6 X_test = vectorizer.transform(texto_comentarios_dev)
     7
     8 model = MLPClassifier(hidden_layer_sizes=(100,), max_iter=1000, random_state=42)
     9 model.fit(X_train, ofensividad_comentarios_training)
```

3. **Modelo basado en la votación de los tres mejores:** En este modelo realizamos tres clasificaciones en paralelo con los mejores modelos. La etiqueta mayoritaria obtenida sería la etiqueta asignada a la predicción.

```
def juntaPredicciones(modelo1,vectorizer1,modelo2,vectorizer2,modelo3,vectorizer3):
    predictions=[]
    aciertos=[]
    for i, comentario in enumerate(texto_comentarios_dev):
      Y = vectorizer1.transform([comentario])
      prediction1 = modelo1.predict(Y)
      Y = vectorizer2.transform([comentario])
      prediction2 = modelo2.predict(Y)
      Y = vectorizer3.transform([comentario])
      prediction3 = modelo3.predict(Y)
      #print(prediction3[0])
      votaciones = [prediction1[0], prediction2[0], prediction3[0]] #elegimos la mas votada
      #votaciones = [prediction1[0], prediction1[0],prediction2[0], prediction3[0]] # le damos
      #votaciones = [prediction1[0],prediction2[0]] # los dos mejores
      prediction = max(set(votaciones), key=votaciones.count)
      #print(prediction)
      predictions.append(prediction)
```

```
juntaPredicciones(model,vectorizer,modelSVMTfidf,vectorizer2,modelDTHV,vectorizer3)
```

Además, como podemos ver en la parte comentada, se fue cambiando el peso de la votación de cada modelo aumentando o disminuyendo el número de instancias en el vector de votación. Sin embargo, el mejor resultado obtenido con este método no mejoraba al obtenido por el mejor modelo por separado. Era de esperar ya que al final estábamos bajando la probabilidad de que el mejor modelo asigne su predicción.

4. **Modelo probando el ajuste del atributo class_weight:** El hecho de tener más textos “NON” que “OFF” nos dio la idea de probar a usar este atributo para que el modelo no tuviera tendencia a favorecer a la etiqueta mayoritaria en el conjunto de entrenamiento. Como podemos ver, se usó el class_weight=balanced para evitar esto y se probó el class_weight en su versión parametrizada con diferentes valores. Sin embargo, los resultados obtenidos no mejoraron a los anteriores.

```
model = DecisionTreeClassifier(random_state=42,class_weight='balanced')
#model = DecisionTreeClassifier(random_state=42,class_weight={"NON": 1, "OFF": 100})
```

5- Mejores modelos

Tras someter a experimentación las diferentes combinaciones de modelos, vectorizadores y pre-procesamientos estos fueron los que mejores resultados obtuvieron y que acabaron siendo entregados:

- Pre-procesamiento normal + CountVectorizer + modelo Decision-Tree Classifier
- Pre-procesamiento normal + Tf-idfVectorizer + modelo SVM (Support Vector Machine)
- Pre-procesamiento normal + CountVectorizer + Red Neuronal MLP (Multi Layer Perceptron)

6- Bibliotecas Python utilizadas

- Emoji para la detección de emoticonos en el texto.
- Spacy para la tokenización de los textos.
- Glob para el recorrido de carpetas.
- Pandas y Nltk para una prueba de procesamiento alternativo.
- Csv para la lectura de los archivos tsv con el Corpus.
- SkLearn para los modelos clasificadores.