Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

Análisis de sentimiento en las redes sociales producido por las Fuerzas Armadas Españolas

Grado en Ingeniería Mecánica

ALUMNO: Alejandro Ortega de los Ríos

DIRECTORES: Milagros Fernández Gavilanes

Andrés Suárez García

CURSO ACADÉMICO: 2019-2020

Universidade Vigo

Centro Universitario de la Defensa
en la Escuela Naval Militar

TRABAJO FIN DE GRADO

Análisis de sentimiento en las redes sociales producido por las Fuerzas Armadas Españolas

Grado en Ingeniería Mecánica

Intensificación en Tecnología Naval

Cuerpo General

Universidade Vigo

RESUMEN

El análisis de sentimiento y emociones consiste en la aplicación del procesamiento del lenguaje natural para evaluar la información subjetiva del contenido de un texto. La información es clasificada en función de la connotación del lenguaje usado en el texto, atendiendo siempre a relaciones estadísticas y de asociación. En el presente TFG se propone realizar un análisis de sentimiento sobre las Fuerzas Armadas en la red social *Twitter*. Para ello se extraerá una cantidad determinada de *tweets* para su posterior análisis. Se implementarán modelos de aprendizaje con supervisión y sin supervisión para realizar el análisis, y se agrupará el texto de acuerdo a si comparten temas en común. Finalmente se procederá a evaluar la precisión del sistema de evaluación generado.

PALABRAS CLAVE

Análisis de sentimiento, Python, Machine Learning, LDA, NLP.

AGRADECIMIENTOS

A Dña. Milagros Fernández Gavilanes y D. Andrés Suárez García por orientarme en el desarrollo del TFG.
A los alféreces de fragata Diego Adolfo García Navarro y Juan Antonio Ramírez Peña por la ayuda prestada en el etiquetado de *tweets*.

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN Y OBJETIVOS

1.1 Descripción

La red social Twitter se trata de una red accesible de forma libre a una gran cantidad de usuarios, por lo que se puede afirmar que no está sesgada y existe por tanto gran variedad de datos y comentarios susceptibles de ser analizados. El interés de este TFG consiste en poder realizar un estudio de análisis de sentimiento tomando como *corpus* los *tweets* de las Fuerzas Armadas. Para ello se pretende desarrollar un programa con el lenguaje de programación Python que resuelva el problema enunciado.

1.2 Organización del TFG

Para realizar el estudio se dividirá el proyecto en los siguientes apartados, mostrados a continuación en Figura 1-1:

Figura 1-1 Esquema ilustrativo de las distintas etapas del proyecto (elaboración propia)

- Recopilación de datos: se procederá a la extracción de una determinada cantidad de tweets por medio de las herramientas para desarrolladores de las que Twitter dispone.
- Preprocesamiento: consiste en eliminar la parte de la información que no aporte significado al proyecto o sea redundante, además de normalizarla para posterior análisis.
- Modelado: consiste en entrenar un algoritmo para que sea capaz de valorar la polaridad (positiva, negativa o neutra) de un tweet en función de su contenido lingüístico. Para ello se emplearán los siguientes métodos de *Machine Learning:*
  - Aprendizaje sin supervisión: la IA que existe detrás del análisis de supervisión aprende de forma autónoma sin intervención de terceros.
  - Aprendizaje supervisado: la IA hace uso de una base de datos con tweets etiquetados con su respectiva polaridad. De este modo compara los nuevos tweets a evaluar con los antiguos para realizar un análisis a priori más preciso.
- Valoración de los resultados: en vista a los resultados obtenidos en el anterior apartado se procederá a una valoración de la fiabilidad del modelo generado.

- Conclusiones: se explicará de forma resumida y detallada los resultados, su fiabilidad, posibles mejoras y líneas futuras de desarrollo para futuros TFGs .

De este modo, se dividirá el código en una serie de funciones que realicen las tareas descritas en Figura 1-1 que dependan a su vez de un programa principal que invoque, de manera ordenada, cada una de estas funciones para la obtención de resultados.

1.3 Librerías empleadas

A continuación, se exponen las principales librerías importadas para el desarrollo del programa y su aplicación en este:

1. Tweepy: utilizada para importar *tweets* de la red social *Twitter*.
2. NLTK: empleado para eliminar palabras de parada.
3. Sklearn: empleada para extracción de características, entrenamiento algoritmos de aprendizaje con supervisión y obtención de resultados.
4. Polyglot: empleado para detección de sustantivos y aprendizaje sin supervisión.
5. VaderSentiment: usado para aprendizaje sin supervisión.
6. Texblob: es una librería de traducción de textos.
7. Gensim: generación de un modelo LDA.
8. Numpy: empleado para operaciones con matrices.
9. Matplotlib: empleada para representación gráfica y comparación de resultados.

2 ESTADO DEL ARTE

2.1 NLP

2.1.1 Definición

NLP (*natural language processing)*, es el campo que estudia las interacciones entre los ordenadores y el lenguaje natural de las personas: español, inglés, francés, etc. A día de hoy existen numerosos ejemplos de *softwares* basados en NLP como Siri o Cortana que son capaces de imitar el lenguaje humano e interactuar con los usuarios.

2.1.2 Funcionamiento

Analizar el lenguaje natural es una tarea harto complicada. Para ello se procede a construir una *pipeline,* lo cual consiste en dividir el procesamiento de texto en una serie de pasos que abordan una serie de problemas de menor entidad. De este modo se recurre a un «divide y vencerás». Para ello, se suelen seguir los siguientes pasos [1]:

1) Fragmentación del texto en oraciones: podemos asumir que cada frase contiene una idea propia distinta del resto pero que a su vez comparten un tema común. Evidentemente será mucho más sencillo analizar oraciones por separado que todo un párrafo.
2) *Tokenización* de palabras: consiste en dividir oraciones en palabras o *tokens*. Es importante señalar que los signos de puntuación (“.”, “…”, “,”, …) se pueden considerar *tokens*, pues modifican el significado del texto.
3) Etiquetación de las palabras (PoS): cada palabra queda clasificada de acuerdo con su categoría gramatical: sustantivo, adjetivo, verbo, etc.
4) *Lematización:* se trata de un proceso lingüístico por el cual se relacionan formas flexionadas de una palabra (femenino, plural, conjugada, etc.) con un lema, es decir, una palabra que represente al conjunto. La importancia de esta etapa radica en la necesidad de que el programa sea capaz de identificar palabras como “orgullo”, “orgulloso” o “orgullosas” como la misma palabra, pues comparten el mismo significado.
5) Identificar palabras de parada: las palabras de parada son palabras que aparecen con gran frecuencia en el texto como “de”, “y”, y “un”. Estas palabras introducen gran cantidad de ruido dado que, al repetirse de forma continuada, el modelo considera de forma errónea que estas poseen un gran valor sintáctico en el texto. Por esta razón es necesario filtrar este tipo de palabras antes de analizar el texto.
6) Relación de dependencia: el objetivo de esta etapa es la de diseñar un árbol sintáctico. La raíz de dicho árbol será el verbo principal de la oración que relacione el resto de los elementos.
7) Extracción de entidades nombradas (NER): consiste en detectar y etiquetar conceptos relacionados el mundo real. Por ejemplo, que un programa sea capaz de identificar ciertas palabras como nombres de personas o empresas, ciudades y países, etc.
8) Detección de relaciones: consiste en obtener por medio de la estadística relaciones entre las distintas palabras y oraciones del texto.

Como salida tras esta secuencia de pasos se obtiene una serie de relaciones entre las diferentes palabras que componen un texto, siendo estas de gran utilidad tanto para el procesamiento del lenguaje natural como para un posterior análisis de sentimiento. En Figura 2-1 se muestra de forma ilustrativa la secuencia que sigue NLP de forma resumida:

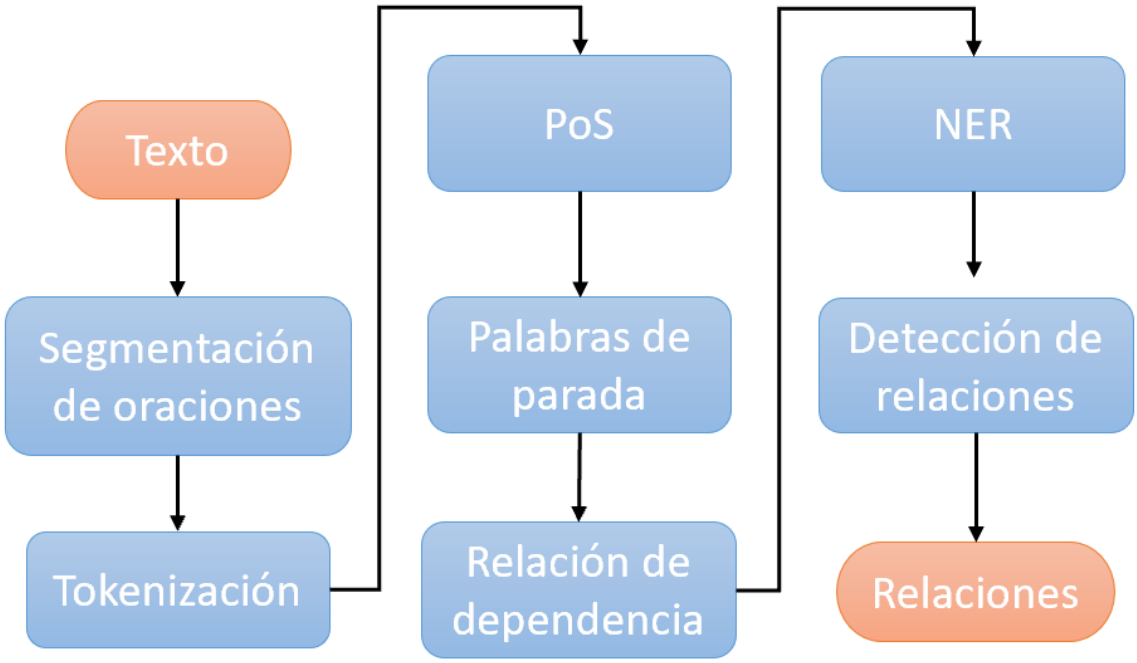

Figura 2-1 Pipeline de procesamiento del lenguaje natural [1]

2.1.3 Dificultades

Dado que el lenguaje natural es un artificio humano, este está sujeto a diferentes ambigüedades y dificultades en distintos los niveles del lenguaje [2]:

- Morfológico: estudia la formación de palabras y su estructura, buscando la unidad mínima de significado, el morfema.
- Léxico: análisis de la categoría gramatical, extracción de raíces, derivaciones, etc.
- Sintáctico: analiza la estructura gramatical de las oraciones haciendo uso de las salidas obtenidas en el nivel anterior.
- Semántico: extracción de significado y reducción de ambigüedades.
- Estructural: establece conexiones entre palabras y oraciones a lo largo del texto.
- Pragmático: trata cómo las oraciones se usan en contextos distintos y de cómo su uso afecta a su significado.

2.1.4 Aplicaciones

Como norma general se puede afirmar que ciertas tareas han sido resueltas gracias a la implementación de modelos NLP. Entre otras, podemos destacar detectores de *Spam* y etiquetación de palabras en una frase (sustantivo, adjetivo, adverbio, …).

Por otro lado, existen otras tareas que aún no han sido completamente resueltas pero que están siguiendo un buen progreso. Ejemplos de estas son el análisis de sentimiento y la traducción de textos.

El procesamiento del lenguaje natural cuenta con infinidad de aplicaciones, permitiendo realizar todo tipo de análisis de texto de forma rápida y automática, permitiendo manejar una inmensa cantidad de datos, y a partir de ellos, obtener todo tipo de aplicaciones: clasificación de textos por categorías, realización de resúmenes automáticos, búsqueda en internet por voz, correctores ortográficos, etc.

2.2 Bag of Words

El modelo de bolsa de palabras (*BoW*) consiste en un algoritmo que nos proporciona la frecuencia con la que aparece una palabra en cada documento. Existen otros modelos, como *tf-idf*, que dan un paso más allá y realizan un cálculo más detallado y realista para determinar el peso de una palabra en un texto (véase el apartado 3.2.3). El objetivo de dicho modelo no es otro que el de extraer características del texto; en otras palabras, busca cuantificar su contenido con la finalidad de ser útil para un algoritmo de aprendizaje. La extracción de características será, por tanto, condición necesaria para realizar el presente TFG, siendo puesta en práctica en la etapa de preprocesamiento del conjunto de datos.

2.3 Word Embeddings

Word Embeddings son representaciones vectoriales de palabras utilizadas para el procesamiento del lenguaje natural (NLP). La representación vectorial de palabras permite realizar una comparación entre estas, permitiendo la clasificación de textos. Además, las palabras con un valor semántico similar tendrán vectores similares. En la Tabla 2-1 se puede ver una simplificación de un modelo de vectores de palabras.

	Reina	Tomate	León	…
Hombre	0.05	0.01	0.2	…
Mujer	0.94	0.01	0.07	…
Animal	0.3	0.03	0.97	…
Comida	0.02	0.98	0.3	…
…	…	…	…	…

Tabla 2-1 Ejemplo simplificado de *Word Embedding*

Al igual que el modelo de bolsa de palabras, este modelo tiene como objetivo presentar el contenido del texto de forma matemática, en este caso vectorial, al modelo de aprendizaje.

2.4 Machine Learning

2.4.1 Definición

Machine Learning (ML) es el campo de estudio que dota a las máquinas de la capacidad de aprender de la experiencia sin haber sido explícitamente programadas para ello. ML se encuentra íntimamente relacionado con la inteligencia artificial (IA).

2.4.2 ¿Por qué usar Machine Learning?

Existen una gran cantidad de problemas que pueden ser resueltos mediante métodos de programación tradicionales. Se programa un determinado algoritmo que cubra todos los escenarios posibles para realizar la tarea en cuestión. Sin embargo, podemos encontrarnos con problemas que no solo sean de gran complejidad, sino que nos hayemos ante un problema en el que no exista un algoritmo conocido para resolverlo o que este solo valga para casos muy específicos.

Por otro lado, *Machine Learning* es considerablemente útil ante problemas de alta complejidad y que requieran de análisis de gran cantidad de información, además de poseer gran capacidad de adaptación en entornos dinámicos, en los que las condiciones bajo las cuales se debe analizar la información varía con el tiempo.

2.4.3 Métodos de aprendizaje

Existen numerosas formas de clasificar sistemas ML, pero el presente texto se limitará a exponer la clasificación más extendida y la más relevante para el desarrollo del TFG:

1) Aprendizaje sin supervisión: la información no es etiquetada de forma previa, sino que el sistema trata de categorizar los datos sin necesidad de un "profesor". Los algoritmos sin supervisión siguen la siguiente clasificación [3]:

    - Algoritmo de agrupación (*clustering)*: al algoritmo se le presentan una serie de datos y sin ningún tipo de identificación previa, es capaz de determinar a qué grupo pertenece cada uno en función de correlaciones que halle entre estos (Figura 2-2). Los algoritmos más conocidos son los siguientes:
        - K-medias
        - Agrupamiento jerárquico (HCA)
        - Algoritmo esperanza-maximización (EM)

    - Visualización y reducción dimensional:
        - Análisis de componentes visuales (PCA)
        - Kernel PCA
        - LLE (*Locally-linear embedding*)
        - Incrustación estocástica de vecinos (t-SNE)

    - Asociación:
        - Asociación a priori
        - Eclat

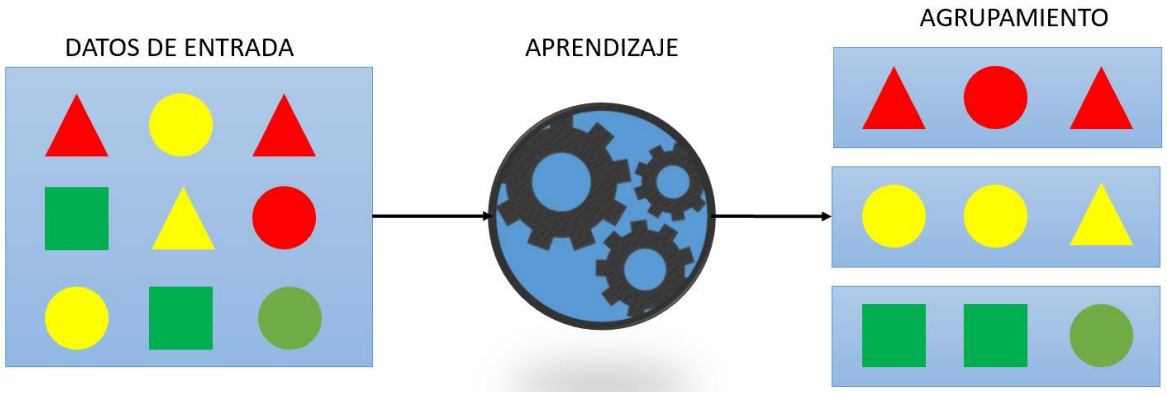

Figura 2-2 Agrupación de datos mediante *clustering* [3]

2) Aprendizaje con supervisión: se obtiene una muestra de la información y se etiqueta según la categoría a la que pertenezca. De este modo el algoritmo aprende a clasificar la información de acuerdo con las asociaciones que sea capaz de establecer entre los datos y sus etiquetas en la muestra que se le otorga. En Figura 2-3 se muestra el funcionamiento de dicho algoritmo.

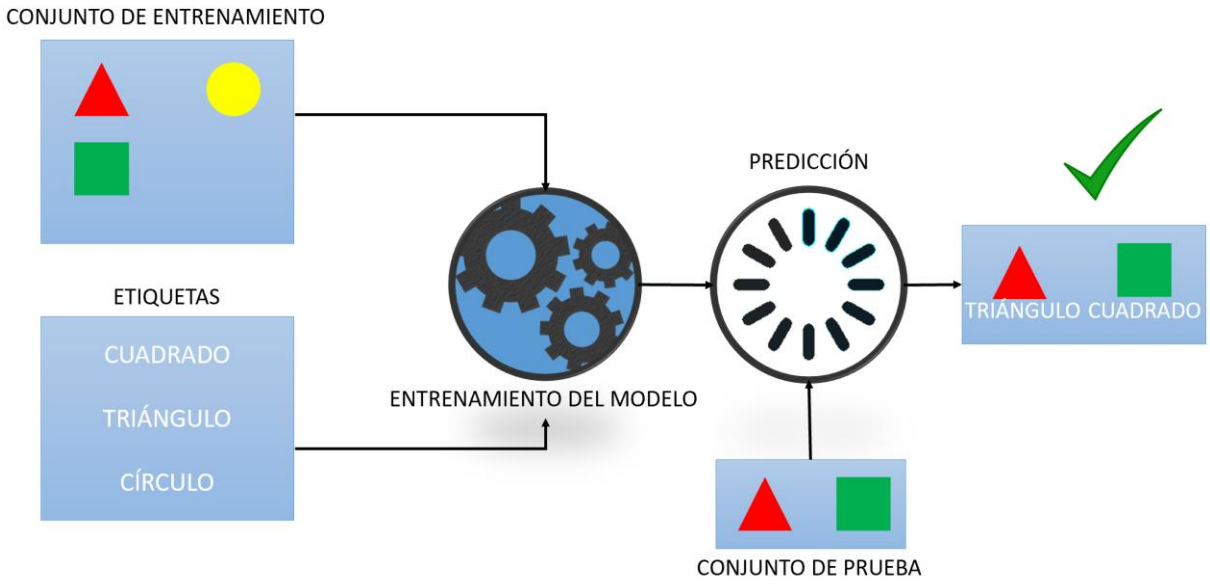

Figura 2-3 Ejemplo de aprendizaje supervisado [3]

Los algoritmos de aprendizaje con supervisión se dividen en aquellos que son de clasificación y los que son de regresión. En los primeros nos encontramos con un etiquetado discreto: "fruta, carne, pescado, …", "positivo, negativo, …", etc. Los de regresión se utilizan para realizar predicciones de variable continua: precios, altura, peso, etc. Entre los algoritmos más utilizados se encuentran [3]:

- *Gauss Naive Bayes*
- *Supported Vector Machines (SVM)*
- *K-nearest neighbours (KNN)*

Estos tres algoritmos serán objeto de estudio en el presente TFG y los resultados obtenidos de cada uno serán comparados con el resto.

3) Aprendizaje semi-supervisado: el sistema trabaja tanto con información etiquetada como con otro tanto sin etiquetar. Un buen ejemplo de esto es *Google Fotos*. Si se guardan todas las fotos de amigos o familiares, el algoritmo será capaz de agrupar fotos por semejanza (aprendizaje sin supervisión) y se pedirá que se etiquete tan solo una foto por categoría (aprendizaje con supervisión), permitiendo deducir la etiqueta del resto de fotos de dicha categoría.

4) Aprendizaje reforzado: un determinado algoritmo de aprendizaje, llamado "agente", puede interactuar con el entorno. En función de las decisiones que tome recibirá penalizaciones o recompensas por parte de este. Su objetivo será encontrar la mejor estrategia para optimizar la obtención de recompensa; como queda reflejado en Figura 2-4:

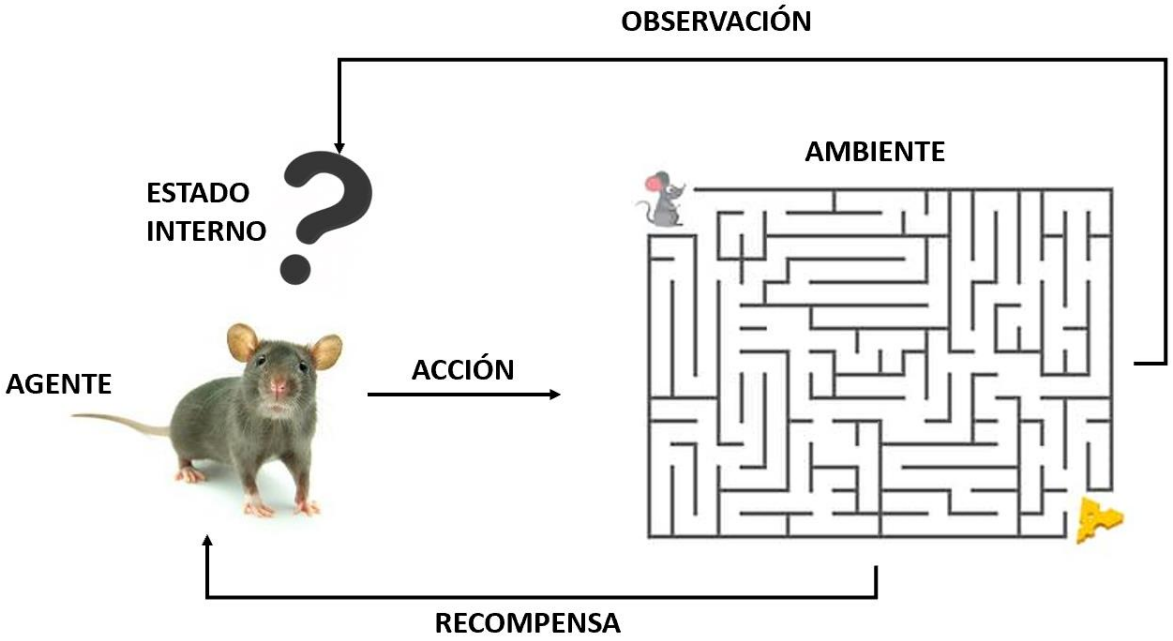

Figura 2-4 Ejemplo de aprendizaje reforzado [4]

2.5 LDA

2.5.1 Definición

LDA (*Latent Dirichlet Allocation*) se trata de un algoritmo ampliamente utilizado para extraer los temas subyacentes en un documento de texto basado en la frecuencia de repetición de las palabras a lo largo del texto. Este algoritmo trata el documento como un conjunto de funciones de distribución de probabilidad, cada una de ellas asociada a una palabra.

Este algoritmo se puede considerar un modelo jerárquico bayesiano [5], es decir un modelo diseñado en distintos niveles que proporciona un resultado aproximado de una serie de parámetros aplicando métodos bayesianos. La formulación matemática de LDA asume que nuestros datos presentan una distribución normal.

Una de las características del LDA es que hace uso de la hipótesis de bolsa de palabras continua, es decir, que la posición que orden que ocupan las palabras no importa en absoluto, únicamente su frecuencia de aparición. De este modo oraciones como “Javier invitó a María” son equivalentes a otras como “María invitó a Javier”.

2.5.2 ¿Por qué usar LDA?

Mientras que algunos algoritmos solo permiten que ciertas palabras pertenezcan a un único grupo (*cluster*), LDA permite que pertenezcan a varios en mayor o menor proporción, permitiendo de este modo un conocimiento más profundo de la temática del documento.

Además, el LDA no se trata exclusivamente de un algoritmo de clasificación, sino también un método de reducción de dimensionalidad [6].

3 DESARROLLO DEL TFG

3.1 Recopilación de datos

3.1.1 Descripción

Como se indicó en el apartado 1.2, este será el punto de partida en el desarrollo del proyecto. El objetivo de esta fase es extraer una cantidad determinada de *tweets*, a saber, lograr al menos unos 3000. Como se indicará más adelante, se hará uso de una serie de herramientas de las que dispone la aplicación de *Twitter* a fin de alcanzar los objetivos propuestos.

3.1.2 Tweepy

Tweepy es una librería de *Python* que permite acceder a la API de *Twitter*. Una API es una interfaz de programación de aplicaciones. En nuestro caso *Twitter* permite acceder a parte de su servicio mediante una API para que pueda crear *softwares* que lo integre. Para hacer uso de esta herramienta es necesario registrarse primero en la aplicación de *Twitter* y crear una nueva aplicación.

Una vez hecho esto se facilitarán una serie de claves para autenticar de forma remota. Por un lado, se tienen las claves "*consumer key"* y *"consumer secret"* que pueden ser entendidas como el nombre de usuario y la contraseña para solicitar acceso remoto a la aplicación. Por otro lado, se cuenta con los *"token"* y los *"token secret"* ¸ que son credenciales para autenticar el acceso a la aplicación. Se hará uso de todas estas claves antes de solicitar la búsqueda de tweets en la aplicación.

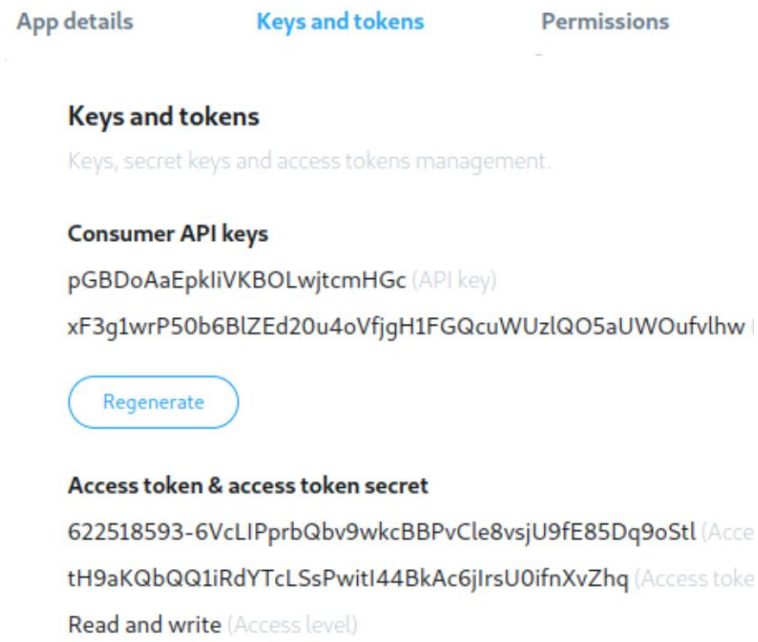

Figura 3-1 Ejemplo de claves y tokens de autenticación en Twitter [7]

Por último, es importante señalar que la API de *Twitter* tiene una serie de limitaciones o restricciones establecidas por los mismos para evitar la saturación de sus servidores en caso de que un gran número de usuarios realicen una cantidad ingente de búsquedas en un momento concreto. De acuerdo con [8], la API solo permite buscar un máximo de 1000 *tweets* cada 15

minutos, y hasta 7 días atrás en el tiempo; lo cual es muy restrictivo para cuentas que no publiquen diariamente gran cantidad de *tweets*. Estas dos limitaciones obligarán a enfocar de una forma determinada una serie de condiciones para realizar la búsqueda sin que *Twitter* deniegue el acceso. Este enfoque se puede comprobar en el código adjuntado en el Anexo I: Código empleado.

3.1.3 Método empleado

A la hora de recopilar datos, se guardaron todos los tweets en formato JSON para su uso posterior.

Los *tweets* siguen un formato de diccionario, es decir, no se trata únicamente de texto sino más bien de una colección de información, valores, asociada a unas claves. Se puede afirmar que los *tweets* son objetos con diferentes atributos, como se puede ver en Tabla 3-1:

```
{
“created_at”: “Thru Apr 06 15:24:15 +0000 2019”,
“id_str”: “850006245121695744”,
“text”: “muy orgulloso de nuestro ejército”,
“user”: {
“id”: 2244994945,
“name”: “Armada Española”,
“screen_name”: “Armada_esp”,
“location”: “Madrid”,
“url”: “http://abcblogs.abc.es/tierra-mar-aire/”,
},
“place”: {
}
“entities”: {
“hastags”: [],
}
...
}}
```

Tabla 3-1 Estructura del *tweet* en formato JSON

Salta a la vista que un único *tweet* aporta información que va mucho más allá del texto escrito en un comentario: fecha de creación, identificación del usuario, localización, etc. A partir de aquí se creará la instancia *“tweet”* con una serie de atributos o métodos para obtener únicamente la información relevante del archivo *JSON* y seguir trabajando con posterioridad con esta información en el resto de las etapas del proyecto de una forma mucho más cómoda y eficiente.

Tweet	**Sentimiento**	***Tags***
Se sabe horarios de visita? 💪 ES	NEU	ARMADA

ojala me llevaran de excursion a algun sitio del ejercito E A E A	POS	ARMADA
mira loca con mis hijosnotemetas	NEG	ARMADA

Tabla 3-2 Ejemplo de un objeto perteneciente a la clase *tweet*

Como se puede apreciar en Tabla 3-2, el contenido del *tweet* no sigue estructura alguna. Se pueden encontrar palabras y números, faltas de ortografía, *emoji's*, vulgarismos, etc. Por este motivo será imprescindible realizar un preprocesamiento al conjunto de datos, a fin de normalizar su contenido para el posterior análisis.

3.2 Preprocesamiento

3.2.1 Descripción

Esta etapa consiste en limpiar el texto de aquellos elementos que dificulten o ambigüen el posterior análisis al realizar el *Machine Learning.* Se trata de preparar los datos para el análisis. De este modo se eliminará todo aquello que no aporte significado a los comentarios escritos por los usuarios.

3.2.2 Normalización del texto

Normalizar el texto consite en volverlo uniforme a ojos del algoritmo. Eliminando ambigüedades y elementos indeseados como los siguientes:

- *Url* del texto
- Saltos de línea
- Mayúsculas
- Nombres de usuarios
- Menciones
- Números
- Tildes y signos de puntuación
- Palabras de parada

Todos estos elementos son conocidos comúnmente como "ruido", ya que su aparición puede llevar a confusiones al algoritmo de clasificación del texto. Palabras repetidas con frecuencia, como por ejemplo "de" o "y" pueden hacerle creer que son de gran relevancia y que aportan significado propio al texto. El caso de las mayúsculas y las tildes es similar, ya que estas solo tienen sentido dentro de la ortografía del lenguaje y no dentro de la semántica (salvo la tilde diacrítica). No interesa que el programa considere que "Hacer" y "hacer" u "organización" y "organización" como palabras distintas, sino como una sola.

Por otro lado, se considera necesario sustituir expresiones ampliamente extendidas en las redes sociales, que no están recogidas en la *RAE*, por otras que sí lo están. Ejemplos de argot en twitter son los siguientes: *"LOL", "WTF", "tqm", "k wapo", "x fvor"*, y otros muchos.

Por último y no menos importante, se debe obtener la raíz o lexema de cada conjunto de palabras. Esto viene motivado por la necesidad de normalizar las distintas derivaciones léxicas de cada palabra, dicho de otra forma, agrupar familias léxicas en su raíz. De este modo el algoritmo considerará las palabras "hacer", "haciendo", "hice", etc. como una sola palabra representada por una palabra que las represente: "hacer".

3.2.3 Extracción de características

Se utilizará la métrica *Term frequency – inverse document frequency* (*Tf*-idf). Se trata de una medida que expresa el peso de una palabra en el documento. La frecuencia del término (*tf*) es básicamente la salida de un modelo de bolsa de palabras (*BoW*), haciendo referencia al número de veces que aparece en un documento.

El segundo término (idf) consiste en un valor que mide la frecuencia con que aparece la palabra en el conjunto total de documentos, expresado como el logaritmo de la inversa de su frecuencia de aparición. El resultado final (*tf-idf*) viene expresado como el producto de ambos parámetros:

$$w_{x,y} = tf_{x,y} \cdot \log\left(\frac{N}{dfx}\right)$$

Ecuación 3-1 Simplificación del cálculo de tdf-idf [9]

Siendo *N* el número total de documentos, dfx el número de documentos que contienen cierta palabra ,$tf_{x,y}$ la frecuencia de aparición de x en y, y $w_{x,y}$ el resultado obtenido tras el producto de los valores *tf-idf* .Habiendo normalizado el texto y, aplicando la Ecuación 3-1, se ha generado un *corpus* que servirá como entrada de los distintos modelos para un posterior análisis de la polaridad de cada uno de los *tweets.* En la Figura 3-2 quedan reflejadas cada una de las distintas etapas en las que se realizó el preprocesado del texto:

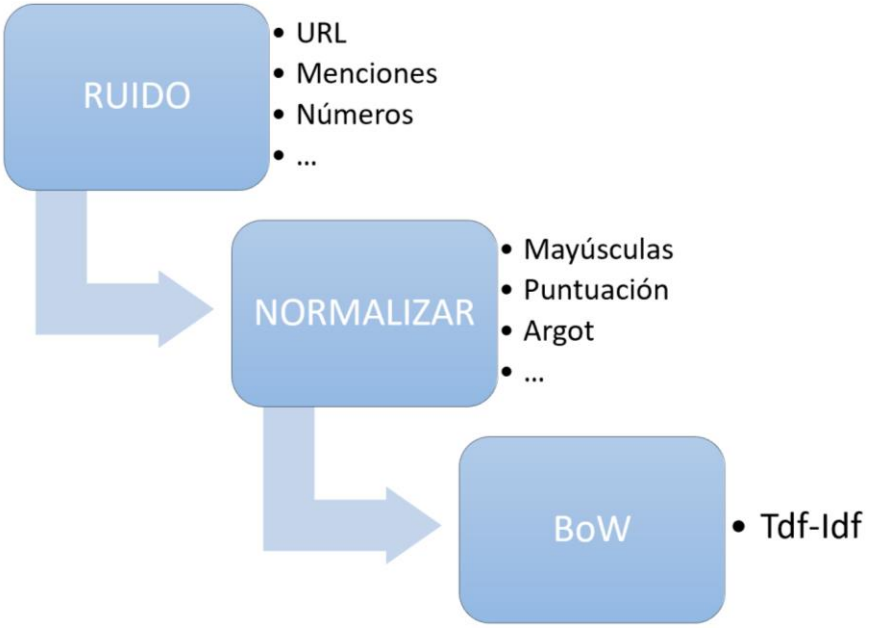

Figura 3-2 Representación esquemática de la fase de preprocesamiento (propia)

3.3 Machine Learning

3.3.1 Introducción

Como ya se adelantó con anterioridad, el proyecto se dividirá en dos fases. Se realizará un análisis sin supervisión, seguido de uno con supervisión. En cada una de las fases se hará uso de una serie de algoritmos y herramientas que serán descritas más adelante.

3.3.2 Aprendizaje sin supervisión

Para realizar el análisis se hizo uso de las herramientas *Polyglot* y *Vader*. El motivo para escoger dos librerías y no una es el siguiente:

- Poca flexibilidad de *Polyglot* para analizar texto ilegible e incapacidad para analizar caracteres con formato *emoji.*
- *Vader* contiene un diccionario de *emoji's,* pero cuenta con un lexicón únicamente en inglés.
- Las librerías de traducción de texto (por ejemplo, *TextBlob*) dependen de la *API* de *Google Translate*, que en versión gratuita limita la traducción a mil palabras por día.

3.3.2.1 Vader

Vader (*Valence Aware Dictionary for Sentiment Reasoning*): se trata de un recurso utilizado para el análisis de sentimiento para valorar tanto la polaridad del lenguaje (positivo, negativo, neutro), como la intensidad de este. Cuando un algoritmo *tokeniza* el texto, a cada uno de los *tokens* ("*great*", "*LOL*", "*:)*", …) le asocia valores dentro del intervalo [-4,4]. Esta asociación la realiza a través de un lexicón (Figura 3-3), un archivo de texto en la que cada fila contiene una palabra seguida de dicho valor y seguido de un vector asociado a este. Existe un archivo alternativo que es el que valora la polaridad de los *emoji's.*

```
abandoned   -2.0    1.09545 [-1, -1, -3, -2, -1, -4, -1, -3, -3, -1]
abandoner   -1.9    0.83066 [-1, -1, -3, -2, -1, -3, -1, -2, -3, -2]
abandoners  -1.9    0.83066 [-2, -3, -2, -3, -2, -1, -2, -2, 0, -2]
abandoning  -1.6    0.8 [-3, -2, -3, -2, -1, -1, -1, -1, -1, -1]
abandonment -2.4    1.0198  [-4, -2, -1, -4, -2, -1, -2, -3, -3, -2]
abandonments    -1.7    0.45826 [-2, -1, -2, -2, -1, -2, -1, -2, -2, -2]
abandons    -1.3    0.9 [-2, -1, -1, -2, -1, -2, -1, -2, 1, -2]
```

Figura 3-3 Lexicón de *VADER* [10]

A la polaridad de las palabras se le añade o se le sustrae cierto valor si se encuentra junto a un incrementador de sentimiento (*booster*). Además, la polaridad se puede invertir si se encuentra junto a un negador. Los incrementadores y negadores se encuentran enumerados en el archivo *vaderSentiment.py*, mostrados en la Figura 3-4:

```
NEGATE = \
    ["aint", "arent", "cannot", "cant", "couldnt", "darent", "didnt", "doesnt",
     "ain't", "aren't", "can't", "couldn't", "daren't", "didn't", "doesn't",
     "dont", "hadnt", "hasnt", "havent", "isnt", "mightnt", "mustnt", "neither",
     "don't", "hadn't", "hasn't", "haven't", "isn't", "mightn't", "mustn't",
     "needn't", "needn't", "never", "none", "nope", "nor", "not", "nothing", "nowhere",
     "oughtnt", "shant", "shouldnt", "uhuh", "wasnt", "werent",
     "oughtn't", "shan't", "shouldn't", "uh-uh", "wasn't", "weren't",
     "without", "wont", "wouldnt", "won't", "wouldn't", "rarely", "seldom", "despite"]
BOOSTER_DICT = \
    {"absolutely": B_INCR, "amazingly": B_INCR, "awfully": B_INCR, "completely": B_INCR, "considerably": B_INCR,
     "decidedly": B_INCR, "deeply": B_INCR, "effing": B_INCR, "enormously": B_INCR,
     "entirely": B_INCR, "especially": B_INCR, "exceptionally": B_INCR, "extremely": B_INCR,
     "fabulously": B_INCR, "flipping": B_INCR, "flippin": B_INCR,
```

Figura 3-4 Listado de incrementadores y negadores de sentimiento [10]

El resultado final del análisis de sentimiento de la oración será resultado de la polaridad de cada una de las palabras que la conforman. Por último, este resultado se normaliza para dar valores en el intervalo [-1, 1].

La métrica que se empleará para el análisis será la siguiente [11]:

1. Positiva para valores mayores de 0.05.
2. Neutral para valores entre -0.05 y 0.05
3. Negativa para valores menores que -0.05.

3.3.2.2 Polyglot

Al igual que *Vader*¸ *Polyglot* hace uso de un lexicón, que en este caso consta de una serie de archivos de texto para cada uno de los idiomas con los que trabaja (hasta 136, entre ellos el castellano) [12]. Además de valorar el sentimiento del texto, puede analizar morfológicamente cada una de las palabras (sustantivo, adjetivo, etc.).

3.3.2.3 Descripción del algoritmo

El método empleado ha sido utilizar *Polyglot* como algoritmo de clasificación por defecto. Al texto se le eliminaron todos los *emoji's* (haciendo uso del lexicón de *Vader*) para minimizar el número de errores en el análisis. En caso de error por ilegibilidad del texto, se hizo uso de *TextBlob* para traducir al inglés. En este caso el límite de traducción de palabras no afectó, dado el reducido tamaño de la muestra que no pudo ser analizada en primer lugar por *Polyglot.* Tras ser traducido se realizó el análisis con *Vader.* La precisión de este algoritmo será explicada en detalle en el apartado 4. A continuación, se muestra en la Figura 3-5 el diagrama de flujo del algoritmo:

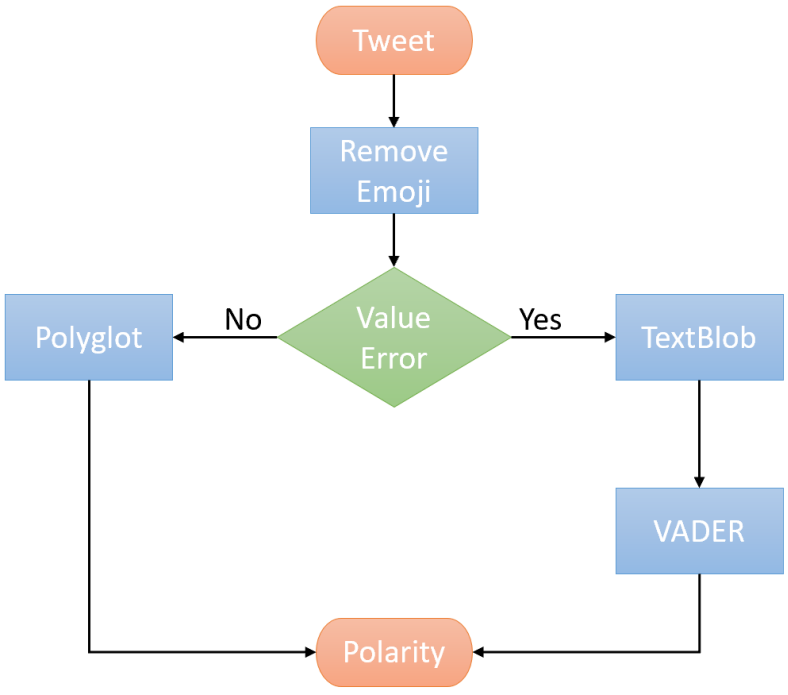

Figura 3-5 Diagrama de flujo del algoritmo sin supervisión (elaboración propia)

3.3.3 Aprendizaje con supervisión

Antes de analizar los *tweets* es necesario primero etiquetar cada uno de ellos según tres clases: positivo ("*POS*"), neutral ("*NEU*"), y negativo ("*NEG*"). La etiquetación es imprescindible para el aprendizaje del algoritmo, de modo que sea capaz posteriormente de realizar predicciones. Los datos serán divididos en dos submuestras: conjunto de entrenamiento y conjunto de prueba, correspondiéndoles a cada uno de ellos el 80% y el 20% de los datos respectivamente. La asignación de un *tweet* a cada una de estas dos submuestras será aleatoria.

3.3.3.1 Cross-fold validation

Según el periódico *ElPaís* [13], en 2014 la empresa *Amazon* diseñó un algoritmo de IA para contratar nuevos empleados de acuerdo con su currículum. Para ello, el algoritmo fue entrenado con los datos de los solicitantes de los últimos 10 años. Al año siguiente la compañía se dio cuenta de que el algoritmo discriminaba a las mujeres candidatas y las descartaba de forma sistemática. La causa de este error fue proporcionarle a la máquina unos datos de entrenamiento notablemente sesgados, haciéndola establecer erróneamente una relación entre la cantidad de mujeres contratadas y su capacidad para ejercer su profesión.

Aquí entra en juego el término validación cruzada, una técnica empleada para evitar sobreajuste (*overfitting*) o subajuste (*underfitting*). El sobreajuste consiste en sobreentrenar a un algoritmo de aprendizaje con unos datos para los que se conoce el resultado deseado. De este modo el algoritmo se vuelve muy preciso para ese conjunto de datos en particular pero incapaz de generalizar el aprendizaje, convirtiéndolo en un modelo poco fiable. Por otro lado, el subajuste se produce cuando no existe suficiente información en el conjunto de entrenamiento, o cuando cuyo contenido sea poco representativo, haciendo que el modelo ignore ciertos parámetros relevantes para clasificar los datos. En la Figura 3-6 se presenta de forma intuitiva cada uno de los casos citados anteriormente:

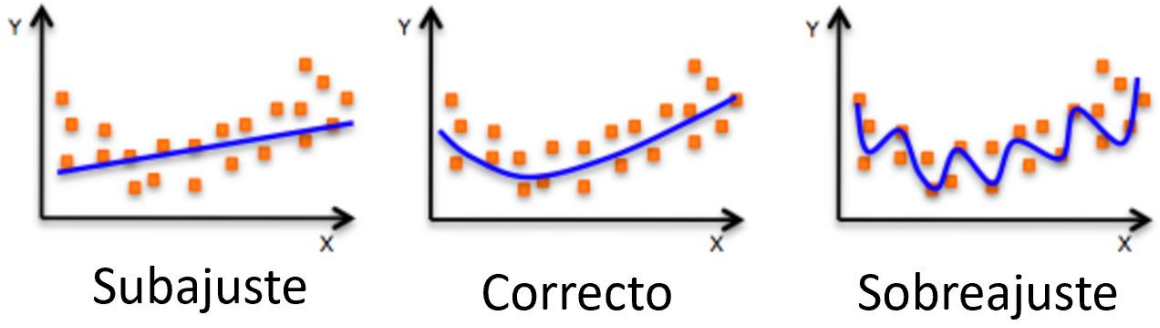

Figura 3-6 Ejemplos de *subajuste* y *sobreajuste* de un conjunto de datos [14]

La validación cruzada consiste en realizar *k* particiones del conjunto de entrenamiento y realizar el aprendizaje con *k-1*, guardando una de ellas para realizar la prueba y medir la precisión del algoritmo en cada uno de los *k-1* aprendizajes realizados. En la Figura 3-7 se presenta esquemáticamente el método de validación cruzada:

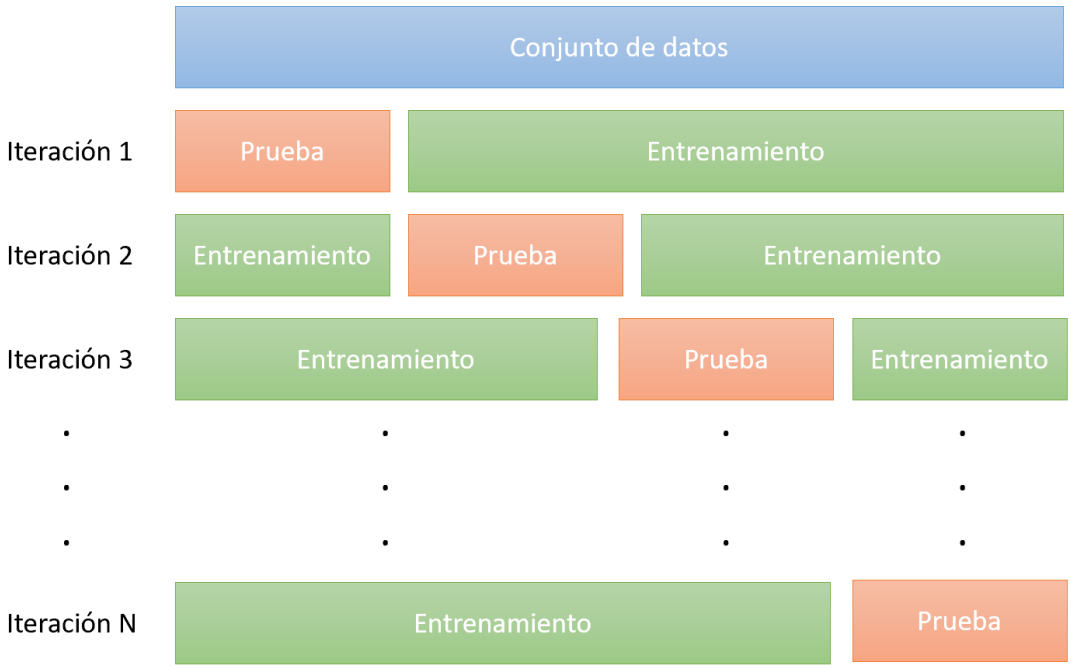

Figura 3-7 Descripción ilustrativa de validación cruzada [15]

Dicho método será empleado en cada uno de los diferentes modelos para descartar la posibilidad de sobreajuste.

3.3.3.2 Algoritmos empleados

Para el análisis se comprobará la precisión de tres algoritmos de *Machine Learning: Gaussian Naive Bayes*, *Support Vector Machine (SVM)*¸ y *K-nearest neighbors (KNN)*. Además, se diseñará un meta-clasificador de algoritmos que consiste en la combinación lineal de las matrices de predicción de los anteriores, usando como coeficientes la precisión de cada uno de ellos. A continuación, se procede a explicar brevemente el funcionamiento de cada uno de ellos.

En primer lugar, *Gaussian Naive Bayes* [16] se trata de un algoritmo de aprendizaje basado en aplicar el teorema de Bayes, asumiendo que cada una de las variables aleatorias son linealmente independientes. El teorema de Bayes enuncia lo siguiente:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \cdot P(x_1, \dots, x_n \mid y)}{P(x_1, \dots, x_n)}$$

Ecuación 3-2 Teorema de Bayes

Además, se considera que la probabilidad de que una variable aleatoria pertenezca a una clase *y* presenta una distribución gaussiana:

$$P(x_i|y) = \int_{-\infty}^{x_i} \frac{1}{\sqrt{2\pi{\sigma_y}^2}} \exp\left(-\frac{(\mathrm{t}-\mu_\mathrm{i})}{2\sigma_\mathrm{y}^2}\right) dt$$

Ecuación 3-3 Probabilidad de pertenecer a una clase *y*

Siendo ${\sigma_y}^2$ y μ_i la varianza y la media de la muestra respectivamente. Aplicando la Ecuación 3-2 y Ecuación 3-3, el modelo clasificará una variable como perteneciente a la clase más probable del conjunto.

En segundo lugar, *SVM* [17] consiste en un algoritmo de clasificación que representa elementos de un flujo de datos como vectores pertenecientes a un espacio n-dimensional. *SVM* calcula un hiperplano de dimensión n-1 que divide los distintos vectores a un lado u otro del hiperplano, actuando como frontera. De este modo obtenemos una serie de conjuntos disjuntos de vectores, asociando cada uno de ellos a una clase.

Salta a la vista que un conjunto de vectores puede ser dividido por hiperplanos diferentes, no existe una solución única. Para determinar el hiperplano óptimo será necesario calcular la distancia de cada vector al hiperplano. La distancia mínima de una clase al hiperplano se denomina margen; de este modo podemos definir el hiperplano óptimo como aquel que maximiza el margen, separando lo máximo posible cada una de las clases:

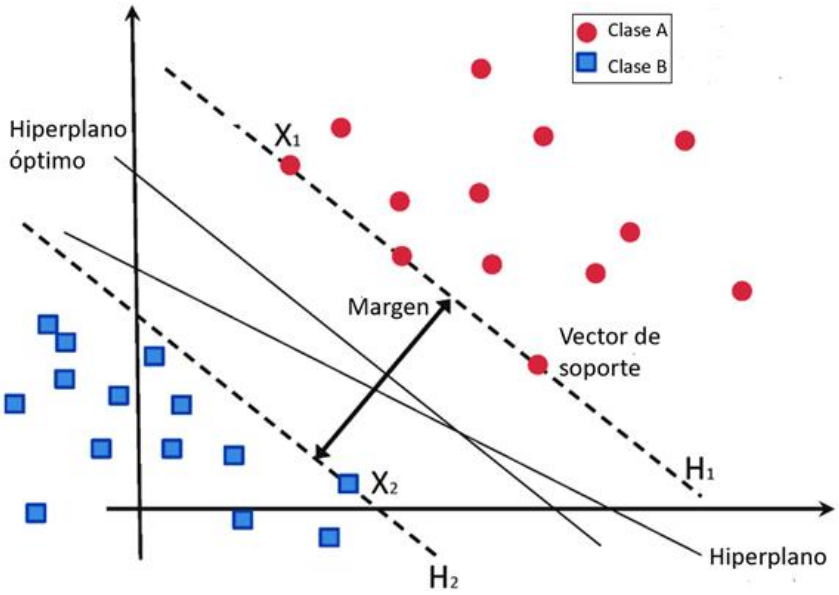

Figura 3-8 Representación del algoritmo SVM en un espacio 2-dimensional [17]

Mientras que en la Figura 3-8 se tiene un problema con solución lineal, se puede dar el caso en el que se tenga un flujo de datos lo suficientemente complejo como para poder ser dividido de forma lineal. Un ejemplo sería un conjunto de vectores distribuidos radialmente por el espacio. Como solución a este problema, se debe hacer uso de clasificadores *SVM* no lineales, transformando el conjunto de vectores mediante una función *kernel*. Dicha función opera sobre los vectores transformándolos en otros vectores pertenecientes a un espacio de dimensión mayor en el que sean linealmente separables. Una vez hecho esto, se puede calcular el hiperplano óptimo y mediante la función *kernel* inversa obtener una solución no lineal en el espacio de partida.

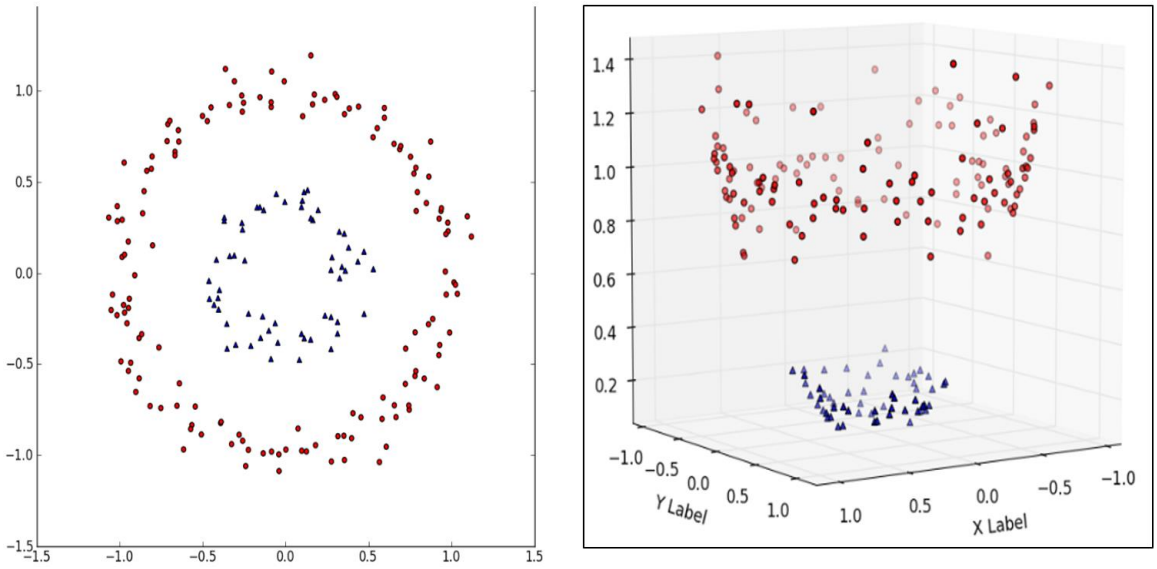

Figura 3-9 Transformación *kernel* de un espacio 2-dimensional [18]

En Figura 3-9 se tiene un espacio con solución no lineal (un círculo en el ejemplo). Tras la transformación *kernel*, se obtiene un espacio 3-dimensional cuya solución es lineal: un plano.

Por último, *KNN* [19] es un algoritmo de clasificación por comparación con *k* vecinos más cercanos. En otras palabras, un elemento será etiquetado como clase "*y*" en función de la cantidad de vecinos pertenecientes a dicha clase. Imaginemos el caso *k*=3, y supongamos que dos de los elementos más cercanos pertenecen a "*y*", y un tercero a la clase "*x*". En este caso este elemento será etiquetado perteneciente a la clase "*y*", puesto que la mayoría de sus *k*-vecinos más cercanos lo son. En términos generales, se puede afirmar que un valor alto de *k* reduce el ruido de los datos y localiza anomalías dentro de estos, pero aumenta el error de decisión en las inmediaciones de la frontera de cada clase; y viceversa [20]. En la Figura 3-10 se muestra una representación gráfica del algoritmo *KNN* para distintos valores de *k*.

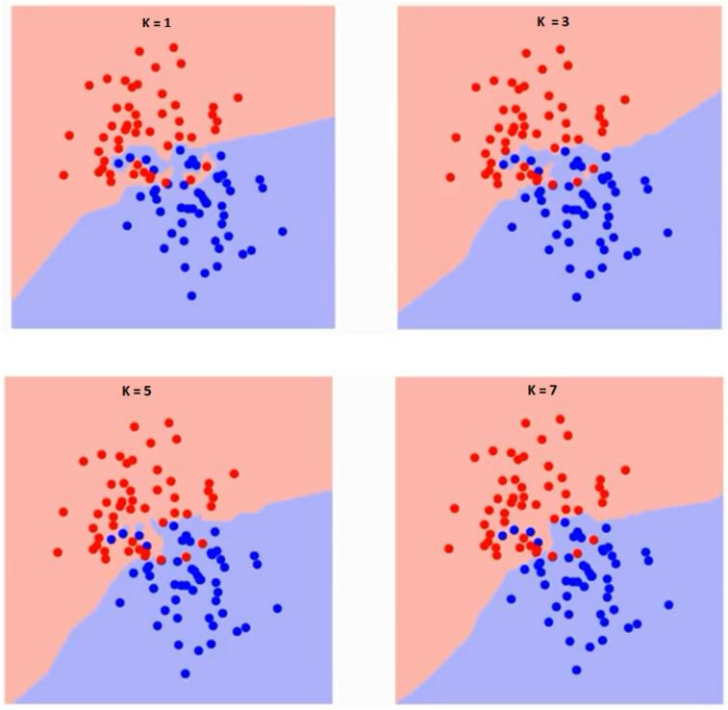

Figura 3-10 Representación gráfica del algoritmo *KNN* en función del parámetro *k* [21]

A la hora de seleccionar el parámetro, *k*, se hará uso de la función *GridSearch*, que realiza una validación cruzada, sobre tres particiones, para cada uno de los valores que pueden tomar los parámetros más representativos en un intervalo dado, devolviendo aquel que ofrece mayor precisión.

Este algoritmo se dice que es de *lazy learning* puesto que no construye un modelo de aprendizaje propiamente dicho, sino que la clasificación la realiza en la etapa de predicción basándose únicamente en la cercanía respecto a otros datos ya clasificados, sin tener en cuenta relaciones obtenidas en la etapa de aprendizaje.

Para optimizar la precisión de cada uno de los algoritmos, se hace uso de *cross-fold validation*. Esto no es más que probar de forma iterativa, con distintas particiones del *Training set*, la precisión del algoritmo variando cada uno de los parámetros. Como salida obtenemos los valores que nos proporcionan mayor precisión. Este método ha sido utilizado con *SVM* y *KNN*.

Como se puede observar, en el Anexo I: Código empleado, no aparece este método dentro del código, dado que requiere un gran tiempo de procesamiento. Lo que se hizo fue realizar este método una única vez, y tras obtener los parámetros óptimos se sustituyó esa parte del código por el algoritmo con dichos parámetros en su interior:

```
>> Input:

param_grid = {"svm_grid":{'C':[0.1, 1, 10, 100, 1000],
'gamma':[1, 0.1, 0.01, 0.001, 0.0001],'kernel':{'rbf'}},
"knn_grid":{'n_neighbors':[3, 5, 11, 19],
'weights':['uniform','distance'],
'metrics':['euclidean', 'manhattan', 'minkowski']}}

clf1 = GridSearchCV(svm.SVC(), param_grid['svm_grid'], cv=3)
clf1.fit(X_train_counts, y_train)
clf2 = GridSearchCV(KNeighborsClassifier(), param_grid['knn_grid'], cv=3)
clf2.fit(X_train_counts, y_train)
print(clf1.best_estimator, clf2.best_estimator)


>> Output:

SVC(C=10, gamma = 0.1, kernel = 'rbf')
KNeighborsClassifier(leaf_size = 30, metric = 'euclidean',
n_neighbors= 11, weights = 'distance')
```

Tabla 3-3 Código empleado para la optimización de parámetros

Como se puede observar en Tabla 3-3, se debe seleccionar una función *RBF Kernel* para optimizar resultados. Una función *RBF Kernel*, para dos vectores x y x', viene representada de la siguiente forma:

$$K(x, x') = \exp(-\gamma \|x - x'\|^2)$$

Ecuación 3-4 Función RBF *Kernel*

Siendo $\|x - x'\|^2$ la distancia euclídea cuadrática y γ una constante (de acuerdo con Tabla 3-3 será igual a 0,1). A partir del resultado obtenido en la Ecuación 3-4 se puede definir una matriz semidefinida positiva a partir de la cual el algoritmo SVM tomará decisiones de clasificación.

3.3.4 Diseño de un metaclasificador

Tras diseñar tres modelos de clasificación diferentes, surge la pregunta de si es posible crear un modelo robusto a partir de un determinado conjunto de modelos. Para ello se diseña un metaclasificador, un modelo de modelos.

Inicialmente, el metaclasificador que se diseñó partía de una idea sencilla e intuitiva, obtener una matriz de predicción como resultado de la media ponderada de las matrices de los modelos anteriores:

$$X = \frac{1}{\alpha_0 + \alpha_1 + \alpha_2}(\alpha_0 X_0 + \alpha_1 X_1 + \alpha_2 X_2)$$

Ecuación 3-5 Matriz de predicción del metaclasificador inicial

Siendo $\alpha_i \epsilon [0,1]$ el valor F-1 del modelo i, y X_i su matriz de predicción. De este modo se da más peso a modelos más robustos en detrimento de los menos fiables. El resultado obtenido en la Ecuación 3-5 es una matriz con coeficientes reales en el intervalo $[-1,1]$. A continuación, se presenta un histograma que describe *grosso modo* la distribución que presentan los datos a la salida del metaclasificador:

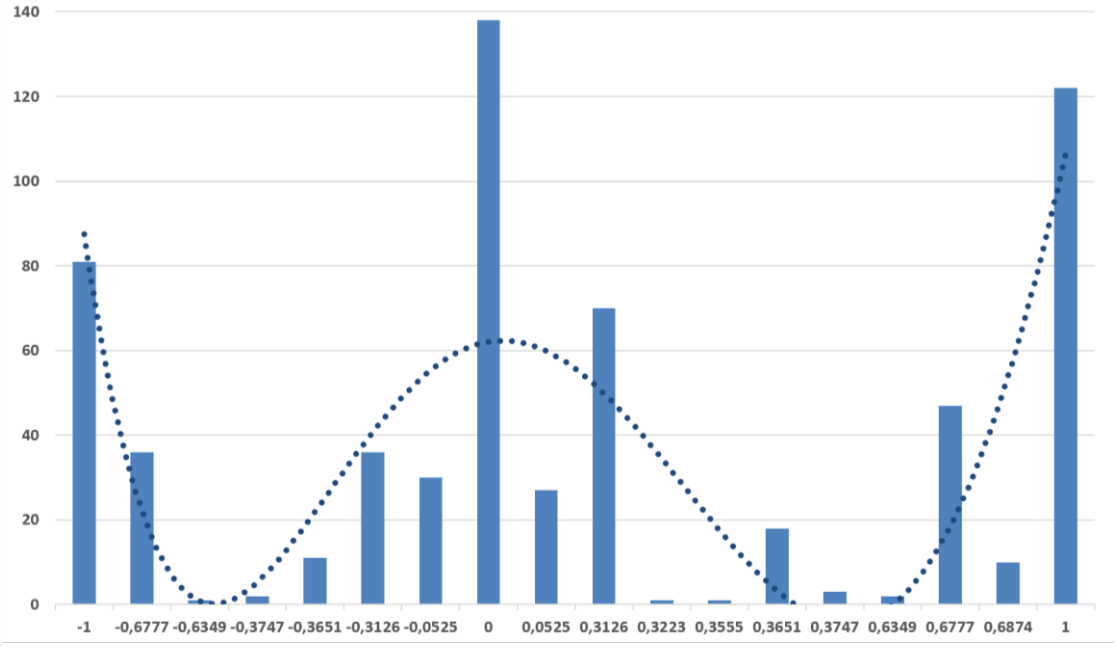

Figura 3-11 Distribución de los valores predecidos por el metaclasificador (propia)

Como se puede observar en la Figura 3-11 Distribución de los valores predecidos por el metaclasificadorFigura 3-11, dicha distribución dista de ser normal. De hecho, los valores se concentran en torno al -1, 0 y 1. Atendiendo a la Ecuación 3-5, esto se puede interpretar como que existe tendencia al consenso entre los tres clasificadores a la hora de predecir, de ahí que la salida sea mayoritariamente uno de estos tres valores. Ahora bien, desde el valor central de la muestra, el cero, hasta los puntos de corte con el eje horizontal, parece comportarse como una distribución normal. Se tomará dicha submuestra como punto de partida para construir una métrica que defina el rango de valores dentro de los cuales un *tweet* será considerado como neutro. Obteniendo la siguiente distribución:

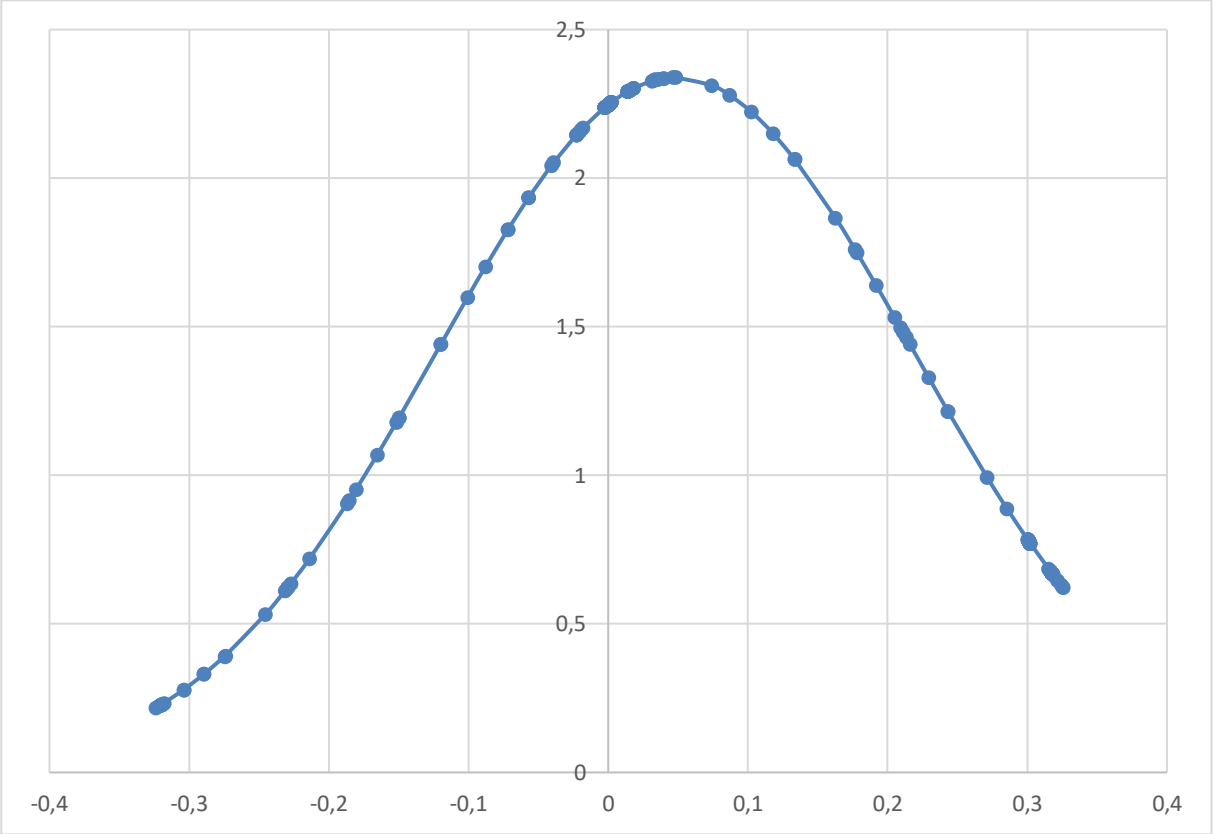

Figura 3-12 Distribución normal de la clase neutral (elaboración propia)

A continuación, se procede a realizar una prueba de normalidad de la distribución mostrada en la Figura 3-12:

	Distribución de la clase neutra	Distribución Normal
Media	0,044	0
Mediana	0	0
Curtosis	-0,24	0
Asimetría	0,07	0

Tabla 3-4 Comparación con la distribución

En Tabla 3-4 se muestran los valores más representativos de una distribución comparados con los de la distribución normal. Como se puede observar, dicha distribución se presenta como cuasi-normal, con un pequeño achatamiento en los valores próximos a la media (curtosis negativa), y una minúscula asimetría hacia la izquierda.

Probada la normalidad de la distribución, se define la clase neutra como el conjunto de valores que encuentra en el intervalo de confianza 2σ (aproximadamente el 95% de los datos). Por medio de la función de distribución acumulada, mostrada en la Figura 3-13, se calculan los límites de dicho intervalo:

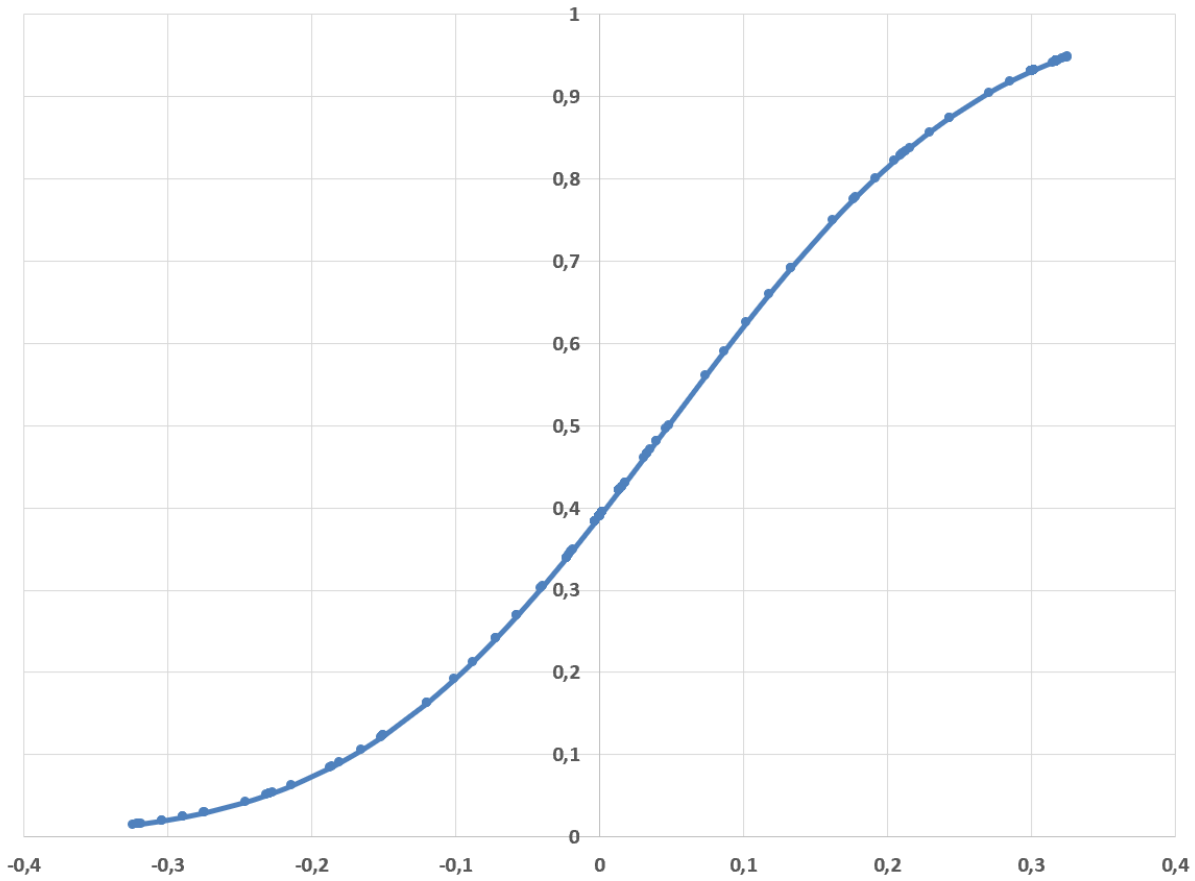

Figura 3-13 Función de distribución acumulada (elaboración propia)

Así pues, queda definida la clase neutra, como el conjunto de valores que se encuentran en el intervalo [-0.32, 0,32]. A continuación y por medio de métodos iterativos, se procede a realizar un ajuste fino en torno a estos valores a fin de optimizar la métrica, buscando aquellos que maximicen el valor F-1. En la Figura 3-14 se muestran los resultados obtenidos:

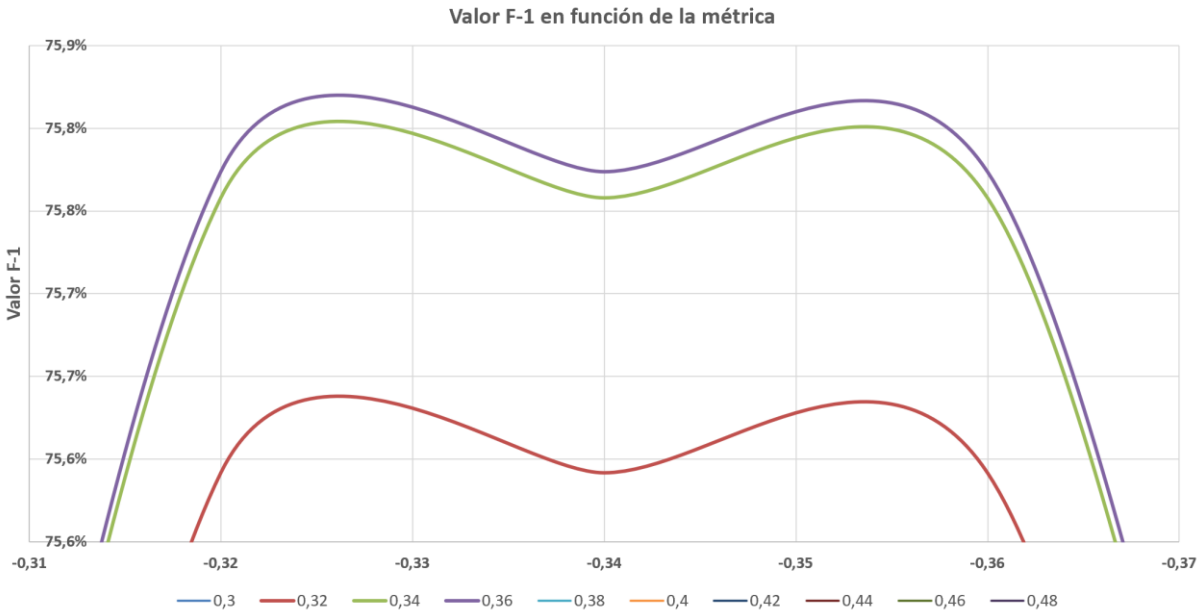

Figura 3-14 Representación gráfica del valor F-1 en función de la métrica (propia)

De acuerdo con Figura 3-14Figura 3-15, el valor F-1 alcanzará su máximo para una métrica tal que la clase neutral se encuentre en el intervalo [-0.32, 0.36]. A partir de estos resultados y los anteriores, se muestra a continuación, en Figura 3-15, la métrica escogida finalmente:

Figura 3-15 Métrica obtenida para el primer metaclasificador (elaboración propia)

Aunque los resultados obtenidos sean más que aceptables, el modelo no supera el valor F-1 obtenido por el algoritmo SVM en solitario (véase el apartado 4.1). Así pues, se ha procedido a diseñar un segundo metaclasificador que sea entrenado con las salidas de los tres algoritmos, de modo que sea capaz de predecir cuál de todas las predicciones es la más idónea en cada caso, y de generar su propio modelo. El algoritmo de clasificación escogido para esta empresa ha sido SVM, en vista a los excelentes resultados obtenidos en el apartado 4.1.

En la Figura 3-16 se muestra un esquema del funcionamiento del metaclasificador:

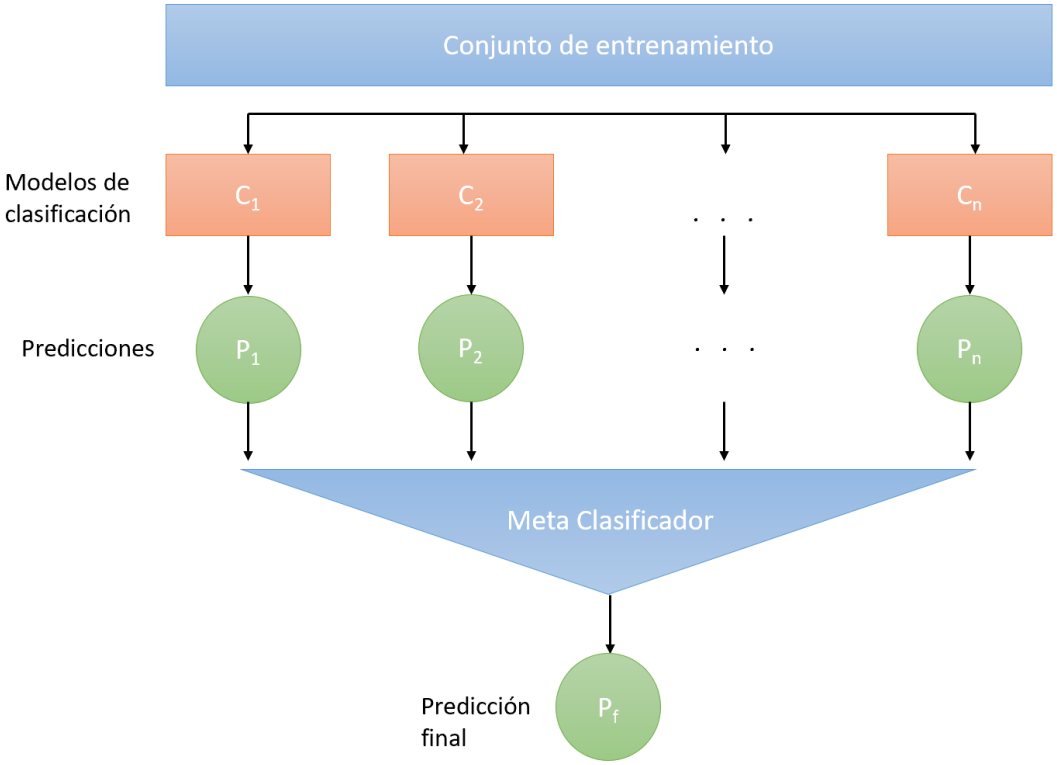

Figura 3-16 Representación esquemática del metaclasificador final (elaboración propia)

3.3.5 Clasificación multiclase y multietiqueta

A la hora de hablar de los distintos algoritmos de clasificación, siempre fueron explicados y ejemplarizados para el caso de etiquetado binomial. Sin embargo, cada uno de los *tweets*, fueron etiquetados de acuerdo con tres categorías. Mientras que algunos algoritmos permiten de forma natural el uso de más de dos instancias, otros son inherentemente binomiales. Haciendo uso de

una serie de estrategias estos últimos podrán ser transformados en clasificadores multiclase. Si bien en la clasificación multiclase las clases se consideran mutuamente excluyentes entre sí, en clasificación multietiqueta no es necesariamente cierto. Dicha clasificación consiste en proporcionar a cada dato con más de una etiqueta. Por ejemplo, podemos etiquetar un *tweet* como positivo, neutro y negativo; todo esto a la vez. Aunque parezca absurdo a la par que contradictorio, la clasificación multietiqueta será el modo de proceder a la hora de construir un metaclasificador (véase el apartado 3.3.4). Cada etiqueta será la salida de uno de los tres clasificadores de aprendizaje supervisado, que servirán como entrada de un modelo que decidirá cuál de estas etiquetas es la adecuada para cada caso.

A continuación, procederé a explicar de manera detallada las dos técnicas de clasificación multi-clase más extendidas: "*One versus One*" y "*One versus Rest*".

One versus Rest (OvR) es una estrategia que consiste en entrenar un clasificador por clase, considerando los elementos de dicha clase como positivos y al resto como negativos. De este modo dividimos un problema multiclase en varios problemas de clasificación binaria.

En cambio, en *One versus One (Ovo)* se construye un clasificador para cada pareja de clases. Esto lleva a un total de $\frac{N(N-1)}{2}$ clasificadores. Siendo *N* el número de clases. Aunque se trate de un método mucho menos sensible a conjuntos de datos desbalanceados,se trata de un método más lento que *OvR* [22], puesto que presenta una complejidad computacional (véase el apartado 3.3.7) mucho mayor.

A la hora de realizar la clasificación multiclase, se hará uso de *OvR* en los diferentes algoritmos.

3.3.6 *Modelado LDA*

Como ya se adelantó en el apartado 2.5, el *LDA* se trata de un algoritmo de clasificación sin supervisión que será de gran utilidad a la hora de determinar qué palabras son más representativas para cada *topic*, además de proporcionar una idea a grandes rasgos de los temas más comentados en *Twitter* en las distintas cuentas de los tres ejércitos. Para ello, se realizará un preprocesado similar al ejecutado en el análisis con supervisión, eliminando además los *emoji's*, ya que su utilidad fuera del análisis de la polaridad de los *tweets* es prácticamente nula y no aportan significado semántico dentro de los *topics* (e.g. ":)" no proporciona información sobre la temática del texto.). Para hacer más eficaz el análisis, únicamente se tomarán en cuenta los sustantivos; para lo cual se hace uso de *Polyglot*. Por último, se eliminan una serie de palabras con una elevada frecuencia de aparición en el texto, pero consideradas como ruido ya que su finalidad es añadir cortesía al lenguaje más que información al texto: "buenos días", "gracias", "adiós", "por favor", etc.

A partir del conjunto de *tweets* se genera un diccionario, sobre el cual se construye a su vez un *corpus*. A continuación, se genera un modelo, obteniendo de este modo cuatro categorías distintas con las palabras más representativas de cada una, como se puede observar en la Tabla 3-5:

```
Topic 1:
Trabajo gente soldados brigada apoyo compañero cuenta historia
seguridad video
Topic 2:
Verdad servicio semana capacidad aniversario vuelo fuerza
```

```
Buque nave pais
Topic 3:
Rey ejercito tropa seguridad fuerzas España trabajo tiempo cosa vida
Topic 4:
Persona vida unidad cuartel bandera salto gobierno tercio mundo
gente
```

Tabla 3-5 Palabras más representativas de cada *topic*

A continuación, en la Figura 3-17 se muestra una representación gráfica de un histograma de las treinta palabras más frecuentes, junto con una distribución de los diferentes *topics.*

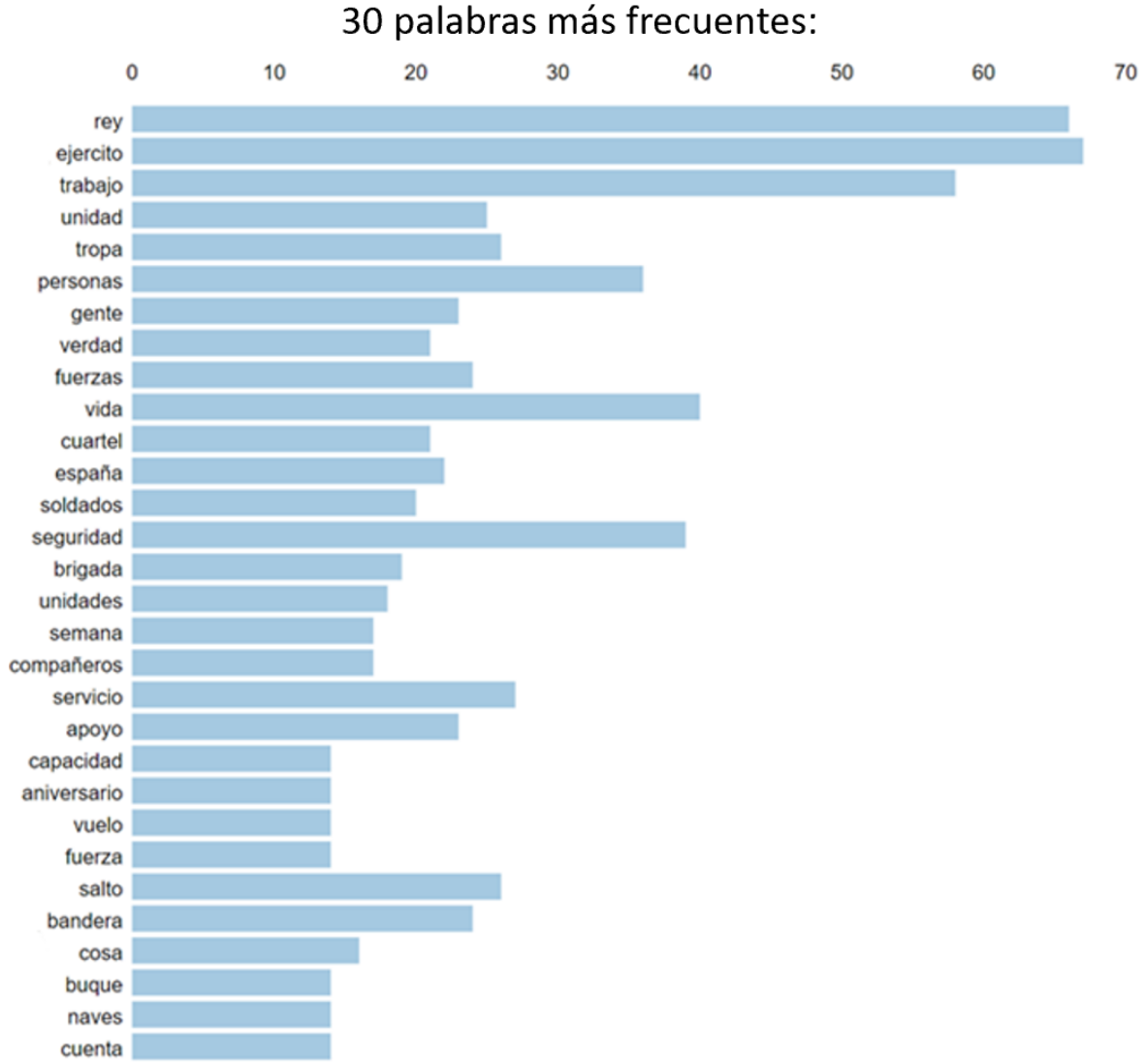

Figura 3-17 Palabras más comunes en el conjunto de *topics* (elaboración propia)

Por último, se muestra en Figura 3-18 una representación gráfica en la que queda reflejada la distancia entre cada uno de los temas de cada grupo de datos:

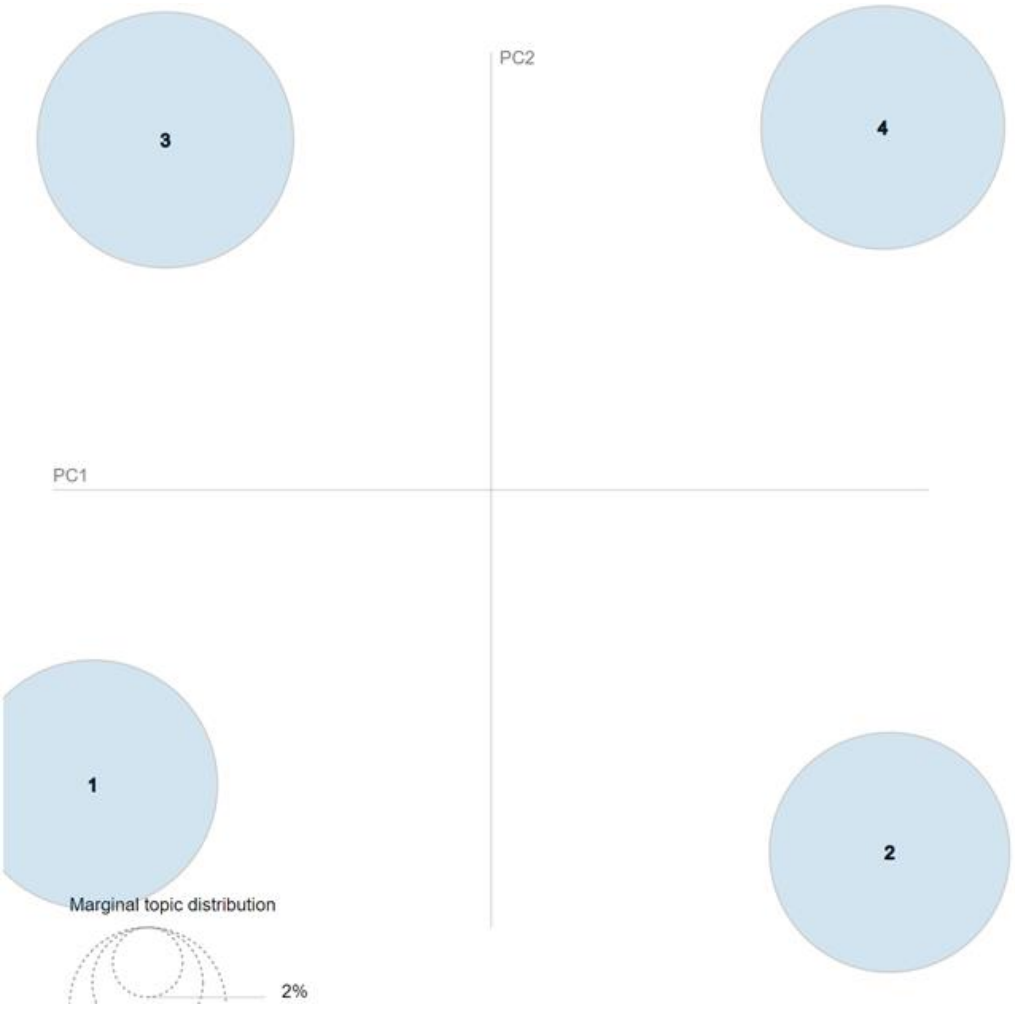

Figura 3-18 Representación de los distintos *topics* en el plano (elaboración propia)

A la hora de dividir el conjunto de datos en diferentes grupos surge la pregunta de en cuántos deben dividirse, o cuál es el número óptimo de categorías para dividir los datos sin que estos pertenezcan a dos de ellas al mismo tiempo, ocasionando redundancias. Así pues, se realizaron agrupamientos de datos con distinto número de *topics*; hasta llegar a cinco, momento en el cual dos de ellos comenzaron a solaparse. Para entender esto en detalle tómese como ejemplo la Figura 3-18. En ella se ven una serie de círculos que representan cada uno de los *topics* del conjunto de datos. El centro de dicho círculo es la media del conjunto, y la distancia entre centros corresponde con la varianza entre categorías. Es decir, cuanto más se encuentren alejados, tanto más diferente es el contenido semántico de estos. Cuanto más solapados se encuentren dos círculos cualesquiera, tanto más similares será el contenido del texto en cuanto a temática; en otras palabras, se produce una redundancia en la clasificación. Por tanto, se dividió el conjunto de datos en cuatro grupos, que es el número máximo de categorías que verifica que todos los conjuntos de datos son disjuntos.

3.3.7 Complejidad computacional

Se puede definir la complejidad computacional como el coste temporal que requiere la ejecución de un algoritmo. Aquellos modelos de computación que requieran de mayores recursos tendrán asociados un alto coste computacional.

Un algoritmo es, a fin de cuentas, una serie de instrucciones que garantizan la solución de un problema en un número finito de pasos. De este modo se pretende calcular cuánto tiempo

requiere al algoritmo resolver el problema a medida que la instancia crece. El tiempo de ejecución del algoritmo queda determinado por las dimensiones de la instancia, además del número de bits requeridos para codificar la información numérica de esta. De ahora en adelante, se hará uso de la notación $O(N, M)$ para referirse al coste computacional de un algoritmo, siendo N el número de instancias y M el número de propiedades de la instancia. En Tabla 3-6 se puede apreciar la complejidad de cada uno de los algoritmos empleados:

	$O(N, M)$
GaussNB	$M \cdot N$
SVM	$M \cdot N^3$
KNN	$M \cdot N$

Tabla 3-6 Complejidad computacional de los algoritmos de aprendizaje

De aquí se deduce que el algoritmo *SVM* tiene con diferencia mayor costo computacional. Por otro lado, *GaussNB* y *KNN* presentan un costo del mismo orden.

A fin de reducir el costo computacional de los distintos algoritmos, se puede recurrir a técnicas de reducción de dimensionalidad. Estos métodos consisten en combinar dos o más variables, o quedarse con las más representativas, sin que esto afecte de manera sustancial a las predicciones del modelo.

Durante el desarrollo del presente TFG, se notó cierta diferencia en el tiempo de ejecución de SVM en comparación con el resto de los algoritmos. Sin embargo, dado que se contó con un *corpus* no muy grande, el tiempo de espera apenas se hizo notar. Por este mismo motivo no se precisó de técnicas de reducción de dimensionalidad u otros métodos similares.

4 RESULTADOS

4.1 Medición de los resultados

4.1.1 *Conjunto de datos importados*

Tras importar un total de 3000 *tweets* se muestra a continuación la frecuencia de aparición de cada clase dentro de la muestra:

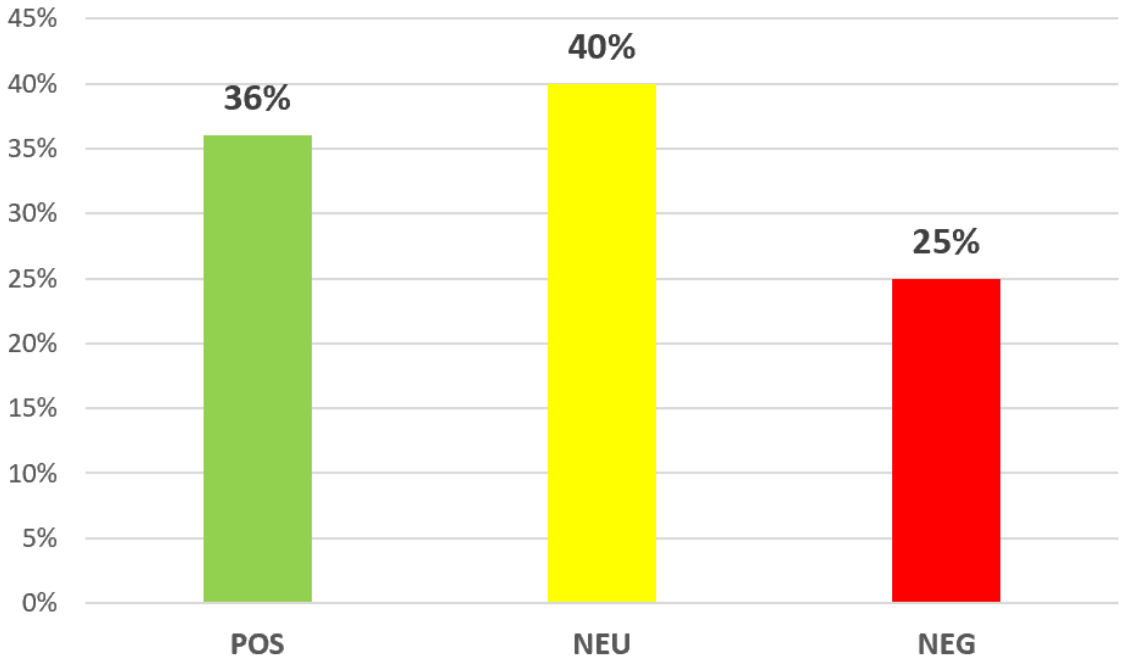

Figura 4-1 Distribución de la muestra atendiendo a su clase (elaboración propia)

Aunque en Figura 4-1 se aprecie un cierto sesgo en la clase de los *tweets* negativos, ciertamente el sesgo afectaría únicamente cuando este sea verdaderamente pronunciado (e.g. con un 2% de los *tweets* negativos). No nos encontramos con un conjunto de datos significativamente desbalanceado, por lo que no requiere validación cruzada. Sin embargo, esta técnica también es útil para probar que el modelo no ha sido sobreentrenado con el conjunto de entrenamiento. En el siguiente epígrafe, se mostrarán los resultados obtenidos para cada una de las particiones entrenadas en la validación cruzada (véase Figura 4-2).

4.1.2 Métodos de evaluación de algoritmos

Los resultados obtenidos de las predicciones obtenidas para cada uno de los algoritmos se pueden comparar mediante los siguientes parámetros:

- Precisión: razón entre el número de elementos clasificados correctamente dentro de una clase y el total de elementos clasificados dentro de la clase.
- Exhaustividad (*Recall*): relación entre los documentos clasificados correctamente dentro de una clase y la suma de todos los elementos de la clase.
- Valor F1 (*F1-Score*): media armónica de los anteriores. Suele emplearse como referencia para comparar el rendimiento entre modelos.

En el caso de que el sistema cuente con más de dos clases, como es el caso, se debe calcular cada una de las métricas anteriores para cada clase y combinarlas entre ellas para obtener una medida global. Existen tres opciones [23]:

- *Macro-averaging:* se calculan las medias de la precisión y exhaustividad de cada clase. Considera que existe el mismo de elementos en cada clase, por lo que sus resultados no son fiables en caso de muestras desbalanceadas.
- *Micro-averaging*: realiza una media ponderada teniendo en cuenta el número de elementos de cada clase. El problema surge cuando existen más de dos clases; la precisión, exhaustividad y el valor F-1 toman los mismos valores.
- *Weighted-averaging*: resuelve el problema de un *corpus* desequilibrado con más de dos clases, realizando una correcta ponderación en base al peso de cada clase.

En vista a la descripción de cada una de estas métricas, se hará uso de *weighted-averaging* dado que es más adecuada para el tipo de datos a analizar.

Además de comparar la precisión de cada modelo por separado, se mostrarán los resultados obtenidos tras realizar *cross-validation* (véase el apartado 3.3.3.1). Para ello, se disponen de diez particiones de la muestra del *training* inicial, realizando el aprendizaje con nueve de estas y siendo sometido a una prueba de precisión con la última de estas. En la Figura 4-2 se pueden apreciar los resultados de los distintos *tests*:

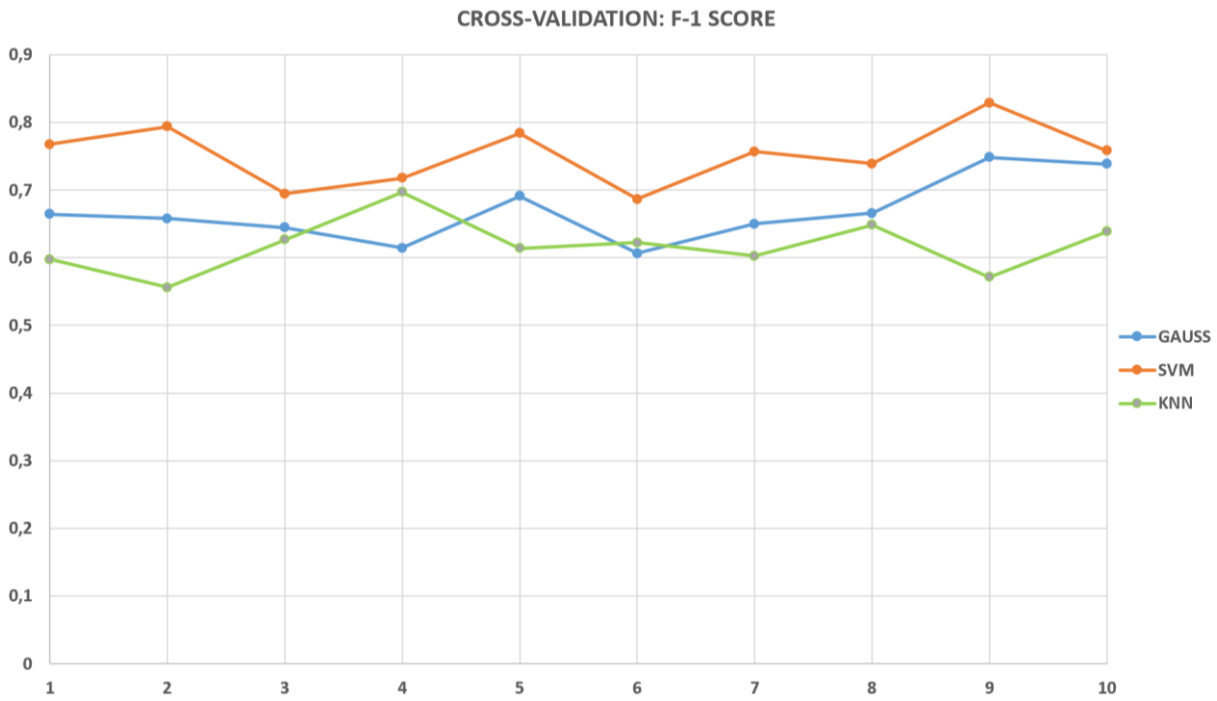

Figura 4-2 Resultados del Valor F-1 tras realizar *cross-validation* (elaboración propia)

Queda probado pues que el valor F-1 de cada uno de los modelos presenta una escasa variabilidad en cada una de las distintas particiones.

Habiendo descartado por completo la posibilidad de sobreajuste se procede a comparar los modelos haciendo uso de las métricas descritas al inicio del epígrafe. En la Tabla 4-1 se muestran los valores de precisión, exhaustividad y valor F-1 de cada modelo dividido por clases:

	POSITIVO			NEUTRAL			NEGATIVO		
	P	R	S	P	R	S	P	R	S
1	0,69	0,72	0,71	0,66	0,53	**0,59**	0,64	0,74	**0,69**
2	0,78	0,68	0,73	0,71	0,8	0,75	0,83	0,79	**0,81**
3	0,91	0,49	**0,63**	0,54	0,93	0,68	0,91	0,57	0,7
4	0,8	0,66	0,72	0,67	0,81	0,73	0,85	0,78	**0,81**
5	0,83	0,88	**0,85**	0,76	0,76	**0,76**	0,84	0,75	0,79
	P: precision R: recall S: F-1 score								
	1: GaussNB 2: SVM 3: KNN 4: Metaclasificador inicial 5: Metaclasificador final								

Tabla 4-1 Comparación de los resultados obtenidos de los modelos con supervisión

Queda reflejada con completa claridad la superioridad del segundo metaclasificador diseñado con respecto al resto de modelos. Además, de aquí se deduce la poca fiabilidad de *GaussNB* y *KNN* respecto al resto de modelos. Este resultado es razonablemente lógico, dado que el primero parte de la presunción de que los elementos del conjunto de datos presentan una distribución normal y cada una de las variables en independiente respecto al resto; y en el segundo, por tratarse de un algoritmo de *lazy learning.*

Por último, también se puede apreciar cierta mejora del segundo metaclasificador con respecto al primero.

A continuación, se presentan los resultados obtenidos con el algoritmo de aprendizaje sin supervisión:

POSITIVO			NEUTRAL			NEGATIVO		
P	R	S	P	R	S	P	R	S
0,53	0,52	**0,53**	0,5	0,34	**0,4**	0,48	0,58	**0,53**
P: precision R: recall S: F-1 score								

Tabla 4-2 Resultados obtenidos con el modelo sin supervisión

En Tabla 4-2 queda probada la abrumadora diferencia entre los resultados obtenidos con el modelo de aprendizaje sin supervisión con respecto a cualquiera de los anteriores.

Por último, a modo de resumen, en Figura 4-3 se muestra el valor F-1 ponderado (aplicando *weighted-averaging*) de cada uno de los modelos en el siguiente diagrama de barras:

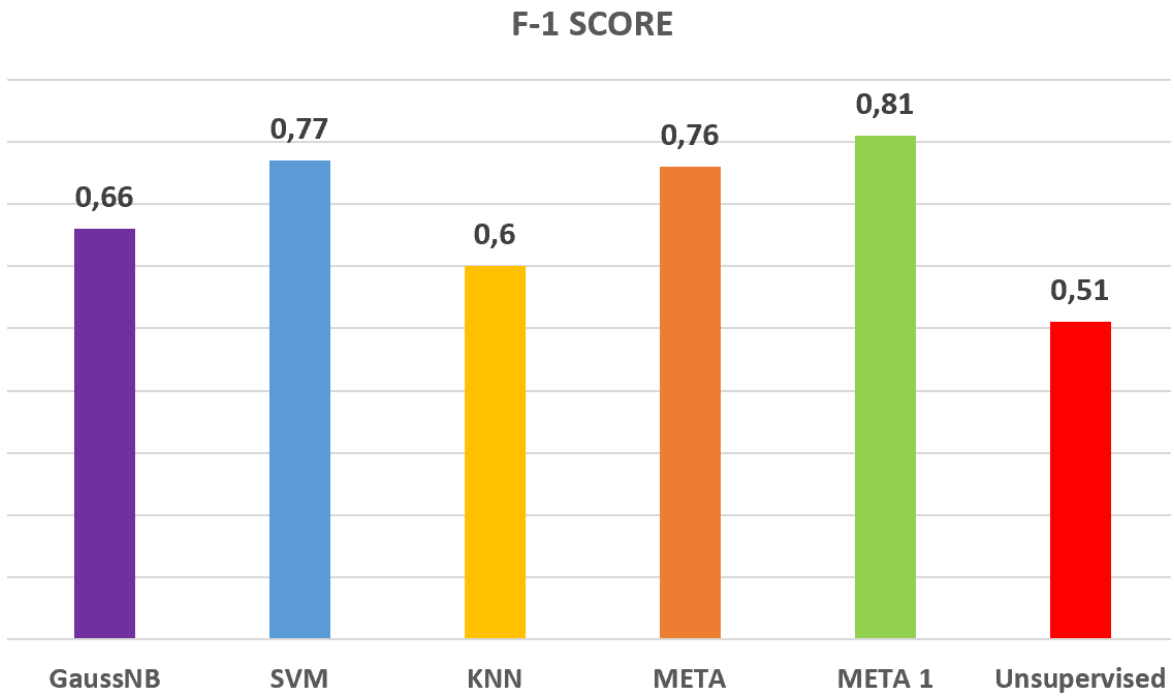

Figura 4-3 Diagrama ilustrativo del valor F-1 medio (elaboración propia)

Una vez concluida la valoración de la precisión de los modelos, cabe preguntarse lo siguiente: ¿se confunden las clases entre sí por igual, o existen tendencias hacia cierta clase a la hora de clasificar incorrectamente un *tweet*? Para abordar dicha cuestión, entra en juego una herramienta que de forma gráfica e intuitiva ayuda a interpretar en qué manera los modelos confunde clases entre sí: la matriz de confusión.

4.1.3 Matriz de confusión

La matriz de confusión es una matriz que tiene como coeficientes falsos positivos, verdaderos positivos, falsos negativos, etc. Dado que se clasificaron los *tweets* de acuerdo con tres categorías, se obtendrá una matriz 3x3. Esta herramienta es de gran ayuda para ver con mayor claridad, y más específicamente, la precisión del modelo por etiquetas y cuáles de estas son más susceptibles de ser confundidas con otras.

Con una etiquetación como la realizada en este TFG (“positivo”, “negativo” y “neutro”), lo más lógico sería que el modelo presente facilidad a la hora de distinguir los *tweets* positivos de los negativos, confundiendo con mayor frecuencia los neutrales con los anteriores. Ciertas expresiones como “viva España” son repetidas una y otra vez en *tweets* etiquetados como positivos. Lo mismo ocurre con los negativos. En cambio, los *tweets* neutrales carecen de expresiones características, volviendo más improbable una predicción acertada de estos. Este fenómeno se puede apreciar a continuación, donde se muestra la matriz de confusión de cada uno de los tres algoritmos empleados:

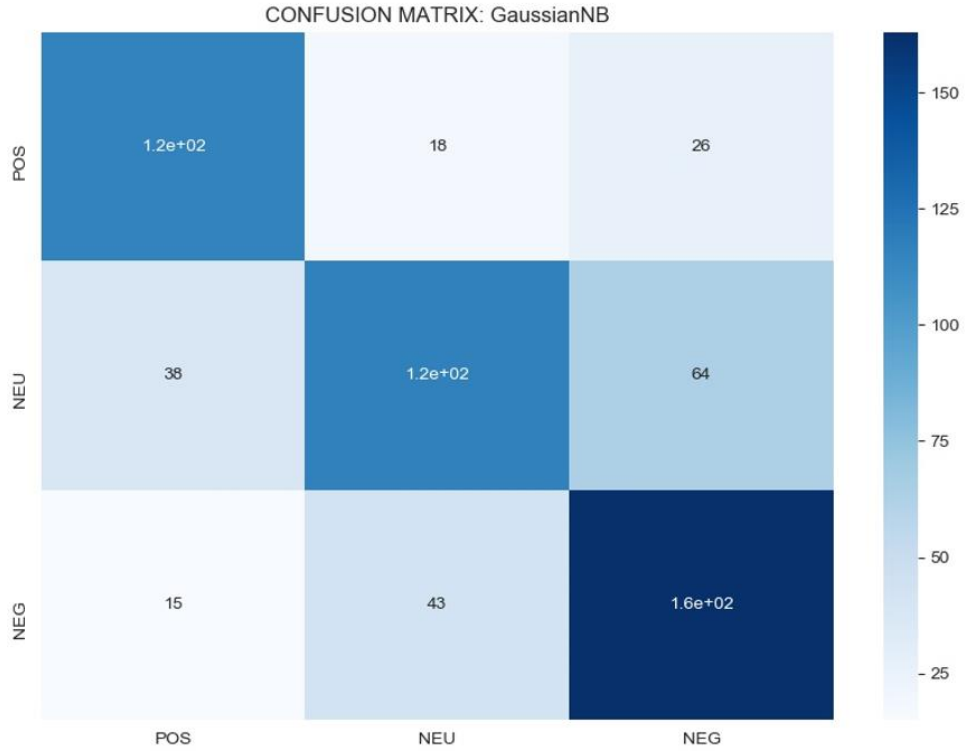

Figura 4-4 Matriz de confusión de *Gaussian Naive Bayes* (elaboración propia)

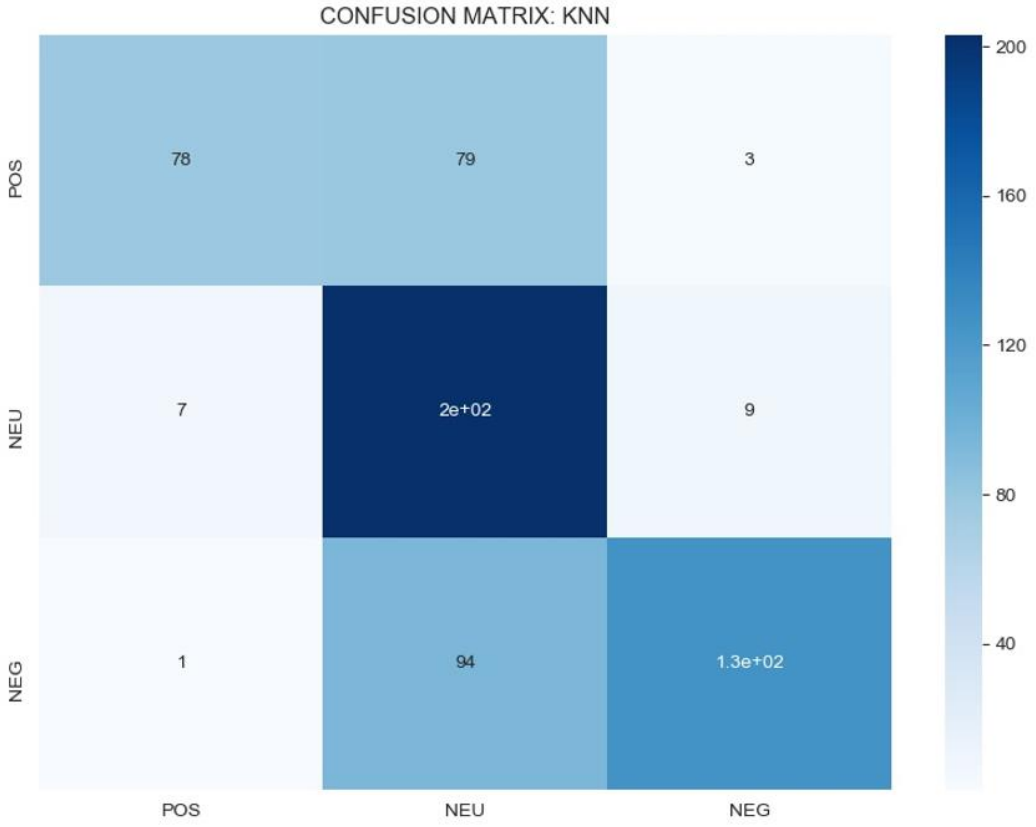

Figura 4-5 Matriz de confusión de *KNN* (elaboración propia)

Los valores del eje vertical corresponden con aquelos que predice el modelo, siendo los del eje horizontal los valores verdaderos del conjunto de datos. Cuanto más preciso sea un modelo, tanto mayor serán los valores de la diagonal principal de la matriz en comparación con aquellos que se encuentren fuera de esta. Como se puede comprobar en Figura 4-5, el modelo KNN

presenta una gran tasa de confusión con la clase neutral con cualquiera de las otras dos. Por otro lado, de acuerdo con Figura 4-4, *Gaussian Naive Bayes* confunde la clase neutral principalmente con la negativa.

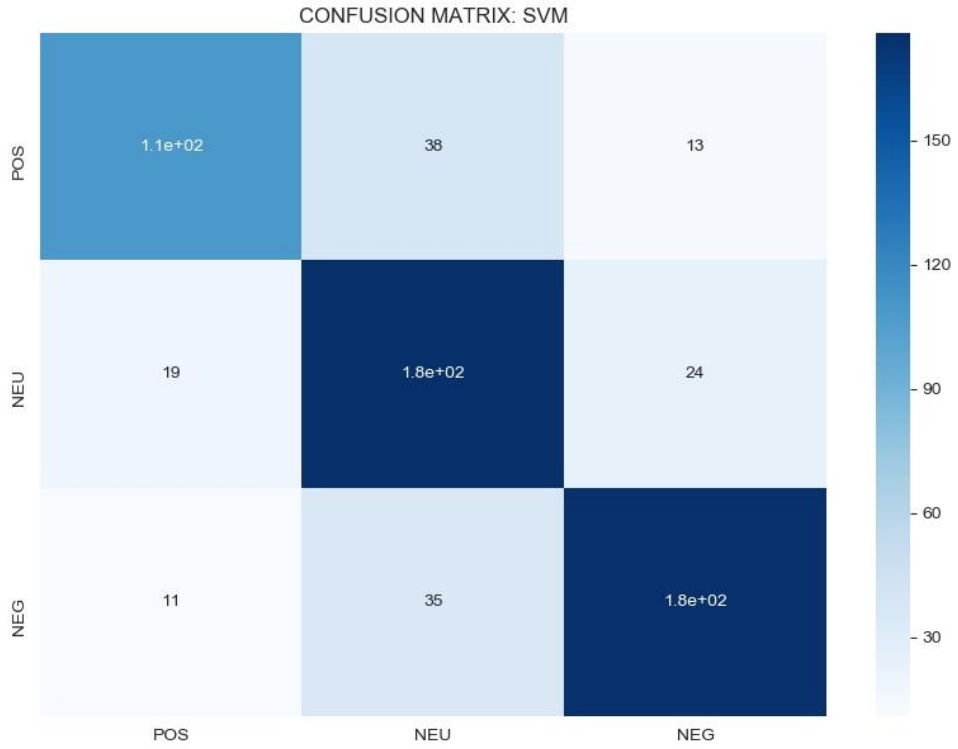

Figura 4-6 Matriz de confusión de SVM (elaboración propia)

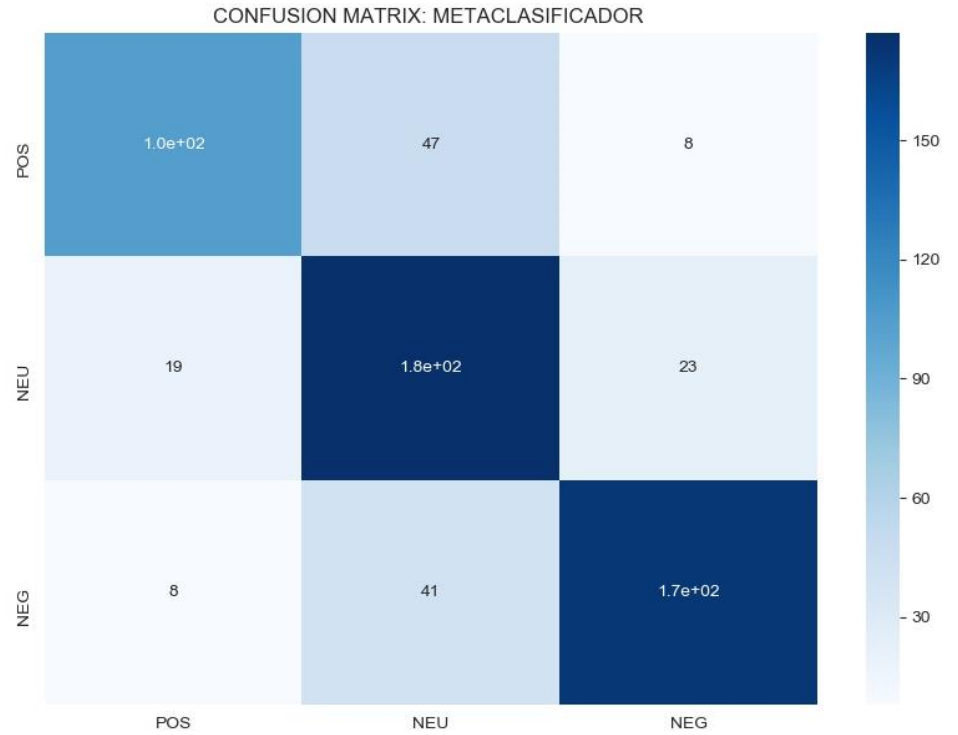

Figura 4-7 Matriz de confusión del metaclasificador inicial (elaboración propia)

En vista a las matrices obtenidas en Figura 4-6 y Figura 4-7, SVM y el metaclasificador inicial poseen características similares en términos de confusión. Son más robustos que

Gaussian NB y KNN, pero siguen teniendo problemas para predecir correctamente los datos pertenecientes a la clase neutra.

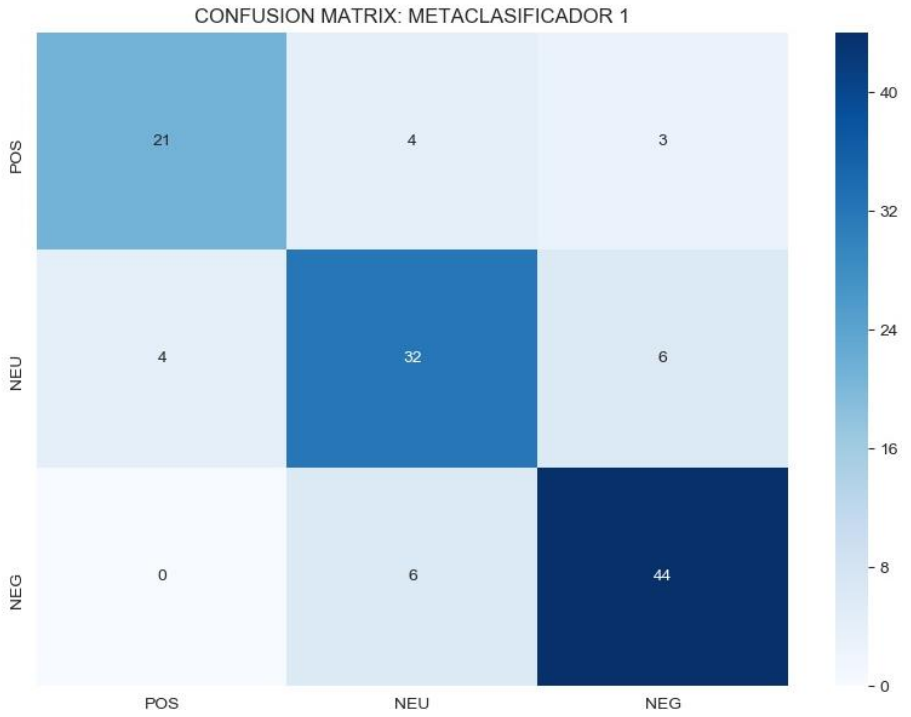

Figura 4-8 Matriz de confusión del metaclasificador final (elaboración propia)

En Figura 4-9 se aprecia cierta mejora en la clasificación de datos de la clase neutral y un alto porcentaje de predicciones correctas en cada una de las clases.

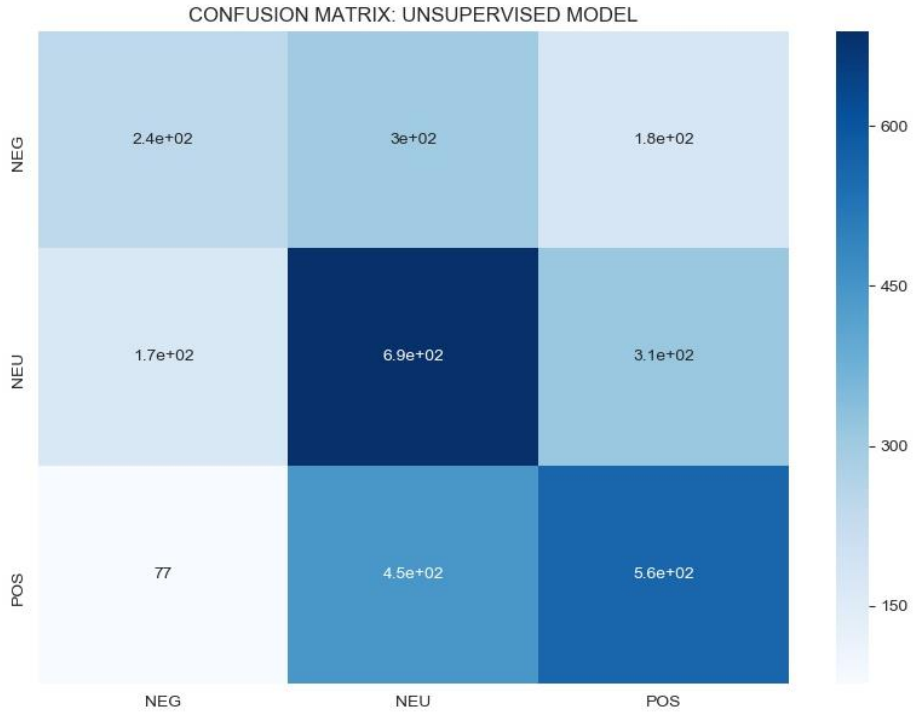

Figura 4-9 Matriz de confusión del modelo sin supervisión (elaboración propia)

En Figura 4-3, se comprobó la poca fiabilidad del algoritmo de aprendizaje sin supervisión, con un valor F-1 muy por debajo del de cualquier otro modelo. Como consecuencia, se obtiene

una matriz de confusión homogénea (Figura 4-9), en la que las clases son confundidas con cierta frecuencia.

5 CONCLUSIONES Y LÍNEAS FUTURAS

5.1 Conclusiones

En vista a los resultados obtenidos, el presente TFG se puede concluir con dos ideas principales. La primera de ellas es la poca fiabilidad que presentó el modelo de aprendizaje sin supervisión a la hora de predecir la polaridad de un conjunto de datos. Dicho modelo presentaba una serie de problemas que consistían principalmente en la limitación de traducción de textos; ya sea por la ilegibilidad de su contenido (faltas de ortografía, vulgarismos, etc.), o por limitaciones impuestas por la librería. Este problema en particular descartó el uso del algoritmo *Vader* como modelo principal de análisis de sentimiento; entrando en juego *Polyglot,* un algoritmo que no solo era poco flexible con el contenido del texto, sino que además no aceptaba *emojis* dentro de este.

En contraposición, se han generado una serie modelos de aprendizaje con supervisión lo suficientemente precisos como para considerarlos sobradamente fiables, obteniendo en el mejor de estos un valor F-1 por encima del 80%.

Por otra parte, atendiendo a los resultados obtenidos con el modelo LDA, se obtuvieron las palabras más representativas del conjunto de *tweets* de las Fuerzas Armadas (rey, ejército, trabajo, unidad, tropa, …) y fueron agrupadas en cuatro categorías distintas. De este modo, es posible hacerse una idea general de los temas que más circulan en las cuentas de los distintos ejércitos sin necesidad de leer los 3000 *tweets* importados, siendo suficiente únicamente las palabras más importantes de cada categoría.

Por último, es importante señalar, que del total de *tweets* importados solo un 25% de estos eran considerados negativos. Dentro de los *tweets* negativos una gran parte de estos eran ajenos a contenidos propios de las Fuerzas armadas y atendían a otros asuntos diferentes.

5.2 Líneas futuras

Durante la realización del TFG se pudo apreciar una falta generalizada de *lexicones* en español en la red. Por estas dos razones, se propone como líneas futuras la elaboración de un *lexicón* en castellano además del diseño de un algoritmo de aprendizaje sin supervisión que haga uso de este para calcular la polaridad de un texto.

Si bien en el presente TFG se ha abordado un estudio de la polaridad de los *tweets* publicados en las diferentes cuentas de los tres ejércitos, se propone también como posible línea futura entrenar un algoritmo de aprendizaje supervisado para que sea capaz de predecir la temática de cada *tweet*.

Por último, tras analizar las cada una de las distintas matrices de confusión de los algoritmos de aprendizaje, se pudo comprobar la dificultad generalizada de estos para predecir correctamente la clase neutral. Por este motivo, se propone optimizar los modelos a fin de subsanar este inconveniente y aumentar la precisión del modelo.

6 BIBLIOGRAFÍA

[1] S. kayte, «Medium,» 30 Agosto 2019. [En línea]. Available: https://medium.com/@bsangramsing/building-a-natural-language-processing-pipeline-9d93fa8ebdfb. [Último acceso: 2020 Enero 3].

[2] C. Español, «Medium,» 6 Noviembre 2017. [En línea]. Available: https://medium.com/@CKEspanol/what-are-the-different-levels-of-nlp-how-do-these-integrate-with-information-retrieval-c0de6b9ebf61. [Último acceso: 2020 Enero 6].

[3] A. Géron, Hands-On Machine Learning with Scikit-Learn & Tensorflow, O'Reilly, 2017.

[4] A. Das, «becominghuman,» Medium, 26 Marzo 2017. [En línea]. Available: https://becominghuman.ai/the-very-basics-of-reinforcement-learning-154f28a79071. [Último acceso: Enero 3 2020].

[5] I. H. Witten, «Data Mining,» de *Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, 2017, p. 654.

[6] M. Lopes, «towardsdatascience,» 28 Marzo 2017. [En línea]. Available: https://towardsdatascience.com/is-lda-a-dimensionality-reduction-technique-or-a-classifier-algorithm-eeed4de9953a.

[7] P. Yu, «towardsdatascience,» Medium, 5 Noviembre 2019. [En línea]. Available: https://towardsdatascience.com/how-to-access-twitters-api-using-tweepy-5a13a206683b. [Último acceso: 20 Diciembre 2019].

[8] Tweepy, «Tweepy,» [En línea]. Available: http://docs.tweepy.org/en/latest/api.html.

[9] B. Stecanella, «Monkey Learn,» 10 Mayo 2019. [En línea]. Available: https://monkeylearn.com/blog/what-is-tf-idf/. [Último acceso: 8 Febrero 2020].

[10] C. Hutto, *vaderSentiment.py,* 2017.

[11] P. Pandey, «Medium,» Analytics Vidhya, 23 Septiembre 2018. [En línea]. Available: https://medium.com/analytics-vidhya/simplifying-social-media-sentiment-analysis-using-vader-in-python-f9e6ec6fc52f. [Último acceso: 12 Febrero 2020].

[12] PyPI, 3 Jul 2016. [En línea]. Available: https://pypi.org/project/polyglot/.

[13] I. Rubio, «Amazon prescinde de una inteligencia artificial por discriminar a las mujeres,» *ElPaís,* 12 Octubre 2018.

[14] S. JAIN, «Analytics Vidhya,» 22 Junio 2017. [En línea]. Available: https://www.analyticsvidhya.com/blog/2017/06/a-comprehensive-guide-for-linear-ridge-and-lasso-regression/.

[15] «Scikit-Learn,» 2017. [En línea]. Available: https://scikit-learn.org/stable/modules/cross_validation.html.

[16] Scikit-Learn, 2017. [En línea]. Available: https://scikit-learn.org/stable/modules/naive_bayes.html. [Último acceso: 24 Enero 2020].

[17] Scikit-Learn, 2017. [En línea]. Available: https://scikit-learn.org/stable/modules/svm.html. [Último acceso: Enero 27 2020].

[18] H. Kandan, «towardsdatascience,» towardsdatascience, 30 Agosto 2017. [En línea]. Available: https://towardsdatascience.com/understanding-the-kernel-trick-e0bc6112ef78.

[19] Scikit-Learn, 2017. [En línea]. Available: https://scikit-learn.org/stable/modules/neighbors.html. [Último acceso: 24 Enero 2020].

[20] A. P. a. C. Williams, «brilliant,» brilliant, [En línea]. Available: https://brilliant.org/wiki/k-nearest-neighbors/. [Último acceso: 11 Febrero 2020].

[21] T. SRIVASTAVA, Analytics Vidhya, 27 Marzo 2018. [En línea]. Available: https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/.

[22] Scikit-Learn. [En línea]. Available: https://scikit-learn.org/stable/modules/multiclass.html.

[23] J. C. S. Sande, «Análisis de Sentimientos en Twitter,» 2018.

[24] O. Harrison, «towardsdatascience,» towardsdatascience, 10 Septiembre 2018. [En línea]. Available: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761. [Último acceso: 2020 02 11].

[25] G. R. d. Villa, «Medium,» 18 Mayo 2018. [En línea]. Available: https://medium.com/@gruizdevilla/introducci%C3%B3n-a-word2vec-skip-gram-model-4800f72c871f.

ANEXO I: CÓDIGO EMPLEADO

```
import preprocesamiento as prep
import pandas as pd
import sentiment as sent
from sentiment import LDA_Model

tweets = prep.tweet()
data = {"Tweets":tweets.message[:3000],
"Sentiment":tweets.sentiment[:3000]}
df = pd.DataFrame(data)
LDA_Model(df)
model = sent.model(df)
print(model.algorithm['SUPERVISED'],
'\n\n',model.algorithm['UNSUPERVISED'])
```

Tabla A-1 main.py

```
import tweepy
import json
import time
consumer_key = '8zLve9P3dx7Cjudh9hLQtByI2'
consumer_secret = 'CgSjulbkLOLxWtuvQnMSEzNhScKUgYs8A4XL5y6kzAvxxzMfGD'
access_token = '1208666253235228673-iF8L5DOXmS62f1vjCR1DihkfPHnED2'
access_token_secret = 'u6TBtelTL4UFi3NKED2lQgA2YCrYEz8Gh2SbPheWsCZPH'
def authentication():
    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_token, access_token_secret)
    api1 = tweepy.API(auth)
    return api1

def Buscar_Tweet(max_tweets,*busquedas):
    api = authentication()
    tags = []
    tweets= []
    for busqueda in busquedas:
        for i in range(max_tweets):
            tags.append(busqueda)
        busqueda = busqueda + "-filter:retweets"
        try:
            [tweets.append(status._json) for status in
            tweepy.Cursor(api.search, q=busqueda,
            tweet_mode='extended',lang='es',locale='es').items(max_tweets)]
        except tweepy.TweepError:
            time.sleep(60 * 15)
            continue
        for i in range(len(tweets)):
            tweets[i]['tag'] = tags[i]
    with open("tweet1.json","w") as file:
        json.dump(tweets,file)

Buscar_Tweet(1000,"@Armada_esp","@EjercitoTierra","@EjercitoAire")
```

Tabla A-2 data.py

```
import json
import re
import pandas as pd
from nltk.corpus import stopwords
class tweet:
    message = []
    user = []
    location = []
    tag = []
    sentiment = []
    def __init__(self):
        with open("tweet.json","r") as file:
            df = pd.read_excel("mensaje.xlsx", sheet_name="sentimiento")
            self.sentiment = df['Sentimiento']
            data = json.load(file)
            for i in range(len(data)):
                self.message.append(Normalize_Text(data[i]['full_text']))
                self.user.append(data[i]['id'])
                self.location.append(data[i]['user']['location'])
                self.tag.append(data[i]['tag'])
    @staticmethod
    def Army(self, army):
        lista = []
        for i in range(len(self.message)):
            if army == self.tag[i]:
                lista.append(self.message[i])
        return lista
def Tokenize(tweet):
    tokens = [t for t in tweet.split()]
    return tokens
def Normalize_Text(text):
    punctuation = ["#", "!", "¡", "?", "¿", ".", "(", ")", "'",
                   ",",">", "<",":", "/", "º", "°", "\"", "%",
                   "€", "$", "f-","]","[", "‘", "’", "“", "”",
                   "«","»", "=", "*", ";","&","±","_",
                   "p-","s-","-"]
    tildes = [("á", "a"), ("é", "e"), ("í", "i"), ("ó", "o"),
              ("ú", "u")]
    argot = [('d', 'de'), ('(k|q)', 'que'),('xo', 'pero'),
             ('xra', 'para'),('fvor','favor'),('dnd','donde'),
             ('tb','también'), ('tuit','tweet'),('x', 'por'),
             ('(xfa|xf|pf|plis|pls|porfa|please)', 'por favor'),
             ('(tq|tk)', 'te quiero'),('(tqm|tkm)',
             'te quiero mucho'),('+', 'más'),('(lol|xd)', 'jaja'),
             ('(omg|OMG)','Oh dios mío'),('tuitear',
             'enviar tweets'),('tks', 'gracias'),
             ('wapo','guapo'),('(pq|pk)','porque')]
    for i in range(len(text)):
        #Se elimina url:
        text= re.sub(r"http\S+", "", text)
        #Se elimina salto de linea:
        text = re.sub("\n", "", text)
        #Se eliminan mayúsculas:
        text= text.lower()
        #Se eliminan nombres:
        text = re.sub('@[^\s]+', '', text)
        #Se eliminan tildes:
        for j in range(len(tildes)):
            text = re.sub(re.escape(tildes[j][0]),
            re.escape(tildes[j][1]), text)
```

```
        #Se elimina puntuación:
        for j in range(len(punctuation)):
            text = re.sub(re.escape(punctuation[j]), " ", text)
        text = "".join(text)
        lista = Tokenize(text)
        for j in range(len(lista)):
        # Se eliminan números:
            lista[j] = ''.join([i for i in lista[j] if
            not i.isdigit()])
        # Se reemplaza argot por palabras:
            for k in range(len(argot)):
                if lista[j] == argot[k][0]:
                    lista[j] = argot[k][1]
            # Se eliminan 'palabras de parada:
            if lista[j] in stopwords.words('spanish'):
                lista[j]= ""
            #Se normalizan las risas:
            lista[j] = re.sub(r'j{2,}', 'j', lista[j])
            lista[j] = re.sub(r'ja{2,}', 'ja', lista[j])
        text = " ".join(lista)
        return text
def Load_Emoji():
    emoji = []
    description = []
    file = open("emoji_utf8_lexicon.txt", "r", encoding='utf-8')
    for line in file:
        for line1 in line.split('\t'):
            emoji.append(line1)
            description.append(line1)
    file.close()
    return emoji
def Emoji(tweet, emoji):
    for i in range(len(emoji)):
        new_emoji = " "+ emoji[i] +" "
        tweet = re.sub(re.escape(emoji[i]), new_emoji,tweet)
    return tweet
```

Tabla A-3 preprocesamiento.py

```
import seaborn as sns
from sklearn.metrics import confusion_matrix
import gensim
import pandas as pd
import pyLDAvis.gensim
import matplotlib.pyplot as plt
sns.set_style('whitegrid')
def Plot_LDA(corpus):
    #Se carga el diccionario, el corpus y el modelo LDA:
    dictionary = \
    gensim.corpora.Dictionary.load('dictionary.gensim')
    lda = \
    gensim.models.ldamodel.LdaModel.load('model10.gensim')
    #Se presenta en pantalla en 'Intertopic distance map':
    lda_display = \
    pyLDAvis.gensim.prepare(lda, corpus,
    dictionary, sort_topics=False)
    pyLDAvis.show(lda_display, local=False)
def Confusion_Matrix(y_test,y_pred,model):
    matrix = confusion_matrix(y_test,y_pred)
```

```
    try:
        df = pd.DataFrame(matrix, index=["POS", "NEU", "NEG"],
        columns=["POS", "NEU", "NEG"])
    except ValueError:
        df = pd.DataFrame(matrix,
        index=["FALSE", "NEG", "NEU","POS"],
        columns=["FALSE", "NEG", "NEU","POS"])
        df = df.drop('FALSE',axis=1)
        df = df.drop('FALSE', axis=0)
    plt.figure(figsize=(10, 7))
    sns.heatmap(df, annot=True, cmap= plt.cm.Blues)
    image = "matrix " + model + ".jpg"
    plt.title("CONFUSION MATRIX: "+ model)
    plt.savefig(image)
    plt.close()
def Plot_Data(tweet):
    data = {"Tweet":tweet.message, "Sentimiento":tweet.sentiment}
    df = pd.DataFrame(data)
    return df
def Sentiment_distribution(data):
    pos = 0;
    neu = 0;
    neg = 0;
    lista = data['Sentiment']
    for sentiment in lista:
        try:
            if sentiment == 'POS':
                pos += 1
            elif sentiment == 'NEU':
                neu += 1
            elif sentiment == 'NEG':
                neg += 1
        except KeyError:
            pass
    pos = (pos / len(data)) * 100;
    neg = (neg / len(data))* 100;
    neu = (neu / len(data)) * 100
    print("POS: ", pos, "  NEU: ", neu, "  NEG: ", neg)
    nombres = ['POS', 'NEU', 'NEG']
    datos = [pos, neu, neg]
    plt.bar(width=0.7, color=['green', 'yellow', 'red'],
    x=nombres, height=datos,edgecolor='black')
    plt.gca().set_yticklabels(['{:.0f}%'.format(x)
    for x in plt.gca().get_yticks()])
    plt.title("DISTRIBUCIÓN DEL SENTIMIENTO DE LA MUESTRA")
    plt.savefig("Sentimiento.jpg")
    plt.close()
```

Tabla A-4 plot.py

```
from gensim import models
from gensim.corpora import Dictionary
from sklearn.metrics import precision_recall_fscore_support
from sklearn.neighbors import KNeighborsClassifier
from plot import Confusion_Matrix as matrix
from plot import Plot_LDA
from sklearn.model_selection \
import train_test_split, cross_val_score
import re
import pandas as pd
```

```
from vaderSentiment.vaderSentiment \
import SentimentIntensityAnalyzer
from sklearn.naive_bayes import GaussianNB
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn import metrics
from sklearn import svm
from textblob import TextBlob
import numpy as np
from textblob.exceptions import NotTranslated
from preprocesamiento import Tokenize
from polyglot.text import Text
class model:
    algorithm = {'SUPERVISED':{},'UNSUPERVISED':{}}
    def __init__(self,df):
        self.algorithm['SUPERVISED']=Sent_analysis(df,plot=False,cros_val=False)
        self.algorithm['UNSUPERVISED'] = Polyglot_analysis(df,plot=True)
def Polyglot_analysis(data,plot):
    analyser = SentimentIntensityAnalyzer()
    data['Tweets'] = Remove_Emoji(data['Tweets'])
    sentiment = []
    for tweet in data['Tweets']:
        tweet1 = Text(tweet)
        try:
            score = tweet1.polarity
            if score >0:
                score = 'POS'
            elif score < 0:
                score = 'NEG'
            else:
                score = 'NEU'
            sentiment.append(score)
        except ValueError:
            tweet = TextBlob(tweet)
            try:
                tweet = tweet.translate(to='en')
                score = analyser.polarity_scores(tweet)
                #Vader compound score metric:
                if score['compound'] >= 0.05:
                    score['compound'] = 'POS'
                elif (score['compound']<0.05) and (score['compound']>-0.05):
                    score['compound'] = 'NEU'
                else:
                    score['compound'] = 'NEG'
                sentiment.append(score['compound'])
            except NotTranslated:
                sentiment.append(False)
    precision, recall, f1_score, support = \
    precision_recall_fscore_support(data['Sentiment'],
    sentiment,average='weighted')
    if plot == True:
        matrix(data['Sentiment'],sentiment, "UNSUPERVISED MODEL")
        print("UNSUPERVISED: \n",
        metrics.classification_report(data['Sentiment'], sentiment,
        target_names=["FALSE", "NEU", "NEG", "POS"]))
    data = {'Precision':precision, 'Recall':recall,'Score':f1_score}
    model = pd.DataFrame(data, index= [0])
    return model
def Remove_Emoji(tweets):
    index = 0
    emoji_pattern = re.compile("["
```

```
    u"\U0001F600-\U0001F64F"  # emoticonos
    u"\U0001F300-\U0001F5FF"  # simbolos y pictogramas
    u"\U0001F680-\U0001F6FF"  # transpore y mapas
    u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
    "]+", flags=re.UNICODE)
    for tweet in tweets:
        lista = []
        tweet = Text(tweet)
        tweet = Tokenize(tweet)
        for word in tweet:
            word = re.sub(emoji_pattern, '', word)
            lista.append(word)
        tweet = " ".join(lista)
        tweets[index] = tweet
        index +=1
    return tweets
def LDA_Model(data):
    #Se eliminan emoji
    data['Tweets'] = Remove_Emoji(data['Tweets'])
    #Se extraen los sustantivos
    data['Tweets'] = PoS(data=data,allowed_postags=['NOUN'])
    #Se elimina el ruido
    data['Tweets'] = Remove_noise(data['Tweets'])
    #Se genera un corpus
    dictionary = Dictionary(data['Tweets'].to_list())
    dictionary.compactify()
    dictionary.filter_extremes(no_below=2, no_above=0.9,keep_n=None)
    dictionary.compactify()
    dictionary.save('dictionary.gensim')
    corpus = [dictionary.doc2bow(text) for text in data['Tweets'].to_list()]
    #Se genera modelo lda
    ldamodel = models.ldamodel.LdaModel(corpus, num_topics=4,
    id2word=dictionary, passes=50)
    print(ldamodel.print_topics(num_topics=4, num_words=10))
    ldamodel.save('model10.gensim')
    #Se muestra graficamente
    display_topics(ldamodel)
    Plot_LDA(corpus)
def display_topics(model):
  for topic_idx, topic in enumerate(model.print_topics()):
    print ("Topic %d:" % (topic_idx))
    print(" ".join(re.findall(r'\"(.[^"]+).?', topic[1])), "\n")
def PoS(data, allowed_postags):
    index = 0
    for tweet in data['Tweets']:
        lista = []
        for i in range(len(tweet)):
            try:
                tweet1 = Text(tweet)
                tag = tweet1.pos_tags
                if tweet1.words[i].pos_tag in allowed_postags:
                    lista.append(str(tweet1.words[i]))
            except ValueError:
                pass
            except IndexError:
                pass
        data['Tweets'][index] = lista
        index += 1
    return data
def Sent_analysis(data,plot,cros_val):
    #Inicializo los resultados a cero
```

```
    p = []; r = []; s = []
    X = data.drop('Sentiment', axis=1)
    y = data.drop('Tweets', axis=1)
    #Extacción de características
    count_vect = TfidfVectorizer(analyzer='word')
    X = count_vect.fit_transform(X['Tweets']).toarray()
    #Partición del corpus en conjunto de entrenamiento y de test
    X_train_counts,X_test_counts,y_train,y_test=train_test_split(X,y,
    test_size=0.2,random_state=42)
    #Gaussian Naive Bayes
    clf = GaussianNB().fit(X_train_counts, y_train)
    y_pred = clf.predict(X_test_counts)
    precision, recall, score, support = \
    precision_recall_fscore_support(y_test,y_pred,average='weighted')
    p.append(precision); r.append(recall); s.append(score)
    #Supported Vector Machines
    clf1 = svm.SVC(C=10, cache_size=200, class_weight=None,coef0=0.0,
    decision_function_shape='ovr', degree=3,gamma=0.1, kernel='rbf',max_iter=-1,
    probability=False,random_state=None,
    shrinking=True,tol=0.001, verbose=False)
    clf1.fit(X_train_counts, y_train)
    y_pred1 = clf1.predict(X_test_counts)
    precision, recall, score, support = \
    precision_recall_fscore_support(y_test,
    y_pred1, average='weighted')
    p.append(precision); r.append(recall); s.append(score)
    #K-nearest neighbors
    clf2 = KNeighborsClassifier(algorithm='auto', leaf_size=30,
    metric='euclidean',metric_params=None, n_jobs=None,n_neighbors=19,
    p=2,weights='distance')
    clf2.fit(X_train_counts, y_train)
    y_pred2 = clf2.predict(X_test_counts)
    precision, recall, score, support = precision_recall_fscore_support(y_test,
    y_pred2,average='weighted')
    p.append(precision); r.append(recall); s.append(score)
    if plot == True:
        print("GaussNB: \n",
        metrics.classification_report(y_test,y_pred,
        target_names=["POS", "NEU", "NEG"]))
        print("SVM: \n",
        metrics.classification_report(y_test,y_pred1,
        target_names=["POS", "NEU", "NEG"]))
        print("KNN: \n",
        metrics.classification_report(y_test,y_pred2,
        target_names=["POS", "NEU", "NEG"]))
        matrix(y_test, y_pred, "GaussianNB")
        matrix(y_test, y_pred1, "SVM")
        matrix(y_test, y_pred2, "KNN")
    if cros_val == True:
        cross = cross_val_score(clf, X, y, cv=10)
        print("GAUSS CROSS_VAL_SCORE: ", cross)
        cross1 = cross_val_score(clf1, X, y, cv=10)
        print("SVM CROSS_VAL_SCORE: ", cross1)
        cross2 = cross_val_score(clf2, X_train_counts,y_train, cv=10)
        print("KNN CROSS_VAL_SCORE: ", cross2)
    # Metaclasificador
    y_pred = np.asarray(Transform_to_float(y_pred))
    y_pred1 = np.asarray(Transform_to_float(y_pred1))
    y_pred2 = np.asarray(Transform_to_float(y_pred2))
    alpha = s
```

```
    precision, recall, score, support = \
    Meta_Classifier(y_pred, y_pred1, y_pred2,alpha, y_test, plot= True)
    p.append(precision); r.append(recall); s.append(score)
    precision, recall, score, support = \
    Meta_Classifier1(y_pred, y_pred1, y_pred2,y_test['Sentiment'],plot)
    p.append(precision); r.append(recall); s.append(score)
    #Resultados(model)
    data = {'Model': ['GaussNB', 'SVM', 'KNN', 'META', 'META 1'],
    'Precision': p, 'Recall': r,'Score': s}
    model = pd.DataFrame(data)
    return model
def Transform_to_float(y_pred):
    lista = []
    for sentiment in y_pred:
        if sentiment == "POS":
            lista.append(1)
        elif sentiment == "NEU":
            lista.append(0)
        elif sentiment == "NEG":
            lista.append(-1)
    return lista
def Transform_to_string(y_pred):
    lista = []
    for sentiment in y_pred:
        sentiment = round(sentiment)
        if sentiment == 1:
            lista.append("POS")
        elif sentiment == 0:
            lista.append("NEU")
        elif sentiment == -1:
            lista.append("NEG")
    return lista
def Remove_noise(tweets):
    noise = ["buenos", "buenas", "dias", "dia", "noches", "noche",
    "saludo","saludos","año","años","vez","veces","favor","adios",
    "gracias", "tardes", "gracia", "enhorabuena","dia"]
    index = 0
    for tweet in tweets:
        for word in tweet:
            for element in noise:
                if word == element:
                    tweets[index].remove(element)
        index += 1
    return tweets
def Meta_Classifier1(y_pred, y_pred1, y_pred2, y, plot):
    y = np.asarray(Transform_to_float(y)).reshape(-1,1)
    X = np.asarray([y_pred, y_pred1, y_pred2])
    X = X.transpose()
    #Se genera conjunto de entrenamiento y de test
    meta_train, meta_test ,y_train, y_test = \
    train_test_split(X,y, test_size=0.2, random_state=42)
    #Se entrena el modelo
    clf3 = svm.SVC(C=10, cache_size=200, class_weight=None,coef0=0.0,
    decision_function_shape='ovr', degree=3,gamma=0.1, kernel='rbf',
    max_iter=-1, probability=False,random_state=None, shrinking=True,
    tol=0.001, verbose=False)
    clf3.fit(meta_train, y_train)
    y_pred3 = clf3.predict(meta_test)
    #Obtencion de resultados
    precision, recall, score, support = \
    precision_recall_fscore_support(y_test,y_pred3,average='weighted')
```

```
    if plot == True:
        matrix(y_test, y_pred3, "METACLASIFICADOR 1")
    print("MetaClassifier: \n",metrics.classification_report(y_test, y_pred3))
    return precision, recall, score, support
def Meta_Classifier(y_pred, y_pred1, y_pred2, alpha, y_test, plot):
    suma = alpha[0] + alpha[1] + alpha[2]
    alpha[0] = alpha[0]/suma; alpha[1] = alpha[1]/suma
    alpha[2] = alpha[2]/suma
    #Obtencion de la matriz de prediccion
    y_pred3 = alpha[0]*y_pred + alpha[1]*y_pred1 + alpha[2]*y_pred2
    #Se asigna un valor de acuerdo con la metrica establecida
    for i in range(len(y_pred3)):
        if y_pred3[i] < -0.32:
            y_pred3[i] = -1
        elif y_pred3[i] > 0.34:
            y_pred3[i] = 1
        else:
            y_pred3[i] = 0
    y_pred3 = Transform_to_string(y_pred3)
    #Obtencion de resultados
    precision, recall, score, support = \
    precision_recall_fscore_support(y_test, y_pred3,
    average='weighted')
    if plot == True:
        print("MetaClassifier : \n",
        metrics.classification_report(y_test,y_pred3,
        target_names=["POS", "NEU", "NEG"]))
        matrix(y_test, y_pred3, "METACLASIFICADOR ")
    return precision, recall, score,support
```

Tabla A-5 sentiment.py