Procesamiento de Lenguaje Natural

Eric S. Téllez Mario Graff

Tabla de contenidos

Prefacio

El curso trata de ser auto-contenido, es decir, no debería de ser necesario leer otras fuentes para poder entenderlo y realizar las actividades. De cualquier manera es importante comentar que el curso está basado en los siguientes libros de texto:

- Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Third Edition draft. Daniel Jurafsky and James H. Martin. pdf
- Introduction to machine learning, Third Edition. Ethem Alpaydin. MIT Press.
- An Introduction to Statistical Learning with Applications in R. Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. Springer Texts in Statistics.
- All of Statistics. A Concise Course in Statistical Inference. Larry Wasserman. MIT Press.
- An Introduction to the Bootstrap. Bradley Efron and Robert J. Tibshirani. Monographs on Statistics and Applied Probability 57. Springer-Science+Business Media.
- Understanding Machine Learning: From Theory to Algorithms. Shai Shalev-Shwartz and Shai Ben-David. Cambridge University Press.

Notación

La Tabla 1 muestra la notación que se seguirá en este documento.

Tabla 1: Notación

Símbolo	Significado
x	Variable usada comunmente como entrada
y	Variable usada comunmente como salida
$\mathbb{R}$	Números reales
$\mathbf{x}$	Vector Columna $\mathbf{x} \in \mathbb{R}^d$
d	Dimensión
$\mathbf{w} \cdot \mathbf{x}$	Producto punto donde $\mathbf{w}$ y $\mathbf{x} \in \mathbb{R}^d$
$\mathcal{D}$	Conjunto de datos
$\mathcal{T}$	Conjunto de entrenamiento
$\mathcal{V}$	Conjunto de validación
$\mathcal{G}$	Conjunto de prueba
N	Número de ejemplos
K	Número de clases
$\mathbb{P}(\cdot)$	Probabilidad
$\mathcal{X}, \mathcal{Y}$	Variables aleatorías
$\mathcal{N}(\mu, \sigma^2)$	Distribución Normal con parámetros μ y σ^2
$f_{\mathcal{X}}$	Función de densidad de probabilidad de $\mathcal{X}$
$\mathbb{1}(e)$	Función para indicar; 1 only if e is true
Ω	Espacio de búsqueda
$\mathbb{V}$	Varianza
$\mathbb{E}$	Esperanza

Licencia

1 Introducción

El **objetivo** de la unidad es

2 Manejando Texto

El **objetivo** de la unidad es

Paquetes usados

```
from microtc.params import OPTION_GROUP, OPTION_DELETE,\
                           OPTION_NONE
from microtc.textmodel import SKIP_SYMBOLS
from b4msa.textmodel import TextModel
from b4msa.lang_dependency import LangDependency
from nltk.stem.snowball import SnowballStemmer
import unicodedata
import re
```

2.1 Introducción

Se podría suponer que el texto se que se analizará está bien escrito y tiene un formato adecuado para su procesamiento. Desafortunadamente, la realidad es que en la mayoría de aplicaciones el texto que se analiza tiene errores de ortográficos, errores de formato y además no es trivial identificar la unidad mínima de procesamiento que podría ser de manera natural, en el español, las palabras. Por este motivo, esta unidad trata técnicas comunes que se utilizan para normalizar el texto, esta normalización es un proceso previo al desarrollo de los algoritmos de PLN.

La Figura 2.1 esquematiza el procedimiento que se presenta en esta unidad, la idea es que se un texto pasa primeramente a un proceso de normalización (Sección 2.2 y Sección 2.3), para después ser segmentado (ver Sección 2.4) y el resultado es lo que se utiliza para modelar el lenguaje.

Las normalizaciones y segmentaciones descritas en esta unidad se basan principalmente en las utilizadas en los siguientes artículos científicos.

Figura 2.1: Diagrama de Pre-procesamiento

1. An automated text categorization framework based on hyperparameter optimization (Tellez et al. (2018))
2. A simple approach to multilingual polarity classification in Twitter (Tellez, Miranda-Jiménez, Graff, Moctezuma, Suárez, et al. (2017))
3. A case study of Spanish text transformations for twitter sentiment analysis (Tellez, Miranda-Jiménez, Graff, Moctezuma, Siordia, et al. (2017))

2.2 Normalización de Texto Sintáctica

La descripción de las normalizaciones empieza presentando las que se puede aplicar a nivel de caracteres, sin la necesidad de conocer el significado de las palabras. También se agrupan en este conjunto aquellas transformaciones que se realizan mediante expresiones regulares o su búsqueda en una lista de palabras previamente definidas.

2.2.1 Entidades

La descripción de diferentes técnicas de normalización empieza con el manejo de entidades en el texto. Algunas entidades que se tratarán serán los nombres de usuario, números o URLs mencionados en un texto. Por otro lado están las acciones que se realizarán a las entidades encontradas, estas acciones corresponden a su borrado o remplazo por algún otro toquen.

2.2.1.1 Usuarios

En esta sección se trabajará con los nombres de usuarios que siguen el formato usado por Twitter. En un tuit, los nombres de usuarios son aquellas palabras que inician con el caracter @ y terminan con un espacio o caracter terminal. Las acciones que se realizarán con los nombres de usuario encontrados serán su borrado o reemplazo por una etiqueta en particular.

El procedimiento para encontrar los nombres de usuarios es mediante expresiones regulares, en particular se usa la expresión `@\S+`, tal y como se muestra en el siguiente ejemplo.

```
text = 'Hola @xx, @mm te está buscando'
re.sub(r"@\S+", "", text)
```

```
'Hola   te está buscando'
```

La segunda acción es reemplazar cada nombre de usuario por una etiqueta particular, en el siguiente ejemplo se reemplaza por la etiqueta `_usr`.

```
text = 'Hola @xx, @mm te está buscando'
re.sub(r"@\S+", "_usr", text)
```

```
'Hola _usr _usr te está buscando'
```

2.2.1.2 URL

Los ejemplos anteriores se pueden adaptar para manejar URL; solamente es necesario adecuar la expresión regular que identifica una URL. En el siguiente ejemplo se muestra como se pueden borrar las URLs que aparecen en un texto.

```
text = "puedes verificar que http://google.com esté funcionando"
re.sub(r"https?://\S+", "", text)
```

```
'puedes verificar que  esté funcionando'
```

2.2.1.3 Números

The previous code can be modified to deal with numbers and replace the number found with a shared label such as `_num`.

```
text = "acabamos de ganar 10 M"
re.sub(r"\d\d*\.?\d*|\d*\.\d\d*", "_num", text)
```

```
'acabamos de ganar _num M'
```

2.2.2 Ortografía

El siguiente bloque de normalizaciones agrupa aquellas modificaciones que se realizan a algún componente de tal manera que aunque impacta en su ortografía puede ser utilizado para reducir la dimensión y se ve reflejado en la complejidad del algoritmo.

2.2.2.1 Mayúsculas y Minúsculas

La primera de estas transformaciones es convertir todas los caracteres a minúsculas. Como se puede observar esta transformación hace que el vocabulario se reduzca, por ejemplo, las palabras *México* o *MÉXICO* son representados por la palabra *méxico*. Esta operación se puede realizar con la función `lower` tal y cómo se muestra a continuación.

```
text = "México"
text.lower()
```

```
'méxico'
```

2.2.2.2 Signos de Puntuación

Los signos de puntuación son necesarios para tareas como la generación de textos, pero existen otras aplicaciones donde los signos de puntuación tienen un efecto positivo en el rendimiento del algorithm, este es el caso de tareas de categorización de texto. El efecto que tiene el quitar los signos de puntuación es que el vocabulario se reduce. Los símbolos de puntuación se pueden remover teniendo una lista de los mismos, esta lista de signos de puntuación se encuentra en la variable `SKIP_SYMBOLS` y el siguiente código muestra un procedimiento para quitarlos.

```
text = "¡Hola! buenos días:"
output = ""
for x in text:
    if x in SKIP_SYMBOLS:
        continue
    output += x
output
```

```
'Hola buenos días'
```

2.2.2.3 Símbolos Diacríticos

Continuando con la misma idea de reducir el vocabulario, es común eliminar los símbolos diacríticos en las palabras. Esta transformación también tiene el objetivo de normalizar aquellos textos informales donde los símbolos diacríticos son usado con una menor frecuencia, en particular los acentos en el caso del español. Por ejemplo, es común encontrar la palabra *México* escrita como *Mexico*.

El siguiente código muestra un procedimiento para eliminar los símbolos diacríticos.

```
text = 'México'
output = ""
for x in unicodedata.normalize('NFD', text):
    o = ord(x)
    if 0x300 <= o and o <= 0x036F:
        continue
    output += x
output
```

```
'Mexico'
```

2.3 Normalización Semántica

Las siguientes normalizaciones comparten el objetivo con las normalizaciones presentadas hasta este momento, el cual es la reducción del vocabulario; la diferencia es que las siguientes utilizan el significado o uso de la palabra.

2.3.1 Palabras Comunes

Las palabras comunes (*stop words*) son palabras utilizadas frecuentemente en el lenguaje, las cuales son necesarias para comunicación, pero no aportan información para discriminar un texto de acuerdo a su significado.

The stop words are the most frequent words used in the language. These words are essential to communicate but are not so much on tasks where the aim is to discriminate texts according to their meaning.

Las palabras vacías se pueden guardar en un diccionario y el proceso de identificación consiste en buscar la existencia de la palabra en el diccionario. Una vez que la palabra analizada se encuentra en el diccionario, se procede a quitarla o cambiarla por un token particular. El proceso de borrado se muestra en el siguiente código.

```
lang = LangDependency('spanish')

text = '¡Buenos días! El día de hoy tendremos un día cálido.'
output = []
for word in text.split():
    if word.lower() in lang.stopwords[len(word)]:
        continue
    output.append(word)
```

```
output = " ".join(output)
output
```

```
'¡Buenos días! día hoy día cálido.'
```

2.3.2 Lematización y Reducción a la Raíz

La idea de lematización y reducción a la raíz (*stemming*) es transformar una palabra a su raíz mediante un proceso heurístico o morfológico. Por ejemplo, las palabras *jugando* o *jugaron* se transforman a la palabra *jugar*.

El siguiente código muestra el proceso de reducción a la raíz utilizando la clase `SnowballStemmer`.

```
stemmer = SnowballStemmer('spanish')

text = 'Estoy jugando futbol con mis amigos'
output = []
for word in text.split():
    w = stemmer.stem(word)
    output.append(w)
output = " ".join(output)
output
```

```
'estoy jug futbol con mis amig'
```

2.4 Segmentación

Una vez que el texto ha sido normalizado es necesario segmentarlo (*tokenize*) a sus componentes fundamentales, e.g., palabras o gramas de caracteres (q-grams) o de palabras (n-grams). Existen diferentes métodos para segmentar un texto, probablemente una de las más sencillas es asumir que una palabra está limitada entre dos espacios o signos de puntuación. Partiendo de las palabras encontradas se empiezan a generar los gramas de palabras, e.g., bigramas, o los gramas de caracteres si se desea solo generarlos a partir de las palabras.

2.4.1 Gramas de Palabras (n-grams)

El primer método de segmentación revisado es la creación de los gramas de palabras. El primer paso es encontrar las palabras las cuales se pueden encontrar mediante la función `split`; una vez que las palabras están definidas estás se pueden unir para generar los gramas de palabras del tamaño deseado, tal y como se muestra en el siguiente código.

```
text = 'Estoy jugando futbol con mis amigos'
words = text.split()
n = 3
n_grams = []
for a in zip(*[words[i:] for i in range(n)]):
    n_grams.append("~".join(a))
n_grams
```

```
['Estoy~jugando~futbol',
 'jugando~futbol~con',
 'futbol~con~mis',
 'con~mis~amigos']
```

2.4.2 Gramas de Caracteres (q-grams)

La segmentación de gramas de caracteres complementa los gramas de palabras. Los gramas de caracteres están definidos como la subcadena de longitud q. Este tipo de segmentación tiene la característica de que es agnóstica al lenguaje, es decir, se puede aplicar en cualquier idioma; contrastando, los gramas de palabras se pueden aplicar solo a los lenguajes que tienen definido el concepto de palabra, por ejemplo en el idioma chino las palabras no se pueden identificar como se pueden identificar en el español o inglés. La segunda característica importante es que ayuda en el problema de errores ortográficos, siguiendo una perspectiva de similitud aproximada.

El código para realizar los gramas de caracteres es similar a la presentada anteriormente, siendo la diferencia que el ciclo está por los caracteres en lugar de la palabras como se había realizado. El siguiente código muestra una implementación para realizar gramas de caracteres.

```
text = 'Estoy jugando'
q = 4
q_grams = []
for a in zip(*[text[i:] for i in range(q)]):
    q_grams.append("".join(a))
q_grams
```

```
['Esto',
 'stoy',
 'toy ',
 'oy j',
 'y ju',
 ' jug',
 'juga',
 'ugan',
 'gand',
 'ando']
```

2.5 TextModel

Habiendo descrito diferentes tipos de normalización (sintáctica y semántica) y el proceso de segmentación es momento para describir la librería B4MSA (Tellez, Miranda-Jiménez, Graff, Moctezuma, Suárez, et al. (2017)) que implementa estos procedimientos; específicamente, el punto de acceso de estos procedimientos corresponde a la clase `TextModel`. El método `TextModel.text_transformations` es el que realiza todos los métodos de normalización (Sección 2.2 y Sección 2.3) y el método `TextModel.tokenize` es el encargado de realizar la segmentación (Sección 2.4) siguiendo el flujo mostrado en la Figura 2.1.

2.5.1 Normalizaciones

El primer conjunto de parámetros que se describen son los que corresponden a las entidades (Sección 2.2.1). Estos parámetros tiene tres opciones, borrar (`OPTION_DELETE`), remplazar (`OPTION_GROUP`) o ignorar. Los nombres de los parámetros son:

- usr_option
- url_option
- num_option

que corresponden al procesamiento de usuarios, URL y números respectivamente. Adicionalmente, `TextModel` trata los emojis, hashtags y nombres, mediante los siguientes parámetros:

- emo_option
- hashtag_option
- ent_option

Por ejemplo, el siguiente código muestra como se borra el usuario y se reemplaza un hashtag; se puede observar que en la respuesta se cambian todos los espacios por el caracter `~` y se incluye ese mismo al inicio y final del texto.

```
tm = TextModel(hashtag_option=OPTION_GROUP,
               usr_option=OPTION_DELETE)
texto = 'mira @xyz estoy triste. #UnDiaLluvioso'
tm.text_transformations(texto)
```

```
'~mira~estoy~triste.~_htag~'
```

Siguiendo con las transformaciones sintácticas, toca el tiempo a describir aquellas que relacionadas a la ortografía (Sección 2.2.2) las cuales corresponden a la conversión a minúsculas, borrado de signos de puntuación y símbolos diacríticos. Estas normalizaciones se activan con los siguiente parámetros.

- lc
- del_punc
- del_diac

En el siguiente ejemplo se transforman el texto a minúscula y se remueven los signos de puntuación.

```
tm = TextModel(lc=True,
               del_punc=True,
               del_diac=False)
texto = 'Hoy está despejado.'
tm.text_transformations(texto)
```

```
'~hoy~está~despejado~'
```

Las normalizaciones semánticas (Sección 2.3) que se tienen implementadas en la librería corresponden al borrado de palabras comunes y reducción a la raíz; estás se pueden activar con los siguientes parámetros.

- stopwords
- stemming

Por ejemplo, las siguientes instrucciones quitan las palabras comunes y realizan una reducción a la raíz.

```
tm = TextModel(lang='es',
               stopwords=OPTION_DELETE,
               stemming=True)
texto = 'el clima es perfecto'
tm.text_transformations(texto)
```

```
'~clim~perfect~'
```

2.5.2 Segmentación

El paso final es describir el uso de la segmentación. La librería utiliza el parámetro `token_list` para indicar el tipo de segmentación que se desea realizar. El formato es una lista de número, donde el valor indica el tipo de segmentación. El número 1 indica que se realizará una segmentación por palabras, los número positivo corresponden a los gramas de caracteres y los números negativos a los gramas de palabras.

Por ejemplo, utilizando las normalizaciones que se tienen por defecto, el siguiente código segmenta utilizando gramas de caracteres de tamañan 4.

```
tm = TextModel(token_list=[4])
tm.tokenize('buenos días')
```

```
['q:~bue',
 'q:buen',
 'q:ueno',
 'q:enos',
 'q:nos~',
 'q:os~d',
 'q:s~di',
 'q:~dia',
 'q:dias',
 'q:ias~']
```

para poder identificar cuando se trata de un segmento que corresponde a una palabra o un grama de caracteres, a los últimos se les agrega el prefijo `q:`. Cabe mencionar que por defecto se remueven los símbolos diacríticos.

El ejemplo anterior, se utiliza para generar un grama de palabras de tamaño 2. Como se ha mencionado los gramas de palabras se especifican con números negativos siendo el valor absoluto el tamaño del grama.

```
tm = TextModel(token_list=[-2])
tm.tokenize('buenos días')
```

```
['buenos~dias']
```

Para completar la explicación, se combinan la segmentación de gramas de caracteres y palabras además de incluir las palabras en la segmentación.

```
tm = TextModel(token_list=[4, -2, -1])
tm.tokenize('buenos días')
```

```
['buenos~dias',
 'buenos',
 'dias',
 'q:~bue',
 'q:buen',
 'q:ueno',
 'q:enos',
 'q:nos~',
 'q:os~d',
 'q:s~di',
 'q:~dia',
 'q:dias',
 'q:ias~']
```

3 Modelado de Lenguaje

El **objetivo** de la unidad es

4 Fundamentos de Clasificación de Texto

El **objetivo** de la unidad es

Paquetes usados

```
from microtc.utils import tweet_iterator, load_model, save_model
from b4msa.textmodel import TextModel
from EvoMSA.tests.test_base import TWEETS
from EvoMSA.utils import bootstrap_confidence_interval
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import recall_score, precision_score, f1_score
from sklearn.naive_bayes import MultinomialNB
from scipy.stats import norm, multinomial, multivariate_normal
from scipy.special import logsumexp
from collections import Counter
from matplotlib import pylab as plt
from os.path import join
import numpy as np
```

4.1 Introducción

Text Categorization is an NLP task that deals with creating algorithms capable of identifying the category of a text from a set of predefined categories. For example, sentiment analysis belongs to this task, and the aim is to detect the polarity (e.g., positive, neutral, or negative) of a text. Furthermore, different NLP tasks that initially seem unrelated to this problem can be formulated as a classification one such as question answering and sentence entailment, to mention a few.

Text Categorization can be tackled from different perspectives; the one followed here is to treat it as a supervised learning problem. As in any supervised learning problem, the starting point is a set of pairs, where the first element of the pair is the input and the second one corresponds to the output. Let $\mathcal{D} = \{(\text{text}_i, y_i) \mid i = 1, \dots, N\}$ where $y \in \{c_1, \dots c_K\}$ and text_i is a text.

4.2 Modelado Probabilistico (Distribución Categórica)

4.2.1 Problema Sintético

The description of Bayes' theorem continues with an example of a Categorical distribution. A Categorical distribution can simulate the drawn of K events that can be encoded as characters, and ℓ repetitions can be represented as a sequence of characters. Consequently, the distribution can illustrate the generation sequences associated with different classes, e.g., positive or negative.

The first step is to create the dataset. As done previously, two distributions are defined, one for each class; it can be observed that each distribution has different parameters. The second step is to sample these distributions; the distributions are sampled 1000 times with the following procedure. Each time, a random variable representing the number of outcomes taken from each distribution is drawn from a Normal $\mathcal{N}(15, 3)$ and stored in the variable `length`. The random variable indicates the number of outcomes for each Categorical distribution; the results are transformed into a sequence, associated to the label corresponding to the positive and negative class, and stored in the list `D`.

```
pos = multinomial(1, [0.20, 0.20, 0.35, 0.25])
neg = multinomial(1, [0.35, 0.20, 0.25, 0.20])
length = norm(loc=15, scale=3)
D = []
m = {k: chr(122 - k) for k in range(4)}
id2w = lambda x: " ".join([m[_] for _ in x.argmax(axis=1)])
for l in length.rvs(size=1000):
    D.append((id2w(pos.rvs(round(l))), 1))
    D.append((id2w(neg.rvs(round(l))), 0))
```

The following table shows four examples of this process; the first column contains the sequence, and the second the associated label.

Text	Label
x w x x z w y	positive
y w z z z x w	negative
z x x x z x z w x w	positive
x w z w y z z z z w	negative

As done previously, the first step is to compute the likelihood given that dataset; considering that the data comes from a Categorical distribution, the procedure to estimate the parameters is similar to the ones used to estimate the prior. The following code estimates the data

parameters corresponding to the positive class. It can be observed that the parameters estimated are similar to the ones used to generate the dataset.

```
D_pos = []
[D_pos.extend(data.split()) for data, k in D if k == 1]
words, l_pos = np.unique(D_pos, return_counts=True)
w2id = {v: k for k, v in enumerate(words)}
l_pos = l_pos / l_pos.sum()
l_pos
array([0.25489421, 0.33854064, 0.20773186, 0.1988333 ])
```

An equivalent procedure is performed to calculate the likelihood of the negative class.

```
D_neg = []
[D_neg.extend(data.split()) for data, k in D if k == 0]
_, l_neg = np.unique(D_neg, return_counts=True)
l_neg = l_neg / l_neg.sum()
```

The prior is estimated with the following code, equivalent to the one used on all the examples seen so far.

```
_, priors = np.unique([k for _, k in D], return_counts=True)
N = priors.sum()
prior_pos = priors[1] / N
prior_neg = priors[0] / N
```

Once the parameters have been identified, these can be used to predict the class of a given sequence. The first step is to compute the likelihood, e.g., $\mathbb{P}(\text{w w x z}|\ \mathcal{Y})$. It can be observed that the sequence needs to be transformed into tokens which can be done with the `split` method. Then, the token is converted into an index using the mapping `w2id`; once the index is retrieved, it can be used to obtain the parameter associated with the word. The likelihood is the product of all the probabilities; however, this product is computed in log space.

```
def likelihood(params, txt):
    params = np.log(params)
    _ = [params[w2id[x]] for x in txt.split()]
    tot = sum(_)
    return np.exp(tot)
```

The likelihood combined with the prior for all the classes produces the evidence, which subsequently is used to calculate the posterior distribution. The posterior is then used to predict the class for all the sequences in $\mathcal{D}$. The predictions are stored in the variable `hy`.

```
post_pos = [likelihood(l_pos, x) * prior_pos for x, _ in D]
post_neg = [likelihood(l_neg, x) * prior_neg for x, _ in D]
evidence = np.vstack([post_pos, post_neg]).sum(axis=0)
post_pos /= evidence
post_neg /= evidence
hy = np.where(post_pos > post_neg, 1, 0)
```

4.2.2 Clasificador de Texto

The approach followed on text categorization is to treat it as supervised learning problem where the starting point is a dataset $\mathcal{D} = \{(\text{text}_i, y_i) \mid i = 1, \ldots, N\}$ where $y \in \{c_1, \ldots c_K\}$ and text_i is a text. For example, the next code uses a toy sentiment analysis dataset with four classes: negative (N), neutral (NEU), absence of polarity (NONE), and positive (P).

```
D = [(x['text'], x['klass']) for x in tweet_iterator(TWEETS)]
```

As can be observed, $\mathcal{D}$ is equivalent to the one used in the Categorical Distribution example. The difference is that sequence of letters is changed with a sentence. Nonetheless, a feasible approach is to obtain the tokens using the `split` method. Another approach is to retrieve the tokens using a Tokenizer, as covered in the Text Normalization Section.

The following code uses the `TextModel` class to tokenize the text using words as the tokenizer; the tokenized text is stored in the variable `D`.

```
tm = TextModel(token_list=[-1])
tok = tm.tokenize
D = [(tok(x), y) for x, y in D]
```

Before estimating the likelihood parameters, it is needed to encode the tokens using an index; by doing it, it is possible to store the parameters in an array and compute everything `numpy` operations. The following code encodes each token with a unique index; the mapping is in the dictionary `w2id`.

```
words = set()
[words.update(x) for x, y in D]
w2id = {v: k for k, v in enumerate(words)}
```

Previously, the classes have been represented using natural numbers. The positive class has been associated with the number 1, whereas the negative class with 0. However, in this dataset, the classes are strings. It was decided to encode them as numbers to facilitate subsequent

operations. The encoding process can be performed simultaneously with the estimation of the prior of each class. Please note that the priors are stored using the logarithm in the variable `priors`.

```
uniq_labels, priors = np.unique([k for _, k in D], return_counts=True)
priors = np.log(priors / priors.sum())
uniq_labels = {str(v): k for k, v in enumerate(uniq_labels)}
```

It is time to estimate the likelihood parameters for each of the classes. It is assumed that the data comes from a Categorical distribution and that each token is independent. The likelihood parameters can be stored in a matrix (variable `l_tokens`) with K rows, each row contains the parameters of the class, and the number of columns corresponds to the vocabulary's size. The first step is to calculate the frequency of each token per class which can be done with the following code.

```
l_tokens = np.zeros((len(uniq_labels), len(w2id)))
for x, y in D:
    w = l_tokens[uniq_labels[y]]
    cnt = Counter(x)
    for i, v in cnt.items():
        w[w2id[i]] += v
```

The next step is to normalize the frequency. However, before normalizing it, it is being used a Laplace smoothing with a value 0.1. Therefore, the constant 0.1 is added to all the matrix elements. The next step is to normalize (second line), and finally, the parameters are stored using the logarithm.

```
l_tokens += 0.1
l_tokens = l_tokens / np.atleast_2d(l_tokens.sum(axis=1)).T
l_tokens = np.log(l_tokens)
```

4.2.2.1 Prediction

Once all the parameters have been estimated, it is time to use the model to classify any text. The following function computes the posterior distribution. The first step is to tokenize the text (second line) and compute the frequency of each token in the text. The frequency stored in the dictionary `cnt` is converted into the vector `x` using the mapping function `w2id`. The final step is to compute the product of the likelihood and the prior. The product is computed in log-space; thus, this is done using the likelihood and the prior sum. The last step is to compute the evidence and normalize the result; the evidence is computed with the function `logsumexp`.

```
def posterior(txt):
    x = np.zeros(len(w2id))
    cnt = Counter(tm.tokenize(txt))
    for i, v in cnt.items():
        try:
            x[w2id[i]] += v
        except KeyError:
            continue
    _ = (x * l_tokens).sum(axis=1) + priors
    l = np.exp(_ - logsumexp(_))
    return l
```

The posterior function can predict all the text in $\mathcal{D}$; the predictions are used to compute the model's accuracy. In order to compute the accuracy, the classes in $\mathcal{D}$ need to be transformed using the nomenclature of the likelihood matrix and priors vector; this is done with the `uniq_labels` dictionary (second line).

```
hy = np.array([posterior(x).argmax() for x, _ in D])
y = np.array([uniq_labels[y] for _, y in D])
(y == hy).mean()
0.974
```

4.2.2.2 Training

Solving supervised learning problems requires two phases; one is the training phase, and the other is the prediction. The posterior function handles the later phase, and it is missing to organize the code described in a training function. The following code describes the training function; it requires the dataset's parameters and an instance of `TextModel`.

```
def training(D, tm):
    tok = tm.tokenize
    D =[(tok(x), y) for x, y in D]
    words = set()
    [words.update(x) for x, y in D]
    w2id = {v: k for k, v in enumerate(words)}
    uniq_labels, priors = np.unique([k for _, k in D], return_counts=True)
    priors = np.log(priors / priors.sum())
    uniq_labels = {str(v): k for k, v in enumerate(uniq_labels)}
    l_tokens = np.zeros((len(uniq_labels), len(w2id)))
    for x, y in D:
        w = l_tokens[uniq_labels[y]]
```

```
        cnt = Counter(x)
        for i, v in cnt.items():
            w[w2id[i]] += v
    l_tokens += 0.1
    l_tokens = l_tokens / np.atleast_2d(l_tokens.sum(axis=1)).T
    l_tokens = np.log(l_tokens)
    return w2id, uniq_labels, l_tokens, priors
```

4.3 Modelado Vectorial

xxx

5 Representación de Texto

El **objetivo** de la unidad es

Paquetes usados

```
from EvoMSA import BoW,\
                   DenseBoW
from microtc.utils import tweet_iterator
from wordcloud import WordCloud
import numpy as np
import pandas as pd
from matplotlib import pylab as plt
import seaborn as sns
```

5.1 Introducción

5.2 Bolsa de Palabras Dispersa

La idea de una bolsa de palabras discretas es que después de haber normalizado y segmentado el texto (Capítulo 2), cada token t sea asociado a un vector único $\mathbf{v_t} \in \mathbb{R}^d$ donde la i-ésima componente, i.e., $\mathbf{v}_{\mathbf{t}_i}$, es diferente de cero y $\forall_{j \neq i} \mathbf{v}_{\mathbf{t}_j} = 0$. Es decir la i-ésima componente está asociada al token t, se podría pensar que si el vocabulario está ordenado de alguna manera, entonces el token t está en la posición i. Por otro lado el valor que contiene la componente se usa para representar alguna característica del token.

El conjunto de vectores $\mathbf{v}$ corresponde al vocabulario, teniendo d diferentes token en el mismo y por definición $\forall_{i \neq j} \mathbf{v_i} \cdot \mathbf{v_j} = 0$, donde $\mathbf{v_i} \in \mathbb{R}^d$, $\mathbf{v_j} \in \mathbb{R}^d$, y $(\cdot)$ es el producto punto. Cabe mencionar que cualquier token fuera del vocabulario es descartado.

Usando esta notación, un texto x está representado por una secuencia de términos, i.e., $(t_1, t_2, ...)$; la secuencia puede tener repeticiones es decir, $t_j = t_k$. Utilizando la característica de que cada token está asociado a un vector $\mathbf{v}$, se transforma la secuencia de términos a una

secuencia de vectores (manteniendo las repeticiones), i.e., $(\mathbf{v_{t_1}}, \mathbf{v_{t_2}}, ...)$. Finalmente, el texto x se representa como:

$$\mathbf{x} = \frac{\sum_t \mathbf{v_t}}{\|\sum_t \mathbf{v_t}\|}, \tag{5.1}$$

donde la suma se hace para todos los elementos de la secuencia, $\mathbf{x} \in \mathbb{R}^d$, y $\|\mathbf{w}\|$ es la norma Euclideana del vector $\mathbf{w}$.

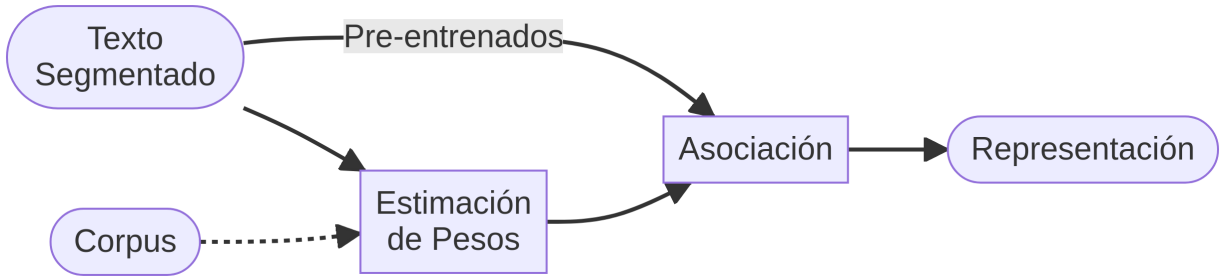

Figura 5.1: Diagrama Bolsa de Palabras Dispersa

Antes de iniciar la descripción detallada del proceso de representación utilizando una bolsa de palabras dispersas, es conveniente ilustrar este proceso mediante la Figura 5.1. El **texto segmentado** es el resultado del proceso ilustrado en Figura 2.1. El texto segmentado puede seguir dos caminos, en la parte superior se encuentra el caso cuando los pesos han sido identificados previamente y en la parte inferior es el procedimiento cuando los pesos se estiman mediante un corpus específico que normalmente es un conjunto de entrenamiento.

5.2.1 Pesado de Términos

Como se había mencionado el valor que tiene la componente i-ésima del vector $\mathbf{v_{t_i}}$ corresponde a una característica del término asociado, este procedimiento se le conoce como el **esquema de pesado**. Por ejemplo, si el valor es 1 (i.e., $\mathbf{v_{t_i}} = 1$) entonces el valor está indicando solo la presencia del término, este es el caso más simple. Considerando la Ecuación 5.1 se observa que el resultado, $\mathbf{x}$, cuenta las repeticiones de cada término, por esta característica a este esquema se le conoce como **frecuencia de términos** (*term frequency (TF)*).

Una manera de pesar los términos que ha sido muy efectiva en recuperación de información es considerar además la frecuencia de los términos en los diferentes documentos. En particular, el método conocido como **TFIDF** descrito por Salton y Yang (1973) propone considerar el producto de la frecuencia del término y el inverso de la frecuencia del término (*Inverse Document Frequency (IDF)*) en la colección.

5.2.2 Ejemplos

```
tm = BoW(lang='es').bow
vec = tm['Buen día']
vec[:3]
```

```
[(11219, 0.3984336285263178),
 (11018, 0.3245843730253675),
 (24409, 0.2377856890280623)]
```

```
[(tm.id2token[k], v)
 for k, v in vec[:3]]
```

```
[('buen~dia', 0.3984336285263178),
 ('buen', 0.3245843730253675),
 ('dia', 0.2377856890280623)]
```

```
txt = 'Buen día colegas'
[(tm.id2token[k], v)
 for k, v in tm[txt][:4]]
```

```
[('buen~dia', 0.24862785236357487),
 ('buen', 0.20254494048246244),
 ('dia', 0.1483814139998851),
 ('colegas', 0.3538047214393573)]
```

Se puede observar como los valores de IDF de los términos comunes cambiaron, por ejemplo para el caso de *buen~dia* cambio de 0.3984 a 0.2486. Este es el resultado de que los valores están normalizados tal como se muestra en la Ecuación 5.1.

La Figura 5.2 muestra la nube de palabras generada con los términos y sus respectivos valores IDF del texto *Es un placer estar platicando con ustedes.*

```
txt1 = 'Es un placer estar platicando con ustedes.'
txt2 = 'La lluvia genera un caos en la ciudad'
vec1 = tm[txt1]
vec2 = tm[txt2]
f = {k: v for k, v in vec1}
np.sum([f[k] * v for k, v in vec2 if k in f])
```

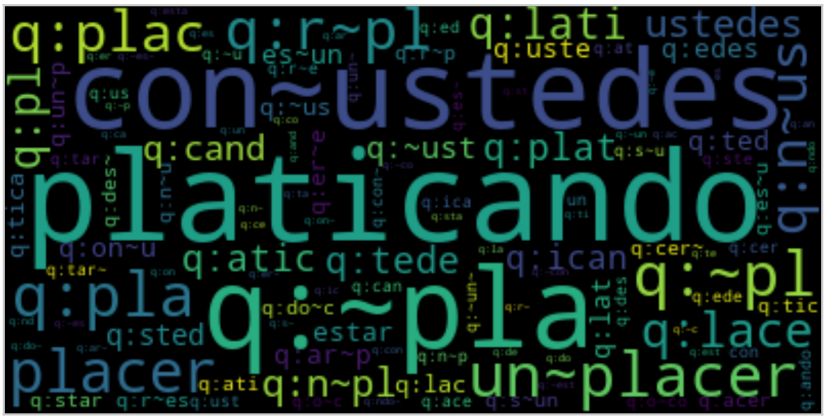

Figura 5.2: Nube de términos

```
0.01645519294478695
```

```
txt1 = 'Es un placer estar platicando con ustedes.'
txt2 = 'Estoy dando una platica en Morelia.'
vec1 = tm[txt1]
vec2 = tm[txt2]
f = {k: v for k, v in vec1}
np.sum([f[k] * v for k, v in vec2 if k in f])
```

```
0.2035427118119315
```

```
fname = 'delitos/delitos_ingeotec_Es_train.json'
D = list(tweet_iterator(fname))
```

```
X = tm.transform(D)
dis = np.dot(X, X.T)
```

5.3 Bolsa de Palabras Densa

5.4 Ejemplos

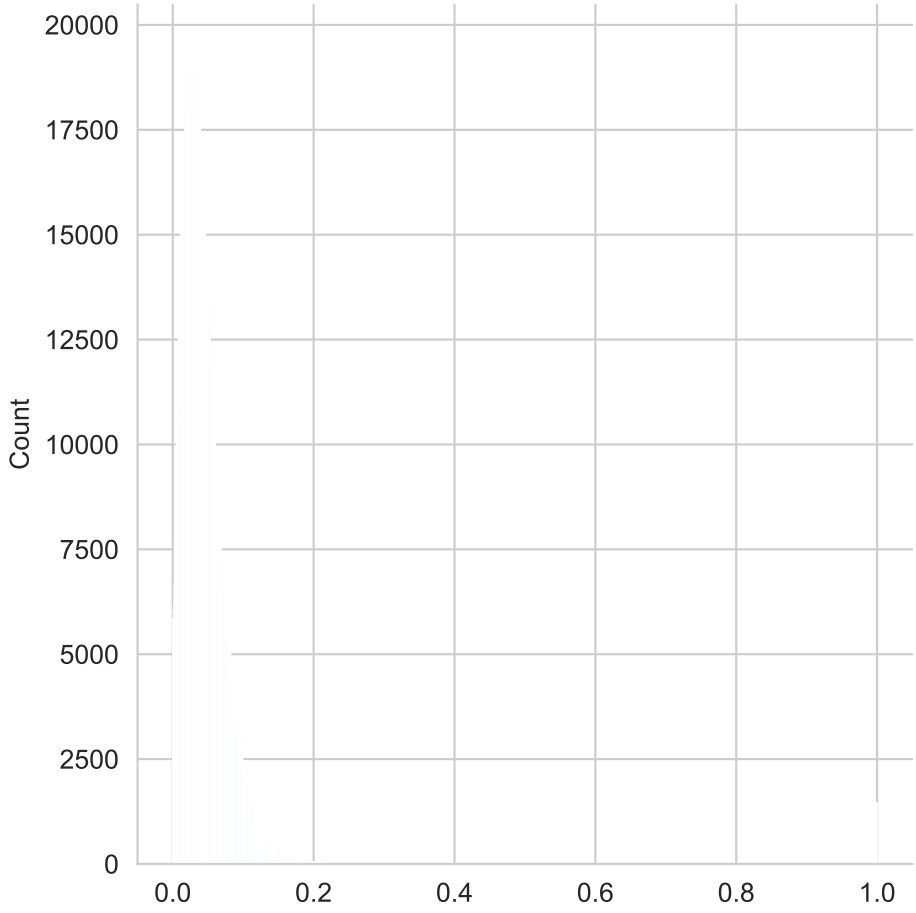

Figura 5.3: Histograma de la similitud

Figura 5.4: Diagrama Bolsa de Palabras Densa

```
dense = DenseBoW(lang='es',
                 voc_size_exponent=15,
                 emoji=True, keyword=True,
                 dataset=False)
txt1 = 'Es un placer estar platicando con ustedes.'
txt2 = 'La lluvia genera un caos en la ciudad.'
txt3 = 'Estoy dando una platica en Morelia.'
X = dense.transform([txt1, txt2, txt3])
np.dot(X[0], X[1]), np.dot(X[0], X[2])
```

```
(0.65677629690981, 0.8046113952647478)
```

```
X = dense.transform(D)
dis = np.dot(X, X.T)
```

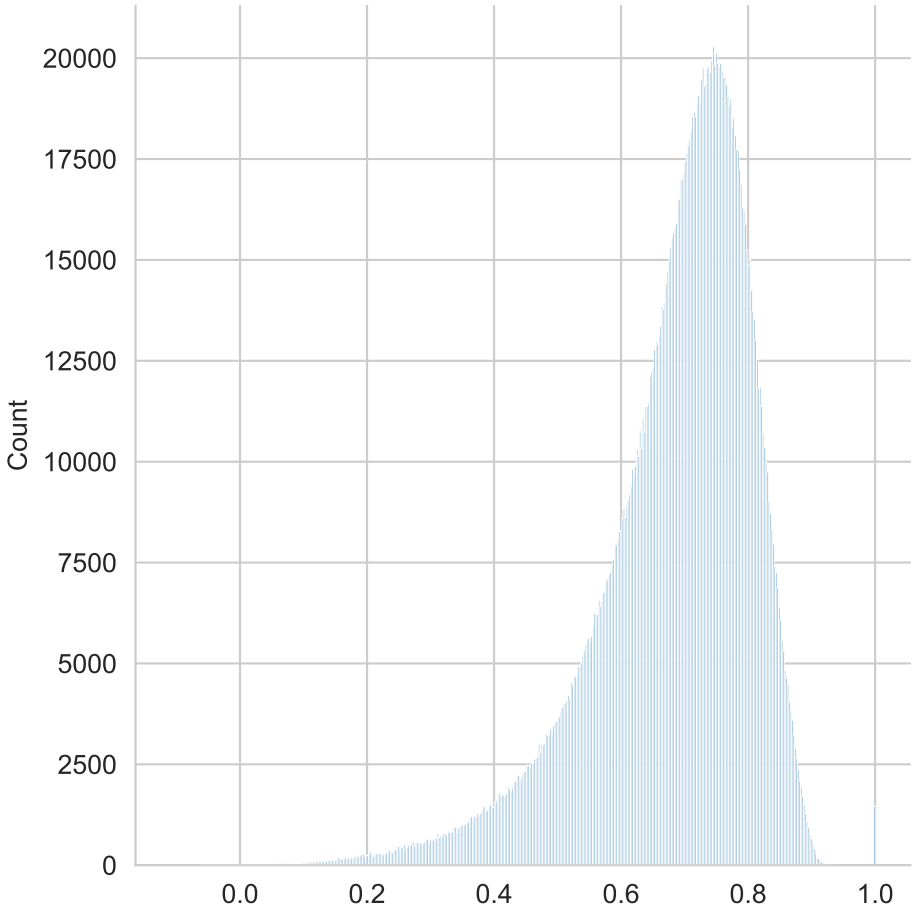

Figura 5.5: Histograma de la similitud usando bolsa de palabras densas

6 Mezcla de Modelos

El **objetivo** de la unidad es

Paquetes usados

```
from EvoMSA import BoW,\
                   DenseBoW,\
                   StackGeneralization
from microtc.utils import tweet_iterator
from IngeoML import CI, SelectFromModelCV
from sklearn.metrics import f1_score,\
                            recall_score,\
                            precision_score
from wordcloud import WordCloud
import numpy as np
import pandas as pd
from matplotlib import pylab as plt
import seaborn as sns
```

6.1 Introducción

El conjunto de datos se puede conseguir en la página de Delitos aunque en esta dirección es necesario poblar los textos dado que solamente se encuentra el identificador del Tweet.

Para leer los datos del conjunto de entrenamiento y prueba se utilizan las siguientes instrucciones. En la variable `D` se tiene los datos que se utilizarán para entrenar el clasificador basado en la bolsa de palabras y en `Dtest` los datos del conjunto de prueba, que son usados para medir el rendimiento del clasificador.

```
fname = 'delitos/delitos_ingeotec_Es_train.json'
fname_test = 'delitos/delitos_ingeotec_Es_test.json'
D = list(tweet_iterator(fname))
Dtest = list(tweet_iterator(fname_test))
```

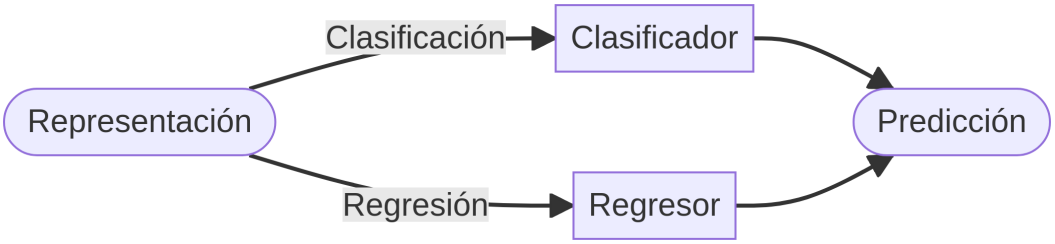

Figura 6.1: Diagrama de predicción

En la siguiente instrucción se observa el primer elemento del conjunto de entrenamiento. Se puede observar que en el campo `text` se encuentra el texto, el campo `klass` representa la etiqueta o clase, donde 0 representa la clase negativa y 1 la clase positiva, es decir, la presencia de un delito. El campo `id` es el identificador del Tweet y `annotations` son las clases dadas por los etiquetadores a ese ejemplo.

```
D[81]
```

```
{'annotations': [0, 0, 0],
 'id': 1107040319986696195,
 'klass': 0,
 'text': 'To loco'}
```

6.2 Bolsa de Palabras Dispersa

Se inicia con la creación de un clasificador basado en una bolsa de palabras dispersa, el clasificador es una máquina de soporte vectorial lineal (`LinearSVC`). La siguiente instrucción usa la clase `BoW` para crear este clasificador de texto. El primer paso es seleccionar el lenguaje, en este caso español (es) y después se entrena usando el método `fit`.

```
bow = BoW(lang='es').fit(D)
```

Habiendo entrenado el clasificador de texto es momento de utilizarlo para predecir, las siguientes dos instrucciones muestra el uso de la instancia `bow` para predecir clase del texto *me golpearon y robaron la bicicleta en la noche.* Se puede observar que la clase es 1, lo cual indica que el texto menciona la ejecución de un delito.

```
txt = 'me golpearon y robaron la bicicleta en la noche'
bow.predict([txt])
```

```
array([1])
```

El método `predict` recibe una lista de textos a predecir, en la siguiente instrucción se predicen todas las clases del conjunto de prueba (`Dtest`), la predicciones se guardar en la variable `hy_bow`.

```
hy_bow = bow.predict(Dtest)
```

Habiendo realizado la predicciones en el conjunto de prueba ($\mathcal{D}$), es momento de utilizar estas para medir el rendimiento, en esta ocasión se mide el valor f_1 para cada clase. El primer valor (0.9461) corresponde a la medida f_1 en la clase negativa y el segundo (0.7460) corresponde al valor en la clase positiva.

```
y = np.r_[[x['klass'] for x in Dtest]]
f1_score(y, hy_bow, average=None)
```

```
array([0.94612795, 0.74603175])
```

Con el objetivo de conocer la variabilidad del rendimiento del clasificador en este conjunto de datos, la siguientes instrucciones calcula el intervalo de confianza; para realizarlo se utiliza la clase `CI` la cual recibe la estadística a calcular, en este caso medida f_1. El siguiente paso es llamar a la clase con las entradas para calcular el intervalo, estas corresponden a las mediciones y predicciones del conjunto de prueba.

```
ci = CI(statistic=lambda y, hy: f1_score(y, hy,
                                         average=None))
ci_izq, ci_der = ci(y, hy_bow)
```

El intervalo izquierdo es $[0.9279, 0.6549]$ y el derecho tiene los valores $[0.9635, 0.8254]$. Para complementar la información del intervalo de confianza, la Figura 6.2 muestra el histograma y la densidad estimada para calcular el intervalo de confianza. Se ve que la varianza en la clase negativa es menor además de que tiene un rendimiento mejor que en la clase positiva.

Una manera de poder analizar el comportamiento del clasificador de texto implementado es visualizar en una nube de palabras las características que tienen el mayor peso en la decisión. Esto se realiza en las siguientes instrucciones siendo el primer paso obtener los coeficientes de la máquina de soporte vectorial lineal, los cuales se guardan en la variable `ws`. El segundo componente es el valor de IDF que tiene cada uno de los términos, esto se encuentran en el atributo `BoW.weights` tal y como se muestra en la segunda instrucción del siguiente código.

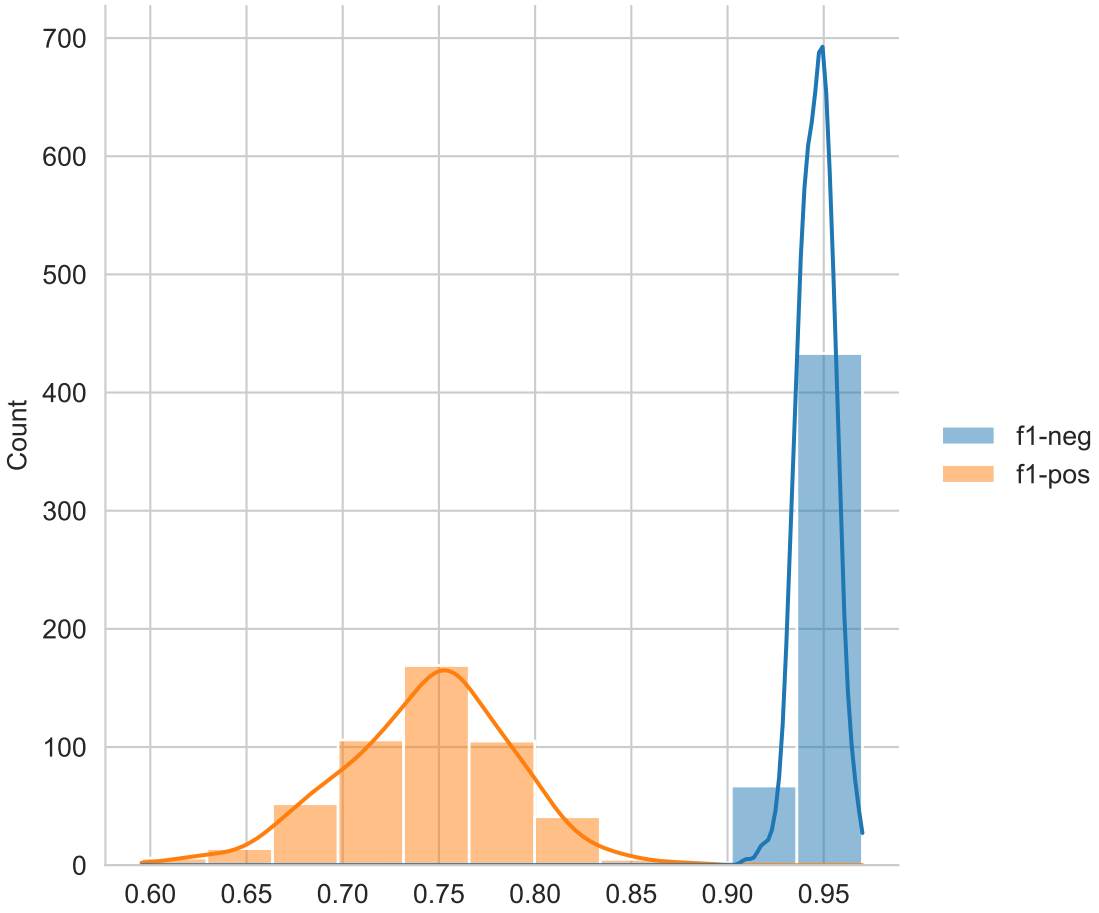

Figura 6.2: Histograma de f1 por clase

```
ws = bow.estimator_instance.coef_[0]
idfs = bow.weights
```

Teniendo los valores del clasificador y del IDF, solamente es necesario obtener su producto y separar los términos positivos de los negativos, tal y como se muestra en el siguiente código. La Figura 6.3 muestra las nubes de palabras generadas, cabe mencionar que aquellos términos que tienen como prefijo `q:` corresponden a q-gramas de caracteres y los términos que tienen el caracter `~` corresponden a bigramas de palabras.

```
tokens_pos = {name: w * idf
              for name, idf, w in zip(bow.names,
                                      idfs, ws)
              if w > 0}
tokens_neg = {name: w * idf * -1
              for name, idf, w in zip(bow.names,
                                      idfs, ws)
              if w < 0}
```

En la figura se puede observar que las características más importantes corresponden a la presencia de *asesinan*, *asesinan a*, *tiroteo* entre otras y por el lado de palabras relacionadas a la clase negativa se observan *secuestrado*, *asesina*, *un asesino*, entre otras. En Sección 6.4 se muestran ejemplos que ayudan a comprender el hecho de que la palabra *asesinan* sea considerada como positiva y por otro lado *asesina* se en la clase negativa.

Figura 6.3: Nubes de términos positivos y negativos

6.3 Bolsa de Palabras Densas

Es ahora el turno de hacer el análisis del clasificador de texto que se basado en bolsa de palabras densas. La siguiente instrucción muestra el uso de este clasificador donde se usa una bolsa de palabras con un vocabulario de 2^{15} (parámetro `voc_size_exponent=15`) y además se seleccionan como representaciones aquellos entrenados en los conjuntos de emojis (`emoji=True`)

y las palabras claves (`keyword=True`). En esta caso, los parámetros del clasificador no son estimados, es decir, no se llama al método `fit`. Esto es porque en este ejemplo se van a seleccionar aquellas representaciones que mejor representan al problema de Delitos utilizando una máquina de soporte vectorial lineal.

```
dense = DenseBoW(lang='es',
                 voc_size_exponent=15,
                 emoji=True, keyword=True,
                 dataset=False)
```

Para seleccionar las características que mejor representan al problema de delitos se utiliza la clase `SelectFromModelCV` la cual usa los coeficientes de la máquina de soporte vectorial para seleccionar las características más representativas, estas corresponden aquellas que tienen los coeficientes más grandes tomando su valor absoluto. La selección se realiza llamando al método `DenseBoW.select` con los parámetros que se observan en las siguientes instrucciones. En particular `SelectFromModelCV` es un método supervisado entonces se utilizarán las clase del conjunto de entrenamiento, y para poder medir el rendimiento de cada conjunto de características seleccionadas se usa una validación cruzada. La última instrucción estima los valores del clasificador con las características seleccionadas.

```
macro_f1 = lambda y, hy: f1_score(y, hy, average='macro')
kwargs = dense.estimator_kwargs
estimator = dense.estimator_class(**kwargs)
kwargs = dict(estimator=estimator,
              scoring=macro_f1)
dense.select(D=D,
             feature_selection=SelectFromModelCV,
             feature_selection_kwargs=kwargs)
dense.fit(D)
```

Como se mencionó la clase `SelectFromModelCV` selecciona aquellas características que mejor rendimiento dan, la clase mantiene los valores estimados en cada selección, las siguientes instrucciones ejemplifican como obtener los valores de rendimiento en las selecciones. La variable `perf` es una diccionario donde la llave es el número de características y el valor es el rendimiento correspondiente. La Figura 6.4 muestra es rendimiento se puede observar la dinámica donde con un poco menos de 1000 características se tiene un valor de rendimiento cercano a 0.9.

```
select = dense.feature_selection
perf = select.cv_results_
```

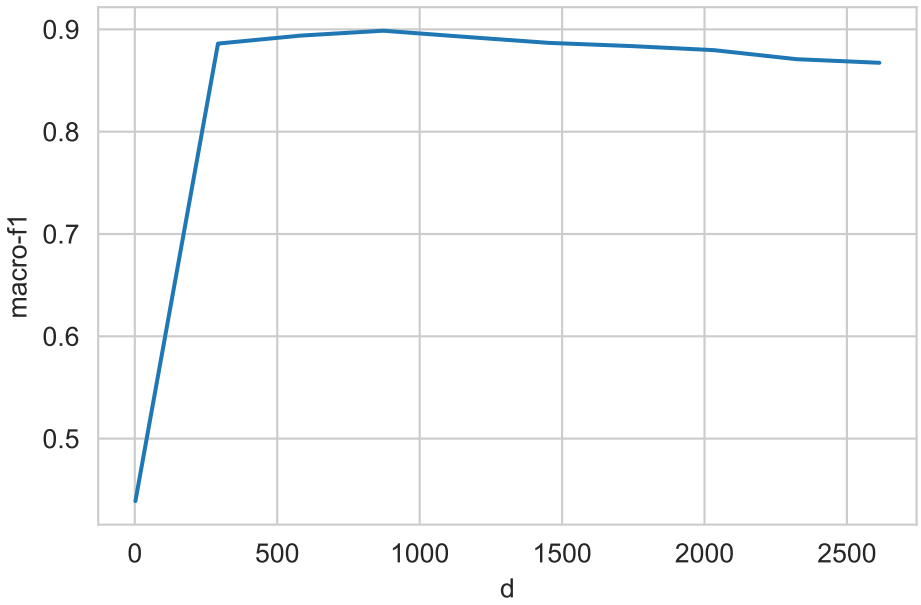

Figura 6.4: Rendimiento Variando el Número de Características

Después de haber seleccionado el número de características, se utiliza un código equivalente al usado en `BoW` para predecir las clases del conjunto de prueba ($\mathcal{G}$), tal y como se muestra en la siguiente instrucción.

```
hy_dense = dense.predict(Dtest)
```

El rendimiento en f_1 del clasificador basado en una bolsa se muestra con el siguiente código. Este valor puntual se complementa con la Figura 6.5 donde se muestra la distribución de esta medición y se compara con la obtenida con el clasificador de bolsa de palabras dispersa (i.e., `bow`).

```
f1_score(y, hy_dense, average=None)
```

```
array([0.94158076, 0.75362319])
```

En un clasificador basado en palabras densas también se puede comprender su comportamiento mostrando aquellas características que tiene un mayor peso al momento de decidir la clase. En las siguientes instrucciones se agrupan las características positivas y las negativas, utilizando el valor estimado por la máquina de soporte vectorial lineal (`w`). Considerando que cada característica está asociada a una palabra o emoji, entonces se pueden visualizar mediante una nube de palabras.

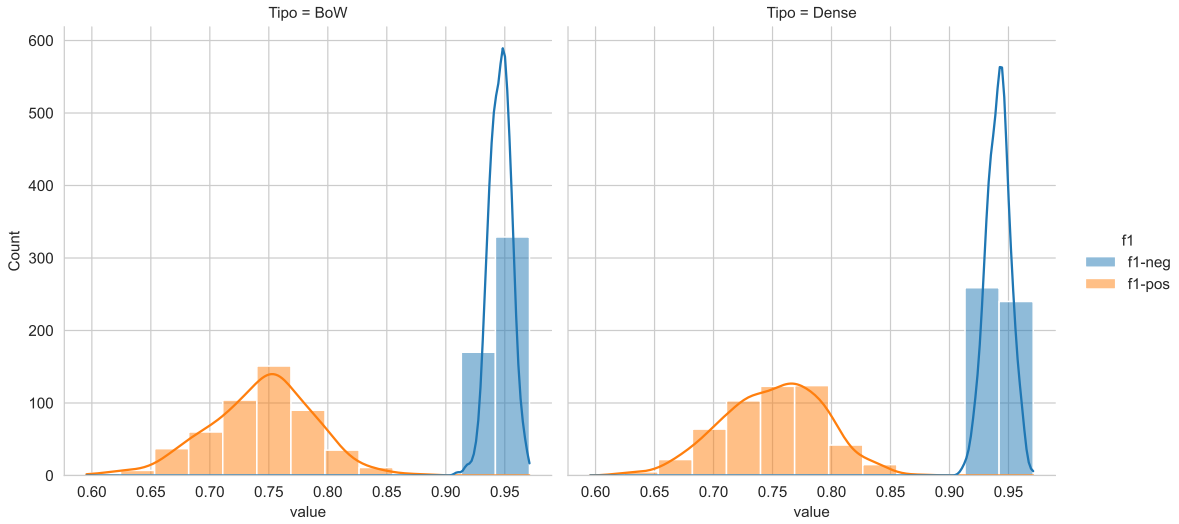

Figura 6.5: Histogramas de f1 por clase

```
w = dense.estimator_instance.coef_[0]
names = np.array(dense.names)
carac_pos = {k: v for k, v in zip(names, w) if v > 0}
carac_neg = {k: v * -1 for k, v in zip(names, w) if v < 0}
```

La Figura 6.6 muestra las nubes de palabras de las características positivas y negativas, se puede observar que una característica significativa de la clase positiva corresponde al modelo **robo**, **muere**, entre otros y de la clase negativa se observa **comentando**, **ocurrir*, entre otras.

Figura 6.6: Nube de características positivas y negativas

6.4 Análisis Mediante Ejemplos

Hasta el momento se ha presentado un análisis global de los clasificadores dispersos (`bow`) y densos (`dense`), en esta sección se especializa el análisis al nivel de ejemplos. Lo primero que se realiza es ver el valor de la función de decisión, el signo de este valor indica la clase, un valor positivo indica clase positiva y el signo negativo corresponde a la clase negativa. El valor absoluto de la función indica de manera proporcional la distancia que existe al hiperplano que define las clases. Dado que se está utilizando este valor para contrastar el comportamiento de los algoritmos entonces la distancia entre el ejemplo al hiperplano está dado por la función de decisión dividida entre la norma de los coeficientes. La primera línea calcula la norma de los coeficientes estimados tanto para el clasificador disperso (`bow_norm`) y el denso (`dense_norm`)

```
bow_norm = np.linalg.norm(bow.estimator_instance.coef_[0])
dense_norm = np.linalg.norm(dense.estimator_instance.coef_[0])
```

Con las normas se procederá a calcular la función de decisión para el ejemplo *Asesinan a persona en Jalisco*, este ejemplo es positivo dado que menciona la ocurrencia de un delito. En las siguientes instrucciones se calcula la distancia al hiperplano la cual se puede observar que es positiva indicando que el texto es positivo.

```
array([[0.03104446]])
```

Complementando la distancia del clasificador disperso se presenta la distancia del clasificador denso en el siguiente código. También se puede observar que su valor es positivo, pero este se encuentre más cercano al hiperplano de decisión, lo cual indica que existe una mayor incertidumbre en su clase.

```
array([[0.00906055]])
```

Realizando el mismo procedimiento pero para texto *La asesina vivía en Jalisco.* Lo primero que se debe de notar es que el texto es negativo dado que se menciona que existe una asesina, pero el texto no indica que se haya cometido algún delito, esta fue una de las reglas que se siguió para etiquetar los textos tanto del conjunto de entrenamiento ($\mathcal{T}$) como del conjunto de prueba ($\mathcal{G}$). Pero es evidente que el texto anterior y el actual son sintácticamente muy similares, pero con una semántica diferente.

El siguiente código predice la función de decisión del clasificador disperso, la distancia es el valor absoluto del número presentado y el signo indica el lado del hiperplano, se observa que es negativo, entonces el clasificador indica que pertenece a la clase negativa.

```
array([[-0.03643009]])
```

El mismo procedimiento se realiza para el clasificador denso como se indica a continuación, obteniendo también un valor negativo y con una magnitud similar al encontrado por el clasificador disperso.

```
array([[-0.03598119]])
```

Continuando con el análisis, se puede visualizar los coeficientes más significativos para realizar la predicción. Por ejemplo, las siguientes instrucciones muestran los 5 coeficientes más significativos para predecir el texto *Asesinan a persona en Jalisco.*

```
[('asesinan', 0.21959413198255906),
 ('asesinan~a', 0.20828285239067854),
 ('q:sina', 0.14511792223092249),
 ('q:n~a~', 0.0838802790777187),
 ('q:an~a', 0.07344372474399327)]
```

Un procedimiento equivalente se realiza para el clasificador denso, tal y como se muestra en el siguiente código.

```
[('ocurrir', 0.06683457750173856),
 ('muere', 0.053880044741064774),
 ('consiguio', -0.05168798790090579),
 ('critican', -0.04459207389659752),
 ('hubieses', -0.04426955144485894)]
```

La Figura 6.7 muestra la nube de palabras de los términos y características más significativas para la predicción del ejemplo positivo (*Asesinan a persona en Jalisco*). La nube de palabras para el ejemplo negativo (*La asesina vivía en Jalisco*) se muestra en la Figura 6.8. La nube de palabras está codificada de la siguiente manera, las palabras que corresponden a la clase positiva está en mayúsculas y las de la clase negativa en minúsculas. Por ejemplo, en Figura 6.7 se observa que la palabra *asesinan* es relevante para la clasificación del ejemplo así como la característica **ocurrir**.

En el caso del ejemplo negativo, la Figura 6.8 muestra q-gramas de caracteres asociados a la clase positiva y también es evidente la palabra *asesina*. En el caso del clasificador denso también se observan características positivas como **ocurrir** y características negativas como **critican** y **empleados**.

Complementando los ejemplos anteriores, la Figura 6.9 muestra la nube de palabras obtenidas al calcular la función de decisión del texto *Le acaban de robar la bicicleta a mi hijo.* Se observa que este texto corresponde a la clase positiva y la función de decisión normalizada del clasificador disperso es -0.0335 y del clasificador denso corresponde a -0.0798. Ambas

Figura 6.7: Nube de características ejemplo positivo

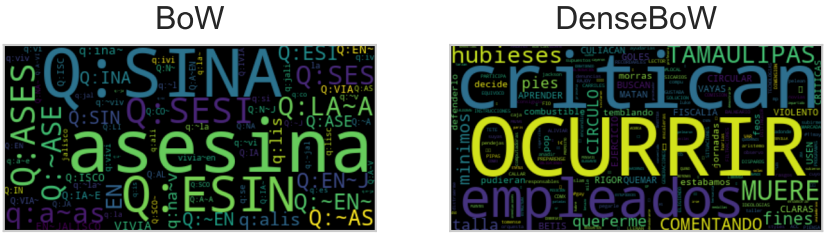

Figura 6.8: Nube de características ejemplo negativo

funciones de decisión indican que la clase es negativa, lo cual es un error. La figura muestra que los q-gramas de caracteres y las características positivas dominan las nubes, pero estas no tienen el peso suficiente para realizar una predicción correcta.

Figura 6.9: Nube de características en un ejemplo positivo con predicción negativa

6.5 Combinando Modelos

La siguiente pregunta es conocer si los modelos anteriores se pueden combinar para realizar una mejor predicción. En esta sección se utiliza la técnica de Stack Generalization (Wolpert (1992), Graff et al. (2020)) para combinar los dos modelos. La siguiente linea entrena el clasificador, el cual recibe como parámetros los clasificador a juntar.

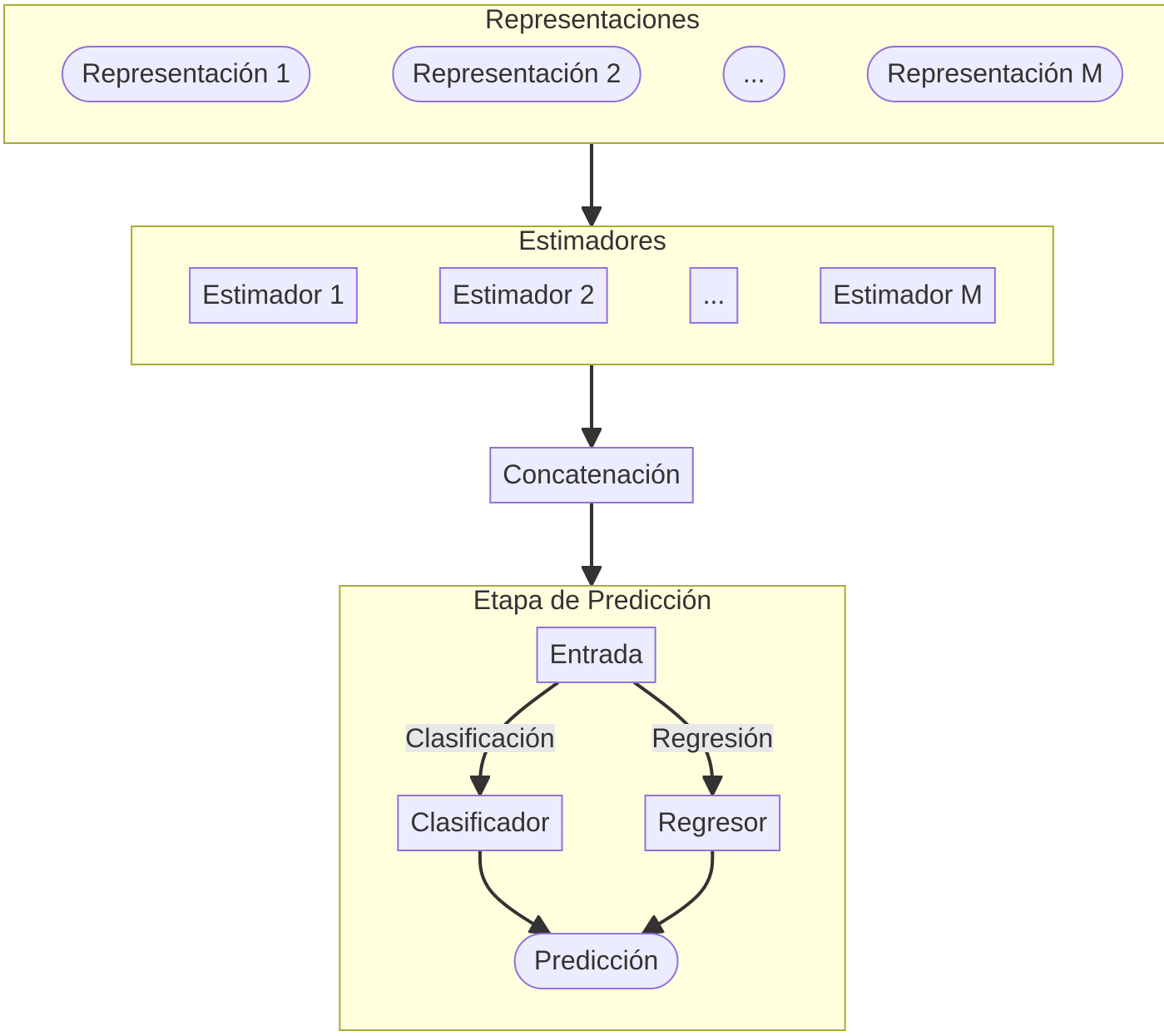

Figura 6.10: Diagrama de predicción en mezcla de modelos

```
stack = StackGeneralization([bow, dense]).fit(D)
```

Siguiendo el procedimiento de los clasificadores dispersos y densos, la siguiente linea predice la clase de los ejemplos del conjunto de prueba y calcula su rendimiento en términos de la medida f_1.

```
hy_stack = stack.predict(Dtest)
f1_score(y, hy_stack, average=None)
```

```
array([0.94791667, 0.79166667])
```

Para poder comparar el rendimiento de los tres clasificadores desarrollados, la Tabla 6.1 presenta el rendimiento con las medidas recall y precision en las dos clases. Se puede observar que el `bow` tiene el mejor recall en la clase negativa y mejor precision en la clase positiva. Por otro lado el mejor recall en la clase positiva y precision en la clase negativa lo tiene `stack`.

Tabla 6.1: Rendimiento

	Recall neg	Recall pos	Precision neg	Precision pos
`bow`	0.9894	0.6184	0.9065	0.9400
`dense`	0.9648	0.6842	0.9195	0.8387
`stack`	0.9613	0.7500	0.9349	0.8382

Con respecto al rendimiento en términos de f_1, la Tabla 6.2 presenta la información con respecto a cada clase y la última columna contiene el macro-f_1. Los valores indican que en la clase positiva el mejor valor corresponde a `stack` lo cual se ve reflejado en el macro-f_1. El algoritmo de Stack Generalization nos indica que se hizo una mejora en la predicción de la clase positiva y la clase negativa se mantuvo constante al menos con respecto de la medida f_1.

Tabla 6.2: Rendimiento

	f1 neg	f1 pos	macro-f1
`bow`	0.9461	0.7460	0.8461
`dense`	0.9416	0.7536	0.8476
`stack`	0.9479	0.7917	0.8698

7 Tareas de Clasificación de Texto

El **objetivo** de la unidad es

8 Bases de Conocimiento

El **objetivo** de la unidad es

9 Visualización

El **objetivo** de la unidad es

10 Conclusiones

El **objetivo** de la unidad es

Referencias

Graff, Mario, Sabino Miranda-Jiménez, Eric S. Tellez, y Daniela Moctezuma. 2020. «EvoMSA: A Multilingual Evolutionary Approach for Sentiment Analysis». *Computational Intelligence Magazine* 15: 76-88. https://ieeexplore.ieee.org/document/8956106.

Salton, Gerard, y Chungshu S. Yang. 1973. «On the specification of term values in automatic indexing». *Journal of Documentation* 29 (abril): 351-72. https://doi.org/10.1108/EB026562/FULL/XML.

Tellez, Eric S., Sabino Miranda-Jiménez, Mario Graff, Daniela Moctezuma, Oscar S. Siordia, y Elio A. Villaseñor. 2017. «A case study of Spanish text transformations for twitter sentiment analysis». *Expert Systems with Applications* 81: 457-71. https://doi.org/https://doi.org/10.1016/j.eswa.2017.03.071.

Tellez, Eric S., Sabino Miranda-Jiménez, Mario Graff, Daniela Moctezuma, Ranyart R. Suárez, y Oscar S. Siordia. 2017. «A Simple Approach to Multilingual Polarity Classification in Twitter». *Pattern Recognition Letters*. https://doi.org/10.1016/j.patrec.2017.05.024.

Tellez, Eric S., Daniela Moctezuma, Sabino Miranda-Jiménez, y Mario Graff. 2018. «An automated text categorization framework based on hyperparameter optimization». *Knowledge-Based Systems* 149: 110-23. https://doi.org/10.1016/j.knosys.2018.03.003.

Wolpert, David H. 1992. «Stacked generalization». *Neural Networks* 5 (2): 241-59. https://doi.org/10.1016/S0893-6080(05)80023-1.