



# Introducción a la Ciencia de Datos

## Edición 2024

Informe de Laboratorio 2

v1.0.0  
02 de julio 2024

Grupo 15  
Matias Rolando  
Emiliano Viotti

# Índice

<b>Índice.....</b>	<b>1</b>
<b>Introducción.....</b>	<b>2</b>
<b>1. Primera Parte: Dataset y Representación Numérica de Texto.....</b>	<b>3</b>
Preparación de los Datos.....	3
1.1. Train/Test Split.....	4
1.2. Balance de Párrafos por Personaje.....	4
1.3. Representación Bag-of-Words.....	5
Vectorización en Bag-of-Words.....	6
Sparsity y Complejidad Computacional.....	6
1.4. N-Gramas y TF-IDF.....	7
N-Grama.....	7
Representación TF-IDF.....	7
1.5. Reducción Dimensionalidad: PCA.....	8
¿Contienen información relevante los símbolos de puntuación?.....	11
¿Se pueden separar los personajes utilizando sólo 2 componentes principales?.....	12
Varianza Explicada en las N Componentes Principales.....	13
<b>2. Segunda Parte: Entrenamiento y Evaluación de Modelos.....</b>	<b>15</b>
2.1. Clasificador Multinomial Naive-Bayes.....	15
2.2. Validación Cruzada (Cross-Validation).....	17
2.3. Selección Modelo.....	18
2.4. Modelando con Otras Arquitecturas.....	19
SVM.....	19
2.5. Experimentando con Personajes Adicionales.....	20
2.6. Técnicas Alternativas de Representaciones de Texto.....	23
Matriz de Coocurrencias (Co-occurrences Matrices).....	23
Word2Vec.....	24
<b>3. Opcionales: Modelo Fasttext.....</b>	<b>26</b>
Aprendiendo Representaciones.....	26
Entrenando un Clasificador de Texto.....	29
Usando embeddings pre-entrenados.....	30
<b>4. Conclusiones.....</b>	<b>32</b>
<b>Referencias.....</b>	<b>33</b>

# Introducción

William Shakespeare, nació el 23 de abril de 1564 en Stratford-upon-Avon, Inglaterra y en lo que hoy en día parecería una vida corta (52 años), se transformó en una figura titánica del mundo de la literatura. Este dramaturgo y poeta inglés dejó un legado imborrable con sus más de 39 obras literarias, existen al menos dos corrientes que discuten incluso hoy en día la atribución de ciertas obras, entre entre las que se destacan sus tragedias y comedias, obras como "Hamlet", "Romeo y Julieta" y "El rey Lear". Ya sea si has leído alguna obra de William Shakespeare o no, es muy probable que reconozcas algunas frases con origen en su obra como "Ser o no ser, esa es la cuestión" o "El amor es un humo hecho con el vapor de suspiros". Estas líneas no solo demuestran su maestría lingüística, sino que también reflejan las intrigas universales sobre el amor, el poder y la tragedia, manteniendo su relevancia a través de los siglos.

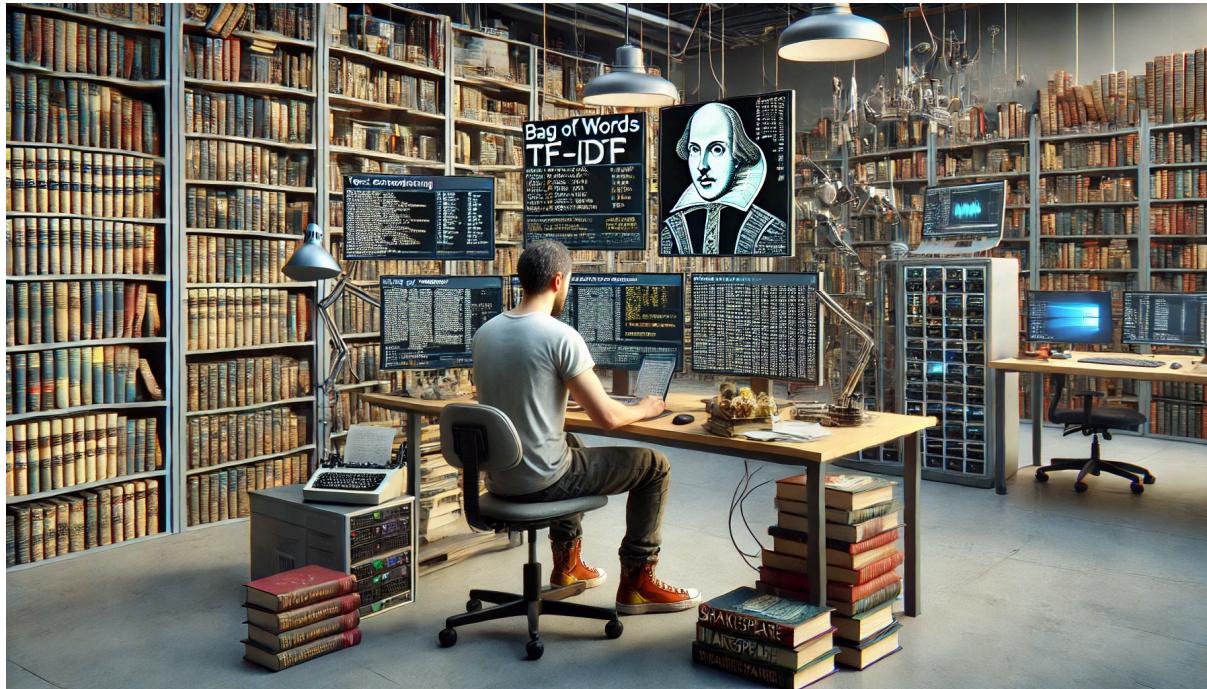
En este trabajo llevado adelante en el contexto del [segundo Laboratorio](#) del curso Introducción a la Ciencia de Datos de la Facultad de Ingeniería, UdelaR, edición 2024, nos proponemos continuar adentrándonos en la obra de William Shakespeare con un enfoque de Machine Learning, modelando sus principales obras y personajes utilizando técnicas de machine learning y procesamiento de lenguaje natural (NLP).

Antes de comenzar con este viaje algunas consideraciones que debes tener en cuenta:

- Esta es la segunda parte de un trabajo más completo, el cual inició con un análisis exploratorio de datos y un análisis con enfoque de ciencia de datos. Para ver los detalles de la primera parte puedes referirte a [Informe Laboratorio 1](#).
- Al presente informe lo acompaña un notebook con el código utilizado para generar los resultados expuestos en este trabajo. El mismo se encuentra disponible en [laboratorio 2.ipynb](#).
- Todos los recursos de este trabajo (notebooks, datasets, scripts, etc.) se encuentran disponibles en el siguiente [repositorio Git](#).

Finalmente, esperamos que disfrutes este viaje a través de los datos, el tiempo y principalmente, de la lengua inglesa, tanto como nosotros lo hemos disfrutado.

# 1. Primera Parte: Dataset y Representación Numérica de Texto



## Preparación de los Datos

El primer paso en este trabajo es la preparación de los datos, etapa que se centra casi y exclusivamente en la columna `PlainText` con la cual construiremos las features con las que entrenaremos diferentes modelos de clasificación. Esta columna contiene el texto plano de los párrafos de la obra de Shakespeare, los cuales podemos asociar a través de otras columnas al personaje al cual pertenece un párrafo.

Para obtener mejores resultados vamos a pre-procesar la columna `PlainText`, normalizando valores y convirtiendo a minúsculas. Además eliminaremos el ruido introducido por ejemplo por símbolos de puntuación.

Para esto utilizamos la función `clean_text` implementada en el Laboratorio 1 la cual realiza las siguientes transformaciones:

- Convierte el valor a minúsculas
- Remueve espacios en blanco al inicio y final
- Reemplaza símbolos de puntuación por un espacio en blanco “ ”

En particular vamos a reemplazar los siguientes símbolos de puntuación: `["[", "\n", ", ", ":" , ";" , ".," , "]"," , "(", ")" , "?", "!", "!", "-", "'", {"", "}" ]`

Por otro lado, combinamos los datos de todas las tablas (`Paragraphs`, `Chapters`, `Works` y `Characters`) en un único dataframe que facilite posteriormente el consolidado de un dataset de entrenamiento y test para párrafos de personajes.

## 1.1. Train/Test Split

El segundo paso en este trabajo y previo a entrenar cualquier modelo, es construir un conjunto de datos para entrenamiento y otro para test, a partir de los datos pre-procesados. Para esto utilizamos la función `train_test_split()` de scikit-learn. En particular, utilizamos el parámetro `stratify` sobre la columna `y` que nos permite mantener el mismo balance de datos entre clases tanto en la partición de entrenamiento como la partición de test. A su vez, fijamos el parámetro `random_state` para hacer reproducibles los resultados. Notar que como columna `y` o target vamos a definir la columna `CharName` que contiene el nombre del personaje al cual está asociado un párrafo.

A continuación se muestra a modo ilustrativo el código python utilizado para construir los datasets de entrenamiento y test. Por más información referirse al notebook que acompaña a este informe.

```
x_train, x_test, y_train, y_test = train_test_split(  
    X, y, test_size=0.3, stratify=y, random_state=RANDOM_STATE  
)
```

## 1.2. Balance de Párrafos por Personaje

Revisemos cómo quedó el balance de párrafos por personaje luego de separar los ejemplos en los dos conjuntos: train y test. En particular vamos a revisar no solo la cantidad de párrafos sino la proporción o porcentaje dentro del subconjunto correspondiente (train o test) para ver si el parámetro `stratify` ha sido efectivo en mantener las proporciones originales.

	character	count	partition	%
0	Antony	177	train	40.4
1	Cleopatra	143	train	32.6
2	Queen Margaret	118	train	26.9
3	Cleopatra	61	test	32.4
4	Queen Margaret	51	test	27.1
5	Antony	76	test	40.4

Como se puede ver tanto la tabla anterior como en la siguiente figura, la función `train_test_split` cuando se la utiliza con la opción `stratify`, genera dos particiones (train y test) manteniendo la proporción original de etiquetas.



Figura 1 - Proporción Párrafos Train/Test

### 1.3. Representación Bag-of-Words

Bag-of-Words (BoW) es una técnica de extracción de features para texto, relativamente simple, que se ha utilizado durante años en diferentes tareas de NLP. Actualmente, existen técnicas más eficientes y flexibles, que permiten capturar mayor información de un texto y en particular capturar información del contexto.

Esta técnica consiste primero en construir un vocabulario, en donde cada palabra única tiene un valor numérico asignado. Para esto, alcanza con tomar un conjunto con todas las palabras o tokens en los textos de entrenamiento y asignarles un identificador numérico, por ejemplo de  $1 \dots N$ , donde  $N$  representa el tamaño del vocabulario (cantidad de palabras o tokens distintos).

Para este proceso primero se separa el texto en palabras, formalmente hablando tokens. Esta tarea no es trivial ya que los tokens podrían estar separados por espacios en blanco " ", símbolos de puntuación como [",", ";", ".", ":" , "?", "!" ] o incluso, una misma palabra podría componerse por dos o más tokens. Este es el caso por ejemplo, del tokenizador utilizado en modelos como GPT-3.5, en donde utilizan [Byte-Pair Encoding](#).

Luego, se convierte cada ejemplo de entrenamiento en un vector de tamaño  $N$ , con  $N$  igual al tamaño de tokens en el vocabulario, donde la posición  $i$ -ésima representa la cantidad de ocurrencias del token  $i$  del vocabulario, en el texto a vectorizar.

Notar que el vocabulario se construye solamente a partir de los tokens de entrenamiento, por lo que una limitación de esta técnica es la incapacidad de modelar tokens nuevos presentes en el conjunto de test.

Por ejemplo si el vocabulario fuese el siguiente `vocab=['hola', 'mundo', 'feliz', '!', '2024']` podríamos tener las siguientes representaciones en vectores Bag-of-Words para los siguientes textos de entrenamiento:

Texto	Vector
hola mundo!	[1, 1, 0, 1, 0]
feliz 2024!!	[0, 0, 1, 2, 1]

## Vectorización en Bag-of-Words

Para convertir los ejemplos en vectores BoW utilizamos la función [CountVectorizer](#) de scikit-learn. En particular esta función tiene dos parámetros `stop_word` y `ngram_range`. El primero permite pasar una lista de stopwords para ignorar tokens de esta lista al momento de vectorizar los ejemplos. El segundo indica el rango de n-gramas a considerar. En particular comenzaremos generando los vectores sin stopwords e indicando unigramas ( $n=1$ ). Más adelante en este trabajo podremos experimentar con diferentes combinaciones de estos parámetros.

Sin stopwords y  $n=1$ , obtenemos un vocabulario compuesto por 2831 tokens.

## Sparsity y Complejidad Computacional

Partiendo de un vector `X_train` de tamaño  $(M,)$  donde cada fila es un `str` y asumiendo un vocabulario de tamaño  $N$  compuesto por los tokens únicos en los  $M$  ejemplos de entrenamiento, la matriz resultante luego de vectorizar los datos de entrenamiento queda de tamaño  $M \times N$ . Donde cada fila representa un ejemplo de entrenamiento. Notar que esta matriz es altamente sparse ya que un vocabulario puede tener miles de tokens, mientras que un párrafo puede contener solamente una fracción de ellos. Por ejemplo, solamente en los párrafos seleccionados de Shakespeare en `X_train`, se tienen un vocabulario de 2831 tokens.

Cabe destacar que bibliotecas como NumPy, almacenan de forma eficiente sparse matrix como esta, reduciendo la cantidad de memoria necesaria para almacenar esta información. [Working With Text Data](#), muestra cual es el costo de mantener en memoria un corpus de texto utilizando BoW, en base a NumPy arrays de tipo `float32`. En particular, el costo crece por cada nuevo ejemplo en una medida del tamaño del vocabulario, ya que cada componente del vector requiere 4 bytes.

n_samples	vocab	memoria
-----------	-------	---------

100	100000	$100 * 100000 * 4 \text{ bytes} = 38 \text{ MB}$
1000	100000	$1000 * 100000 * 4 \text{ bytes} = 381 \text{ MB}$
10000	100000	$10000 * 100000 * 4 \text{ bytes} = 3.7 \text{ GB}$

Si bien hoy en día 4GB es menos de la mitad de la memoria que tiene un PC, los conjuntos de entrenamiento suelen tener más de 10.000 ejemplos así como más de 100.000 tokens de vocabulario. Por esta razón, incorporar más párrafos, por ejemplo de otros personajes o de otras obras, incrementa sustancialmente la memoria requerida para trabajar.

## 1.4. N-Gramas y TF-IDF

### N-Grama

Un n-grama es una extensión natural de Bag-of-Words, en la que en lugar de tomar los tokens individuales de un texto para la representación en vectores, se toman combinaciones de  $n$  tokens consecutivos. Por ejemplo en bi-gramas ( $n=2$ ) se toman combinaciones de dos tokens, tri-gramas ( $n=3$ ) combinaciones de tres tokens y así sucesivamente. Uni-grama es el caso particular con  $n=1$ , en donde se obtiene la representación BoW que usamos en la sección anterior.

Esta técnica permite capturar más información del contexto en el que aparece un token (palabra), en particular combinada con otras palabras. Por otro lado, al contemplar combinaciones de  $n$ -tokens, incrementa el tamaño del vector resultante.

Por ejemplo para los siguientes textos:

```
X_sample = np.array(['Laboratorio', 'Intro DS', 'Machine Learning'])
```

Con uni-gramas tenemos el siguiente resultado vectorizando para “Intro DS”:

```
[[1 1 0 0 0]]
```

Por otro lado con bi-gramas tenemos el siguiente resultado:

```
[[1 1 1 0 0 0 0]]
```

### Representación TF-IDF

Como bien se detalla en [From occurrences to frequencies](#), contar la cantidad de ocurrencias de un token es un buen punto de partida pero no es suficiente. En particular esta representación no es invariante al largo de un texto o documento, ya que a mayor largo, mayor conteo de ocurrencias de un token y por ende dos vectores correspondientes a dos textos, a pesar de hablar del mismo tema, pueden ser muy diferentes.

Una forma de mitigar esto es dividir el número de ocurrencias de cada token, por la cantidad de tokens en el documento. A esto se lo considera  $tf$  (Term Frequency).

No obstante, las frecuencias de tokens por sí solas, no contemplan el desbalance que se puede dar entre tokens muy frecuentes y quizás menos informativos contra tokens más raros y quizás más informativos. Para solventar esto, se multiplica el factor  $tf$  por un factor  $idf$  (Inverse Document Frequency). Este factor es el inverso de la frecuencia en la que aparece el token  $t$  en todos los documentos del conjunto de entrenamiento.

De esta forma la frecuencia TF-IDF de un token  $t$  en un documento  $d$  para un conjunto de entrenamiento  $D$  queda dado por:

$$TF - IDF(t, d, D) = TF(t, d) \times IDF(t, D)$$

Comenzamos incorporando esta técnica en los ejemplos de entrenamiento, seteando el parámetro `use_idf=False`. De esta forma estamos cambiando el conteo de tokens de BoW por las frecuencias de tokens en el documento. Más adelante en este trabajo, analizaremos el impacto de utilizar `use_idf=True`.

Nótese que al modificar la forma en que se cuentan los token en un documento, cambia el contenido del mismo pero no el tamaño, ya que el vocabulario mantiene el mismo tamaño.

## 1.5. Reducción Dimensionalidad: PCA

Las representaciones TF-IDF de los ejemplos de entrenamiento son vectores de 2831 componentes. Esto es relativamente útil para capturar información, pero imposibilita cualquier tipo de análisis gráfico en las que estamos restringidos a típicamente dos o tres dimensiones. Para visualizar los datos necesitamos reducir la dimensionalidad de los vectores. Una técnica ampliamente utilizada para esto es PCA (Principal Component Analysis). En particular scikit-learn implementa esta técnica en la clase [PCA](#).

Comencemos por analizar las dos componentes principales utilizando la clase PCA sobre los vectores resultantes de la codificación TF-IDF (ver en la figura 2). A priori, no se ve ningún patrón claro, que permita separar los párrafos por personaje, de hecho se aprecia bastante ruido en la visualización. Veamos si podemos reducir el ruido en la imagen experimentando con diferentes parámetros en la vectorización.

Lo primero que vamos a hacer es remover stopwords del idioma inglés, utilizando el parámetro `stop_words="english"` en la clase `CountVectorizer`. Este parámetro en la etapa de vectorización con BoW tiene un efecto importante en los vectores resultantes ya que elimina palabras del vocabulario y por consiguiente los vectores se reducen. En particular, pasamos de vectores de 2831 componentes a 2613. En la figura 3, cuadrante superior derecho, se pueden ver los resultados. En particular se aprecia una disminución del

ruido. De todas formas, no es suficiente distanciar en el espacio, los vectores correspondientes a párrafos de diferentes personajes.

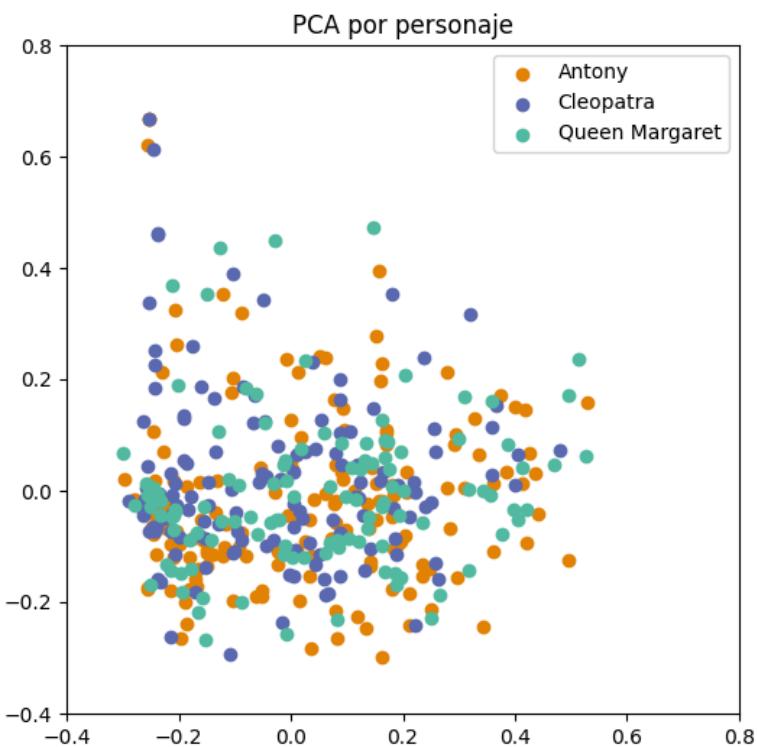


Figura 2 - PCA por Personaje

Veamos si podemos lograr algún cambio significativo, cambiando la lista de stopwords de scikit-learn por la lista de stopwords del modelo `en-core-web-lg` de spacy. Adicionalmente, podemos incorporar el conjunto de stopwords identificadas en el Laboratorio 1 entre las que se incluyen las contracciones de inglés como “ve”, “ll” y “t” así como las palabras “thy” y “thou”.

Incorporando estos cambios en la vectorización, obtenemos resultados notoriamente diferentes. Notar que el tamaño de los vectores no cambia significativamente: 2607 componentes. Si bien los párrafos de los tres personajes se encuentran superpuestos, imposibilitando algún tipo de diferenciación visual (ver cuadrante inferior izquierdo), se distinguen algunos patrones.

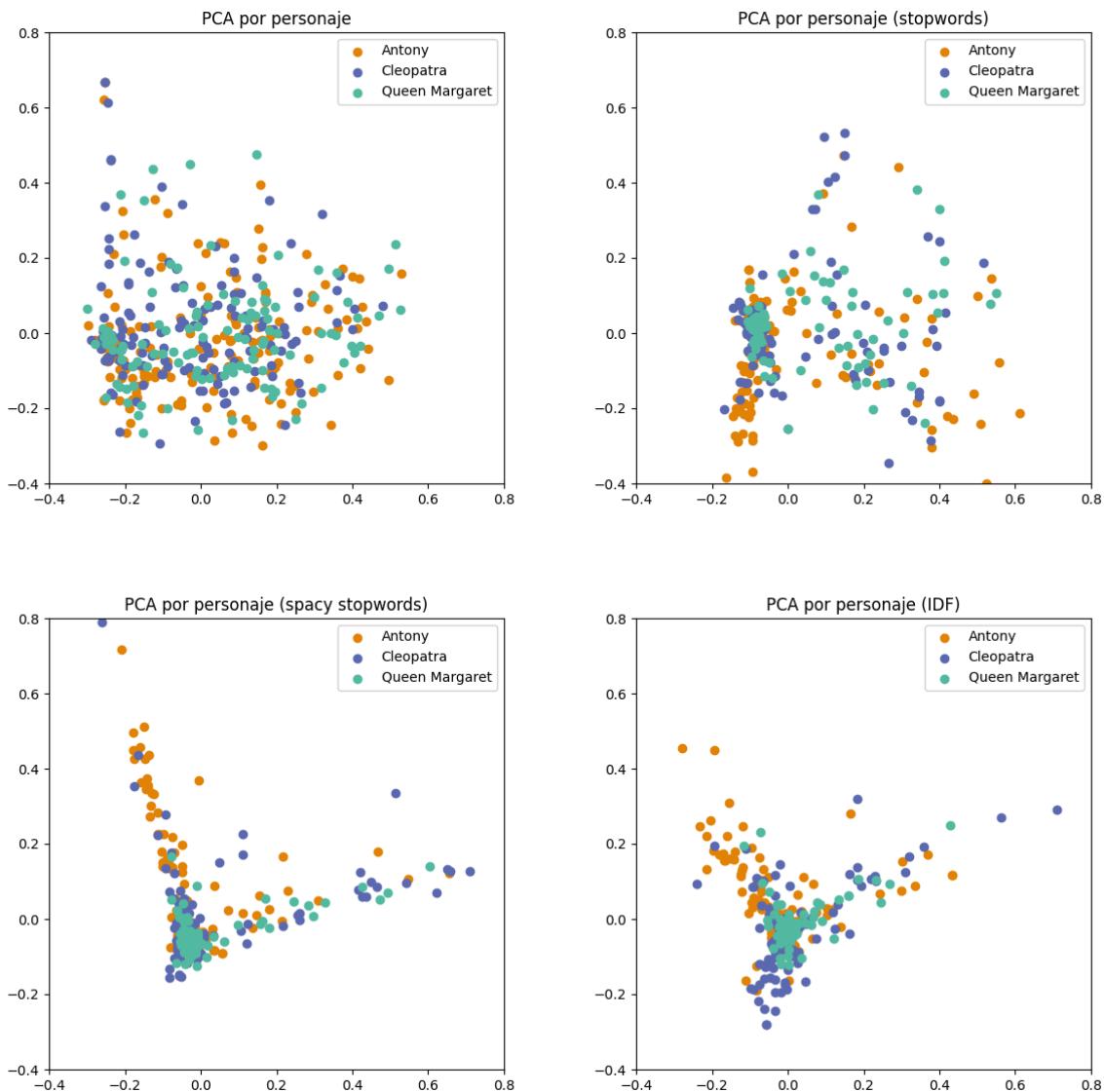


Figura 3 - PCA por personaje (de izq a der)  
 PCA, PCA + Stopwords, PCA + Spacy Stopwords, PCA + IDF

A continuación, veamos que sucede incorporando el parámetro `use_idf=True` en la vectorización TF-IDF el cual disminuye el peso de tokens muy frecuentes para nivelarlos con tokens poco frecuentes, vemos cambios sutiles en las dos principales componentes (ver cuadrante inferior derecho). En particular se distingue una sección de párrafos del personaje Antony diferenciable sobre la parte superior izquierda. Por otro lado, los párrafos asociados al personaje Queen Margaret se concentran sobre el  $<0,0>$  de la imagen. Sin embargo, es muy difícil separarlos visualmente.

Como hemos explicado anteriormente en este documento, una variante en la vectorización de textos es el uso de n-gramas. Veamos cómo impacta a nuestro análisis PCA la utilización de bi-gramas, tri-gramas y cuatri-gramas (además de uni-gramas). En la figura 4 se pueden ver los resultados para n-gramas con  $n$  de 1 .. 4.

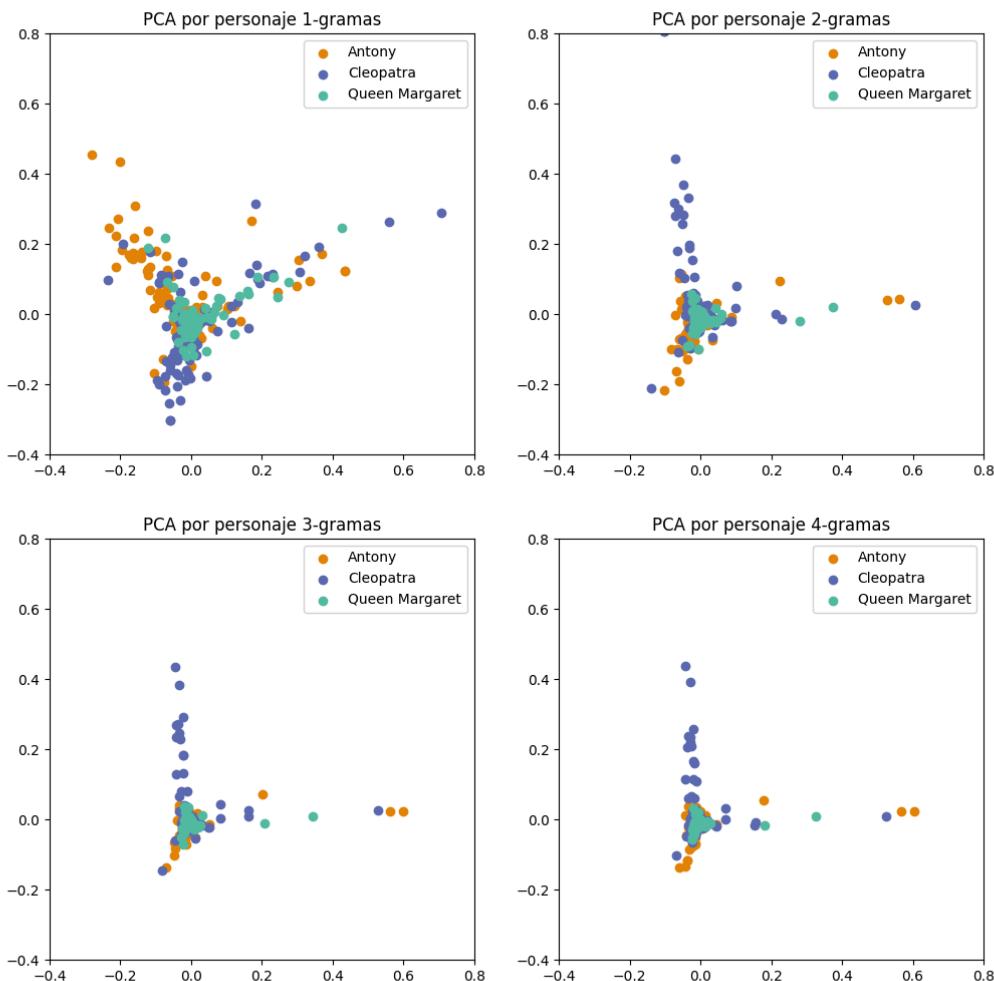


Figura 4 - PCA por Personaje usando N-Gramas

Del gráfico anterior deducimos que a medida que tomamos n-gramas cada vez más grandes, los datos reducidos por PCA comienzan a estar cada vez más concentrados. De todas formas, se puede observar una pequeña reducción del ruido, apreciando un conjunto de párrafos relacionados al personaje Cleopatra que son fácilmente distinguibles del resto (parte superior del gráfico). Por otro lado, los párrafos de Queen Margaret siguen estando concentrados junto a los de Antony. A su vez, se mantiene una concentración de los vectores correspondientes a los tres personajes en el  $<0,0>$ .

¿Contienen información relevante los símbolos de puntuación?

En algunos escenarios, símbolos de puntuación pueden contener información relevante para el contexto y por lo tanto, preservarlos en los ejemplos de entrenamiento puede

llevarnos a mejores resultados. Analizamos si mantener los símbolos de puntuación tiene algún impacto significativo en las dos principales componentes.

Para esto primero pre-procesamos nuevamente los datos de entrenamiento utilizando la función `clean_text` y en este caso, no reemplazamos los símbolos de puntuación. Además, necesitamos indicar el parámetro `token_pattern` en la función `CountVectorizer` cuando construimos la representación BoW. Esto es porque por defecto esta función, ignora los símbolos de puntuación.

```
token_pattern=r" (?u) \b\w\w+\b|\\!|\\?|\\\"|\\'|\\.|\\;|\\:"
```

En la Figura 5 podemos ver el resultado del PCA sobre los párrafos conteniendo los símbolos de puntuación. A priori no parece aportar información relevante sobre los personajes, por lo contrario, parece agregar ruido a la visualización.

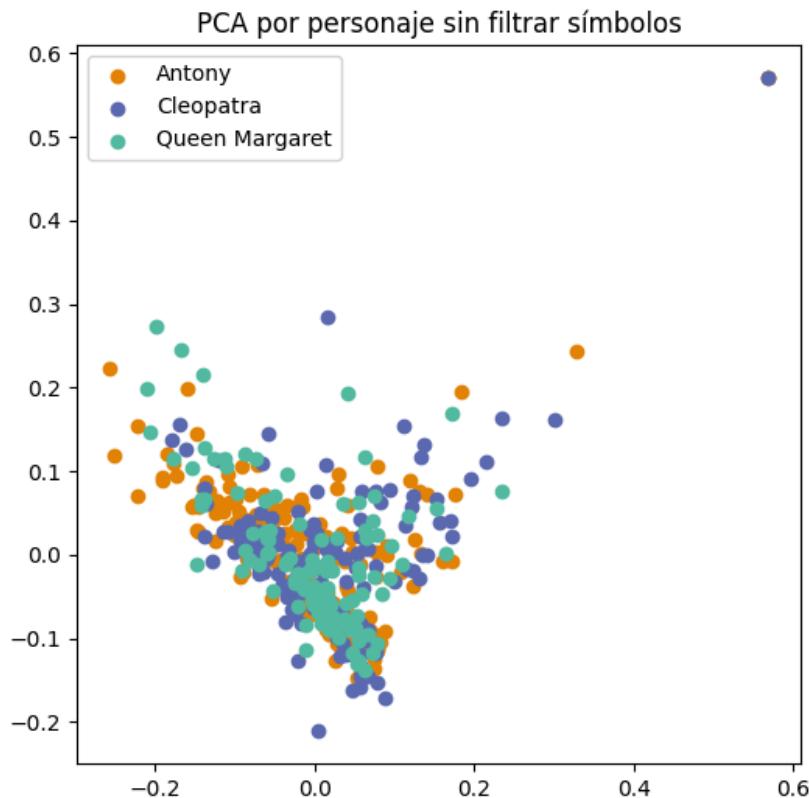


Figura 5 - PCA por Personaje Con Símbolos Puntuación

¿Se pueden separar los personajes utilizando sólo 2 componentes principales?

En base a las imágenes generadas para los resultados de PCA para las dos componentes principales, utilizando las diferentes variantes de pre-procesamiento y vectorización con las que experimentamos, concluimos que no es posible separar los párrafos correspondientes a

estos tres personajes, utilizando solamente dos componentes principales. Esto puede deberse a que las dos componentes principales no capturan suficiente información sobre los párrafos seleccionados. Tener en cuenta que estamos reduciendo la información de features muy primitivas (BoW y TF-IDF) en vectores muy grandes y esparsos (2600-2800 componentes) a dos componentes. En particular estos resultados van en consonancia con los resultados obtenidos analizando la varianza explicada en las N componentes principales, resultados que presentamos a continuación.

## Varianza Explicada en las N Componentes Principales

La varianza explicada en PCA se refiere a la proporción de la varianza total del dataset, capturada por cada componente principal. En otras palabras, mide cuánta información (varianza) en el dataset original es capturada por cada componente principal.

En la siguiente imagen (figura 6) se puede ver la varianza capturada por las N componentes principales, variando N entre 1 y 10. En particular sorprende la poca varianza capturada por las dos primeras componentes principales. Incluso contemplando las primeras 10 componentes principales se alcanza apenas una varianza explicada de 0.0868. Esto explica en gran medida los pobres resultados intentando separar los párrafos de los personajes utilizando las dos componentes principales.

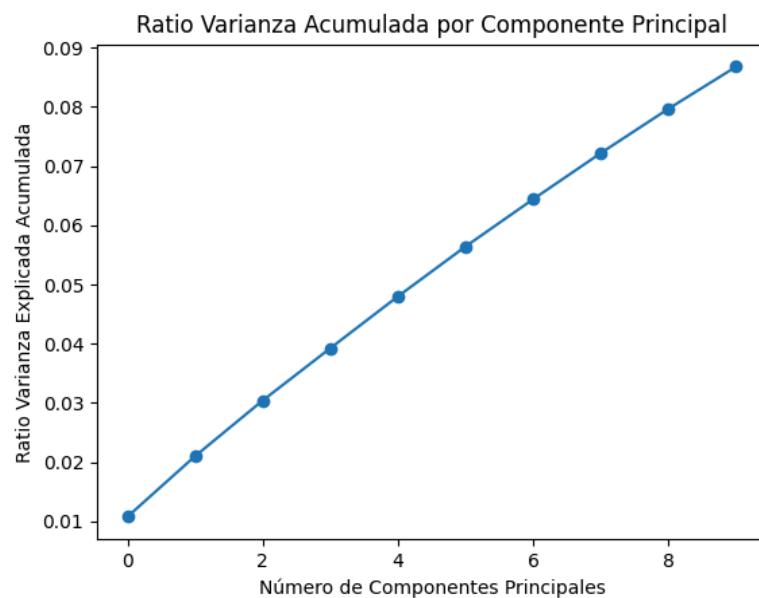


Figura 6 - Ratio Varianza Acumulada por Componente Principal Top 10

Cuando se trabaja con PCA una buena medida de éxito, usualmente utilizada es capturar el 0.8 de la varianza explicada. Cuando se logra capturar este valor de la varianza explicada con las dos componentes principales, se entiende que se logra un buen PCA sobre los datos. En la figura 7 se puede ver la varianza explicada para las primeras 300 componentes principales (recordemos que los vectores tienen del orden de 2600 componentes). Hace falta cerca de 300 componentes principales para capturar 0.8 de la varianza explicada.

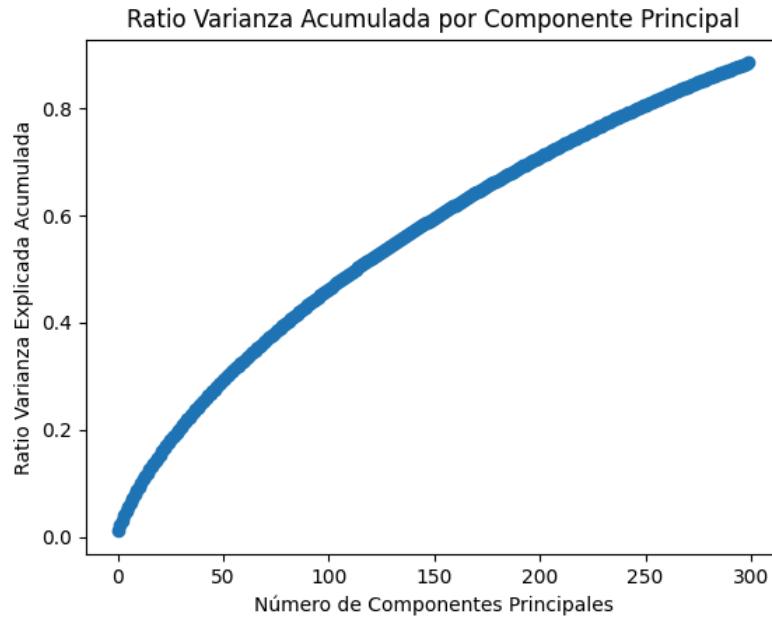
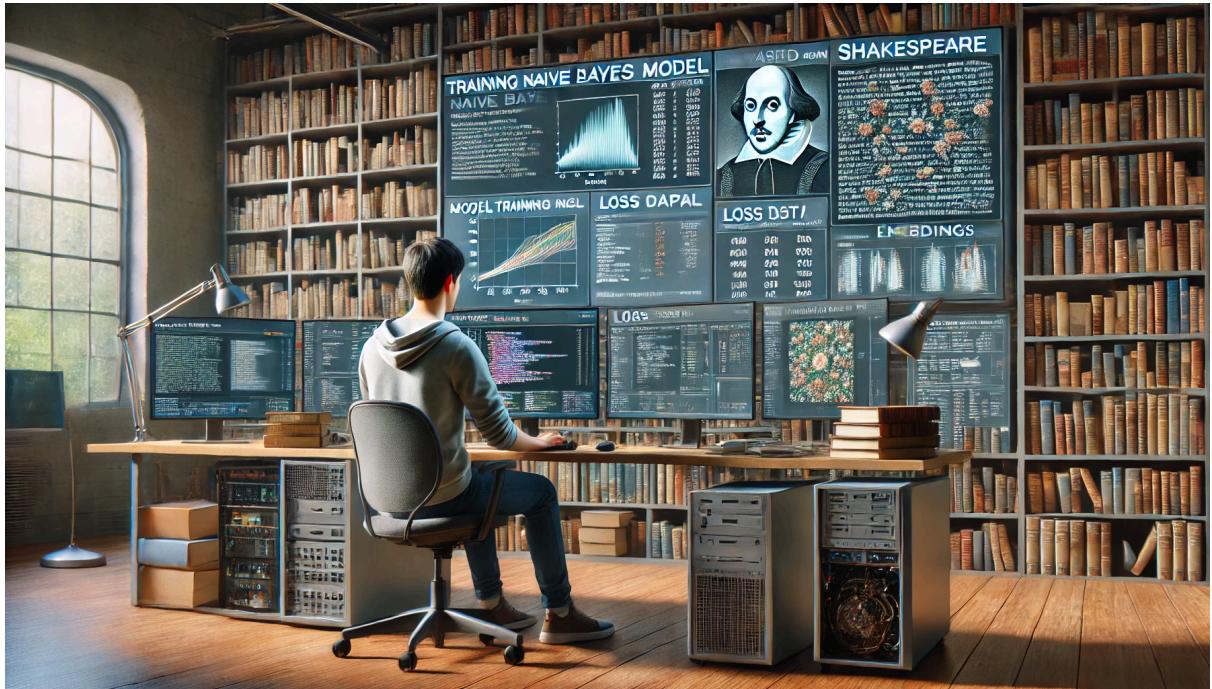


Figura 7 - Ratio Varianza Acumulada por Componente Principal Top 300

En conclusión podemos sacar en limpio que las features utilizadas (BoW, TF-IDF y N-Grams) generan representaciones vectoriales muy dispersas y de alta dimensionalidad. Estos vectores poco densos por definición, capturan poca información y no favorecen métodos como PCA para la reducción de la dimensionalidad. En particular, hacen muy difícil obtener conclusiones a partir de una análisis visual, ya que estamos perdiendo información relevante presente en las otras componentes.

## 2. Segunda Parte: Entrenamiento y Evaluación de Modelos



### 2.1. Clasificador Multinomial Naive-Bayes

El clasificador Naive-Bayes es un clasificador probabilístico, con base en el teorema de Bayes. Este modelo asume que las características de un dataset son mutuamente independientes. En la realidad este no es el caso pero el clasificador tiende a obtener resultados correctos incluso cuando la suposición anterior no se cumple.

Luego de entrenar este modelo y predecir el posible personaje relacionado a los párrafos del conjunto de testing se pueden obtener diversas métricas. En la siguiente tabla se resumen la precisión, recall para cada personaje y la accuracy total.

Personaje	Precision	Recall
Antony	0,51	0,86
Cleopatra	0,58	0,30
Queen Margaret	0,79	0,45
<b>Accuracy</b>		0,56

Tabla 1: Valores de precision, recall y accuracy para los personajes utilizados en el entrenamiento y posterior medida del desempeño del clasificador Naive-Bayes.

Los datos presentados en la tabla anterior fueron obtenidos utilizando el método `classifications_report` de la librería `sklearn`<sup>5</sup>.

Si se toman en cuenta todas las instancias en el que el modelo predice a un cierto personaje dado un párrafo, la precisión define cuántas de ellas pertenecían realmente a esa clase. Formalmente se define como el cociente de predicciones correctas sobre todas las predicciones hechas para esa clase (correctas e incorrectas). Por ejemplo en el caso de Antony, de todas las instancias que el modelo clasificó como Antony sólo el 51% de estas fueron correctas. Por otro lado, si se toman en cuenta todas las instancias verdaderas de un personaje, el recall define cuántas de ellas fueron correctamente predecidas por el modelo. En el caso de Antony, el modelo predijo correctamente el 86% de los párrafos que originalmente correspondían a este personaje.

A partir de las predicciones del modelo se puede generar una matriz de confusión cómo se observa en la siguiente figura:

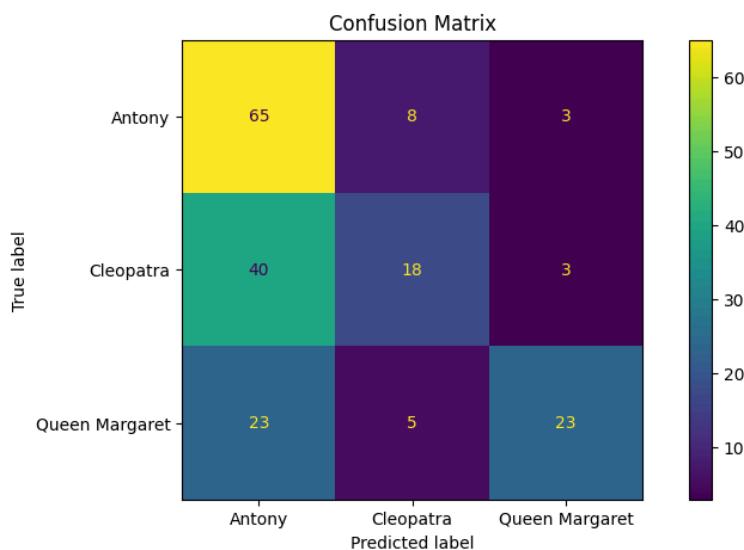


Figura 8: Matriz de confusión para las predicciones del modelo Naive-Bayes.

De la matriz de confusión anterior, se puede observar cómo el modelo encuentra y clasifica correctamente una gran proporción de párrafos que pertenecen a Antony (65 casos). En cambio, con los otros dos personajes, tiende a pasar por alto varios párrafos. En particular, se nota un sesgo hacia la predicción de la categoría Antony, muchas veces errónea, reduciendo así la precisión de la categoría Antony y el recall de las otras dos (Cleopatra y Queen Margaret). En pocas palabras, el modelo genera un sesgo de clasificación hacia la etiqueta Antony, logrando un alto recall en esta categoría, baja precisión y por consiguiente además, con resultados sensiblemente menores para las otras dos categorías.

### ¿Qué problemas puede tener mirar solamente Accuracy?

La accuracy se define para un problema de clasificación como el cociente de predicciones correctas sobre el total de predicciones realizadas.

Justamente por todo lo anterior y contemplando el desbalance en los datos de entrenamiento, la métrica accuracy no es una medida de gran valor para medir el desempeño global de este clasificador. Otra forma de ver esto es, si la mayoría de los

párrafos pertenecen al personaje Antony, con un modelo con sesgo de clasificación al personaje Antony, se puede obtener una Accuracy alto. Por ejemplo, si el 90% de los párrafos pertenecen a Antony, un modelo que sólo clasifica a los párrafos como pertenecientes cómo Antony tendría un 90% de accuracy incluso teniendo en cuenta que no está prediciendo correctamente ninguno de los otros personajes.

Por esta razón en cualquier problema de clasificación no es bueno mirar solamente la Accuracy, ya que puede ocultar problemas en categorías o etiquetas específicas. Con más razón aún, cuando se trabajan con datos desbalanceados en las clases a predecir, no es recomendable mirar solamente la Accuracy.

## 2.2. Validación Cruzada (Cross-Validation)

La validación cruzada es un método estadístico utilizado para poder evaluar los resultados de un modelo de machine learning. Se intenta ver cómo el modelo se desempeña en un dataset independiente, de manera que el desempeño no esté atado a la separación de los datos entre entrenamiento y evaluación.

Existen varios tipos de validación cruzada, en primer lugar tenemos la validación cruzada de K iteraciones. En esta el dataset es particionado en k subset iguales del dataset, se evalúa el modelo en cada uno y se genera un promedio de los resultados.

Otro tipo se denomina Validación Cruzada Estratificada, esta es una variación de la anterior con la diferencia que los subsets son creados de manera que se mantenga la misma proporción de cada clase en cada división.

Si para nuestro modelo Naive-Bayes realizamos validación cruzada podemos obtener la siguiente figura:

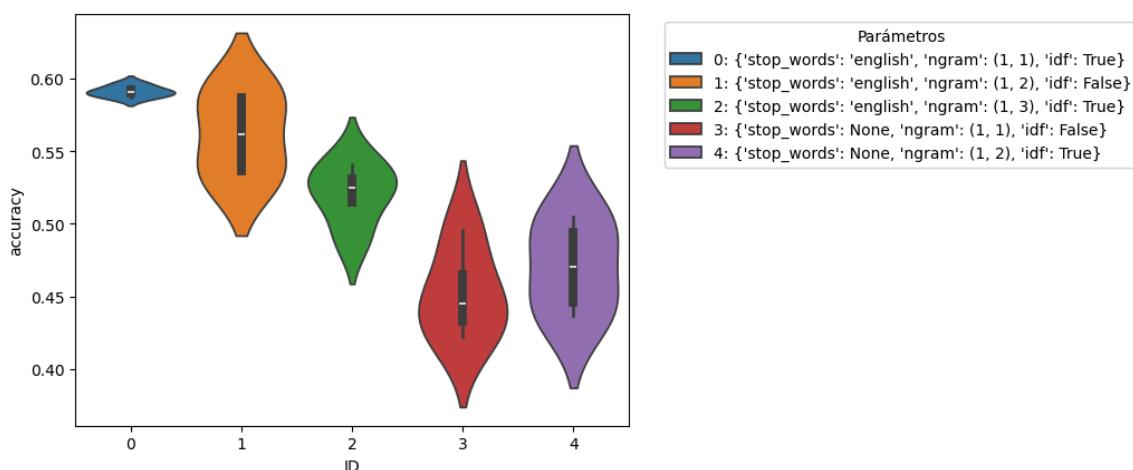


Figura 9: Resultados de la validación cruzada del modelo Naive-Bayes con diferentes juegos de parámetros detallados en la leyenda de la figura.

De esta figura podemos realizar diversas observaciones. El nivel de accuracy más alto se obtiene con los parámetros `{'stop_words': 'english', 'ngram': (1, 1), 'idf': True}`, esto sugiere que si se remueven las stopwords del Inglés y usando unigramas como tokens de entrada para IDF se obtiene el mejor desempeño para este este dataset en particular.

A su vez, podemos observar que la accuracy tiende a ser más alta cuándo se utiliza stopwords en inglés en relación a cuándo no se utiliza, se observa en los conjuntos de parámetros 1, 2 y 3 en comparación con el 4 y 5.

Otra observación está relacionada con el uso de IDF, dado que cuándo es utilizado la accuracy tiende a ser más alta, esto sugiere que el uso de IDF el cual reduce el peso de las palabras que aparecen repetidas veces en muchos documentos, mejora el desempeño del modelo.

## 2.3. Selección Modelo

Cómo se vió reflejado en la sección anterior la mejor configuración de parámetros es aquella que utiliza la stopwords de inglés, IDF y unigramas. Los resultados entrenando el modelo con esta configuración utilizando todos los datos del dataset de entrenamiento se muestran en la siguiente tabla.

Personaje	Precision	Recall
Antony	0,52	0,88
Cleopatra	0,59	0,31
Queen Margaret	0,81	0,41
<b>Accuracy</b>	0,57	

Tabla 2: Valores de precision, recall y accuracy para los personajes utilizados en el entrenamiento y posterior medida del desempeño del clasificador Naive-Baye con los parámetros `{'stop_words': 'english', 'ngram': (1, 1), 'idf': True}`.

En este caso el modelo en comparación con el modelo evaluado en la sección 3.1 tiene un desempeño mayor. Se ve cómo aumenta la precisión en todos los personajes, pero esta diferencia se ve marcada en los personajes Queen Margaret y Cleopatra. Se observa también un aumento en el recall de los personajes mencionados, el modelo pudo clasificar mejor a los personajes que no eran Antony al contrario de la versión anterior del modelo.

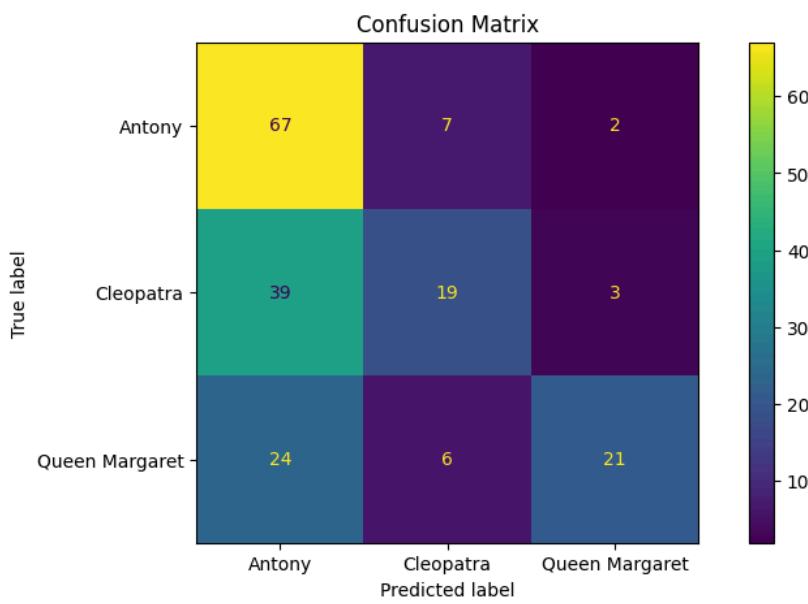


Figura 10: Matriz de confusión para las predicciones del modelo Naive-Bayes con los parámetros `{'stop_words': 'english', 'ngram': (1, 1), 'idf': True}`.

En la figura anterior se muestra la matriz de confusión en la cuál se observa un desplazamiento de las predicciones del modelo alejándose de siempre predecir a Antony como personaje, en comparación la matriz de confusión de la sección 2.1.

Los modelos basados en bag-of-words (BoW) y TF-IDF cómo bien mencionamos anteriormente, fueron el estado del arte para tareas de análisis de texto por varios años. Sin embargo, tienen varias limitaciones y desventajas. Por esa razón fueron reemplazados por técnicas más poderosas y eficientes. Una de las principales desventajas de estos algoritmos es la pérdida del contexto al tokenizar los textos, ambos algoritmos consideran a los documentos como colecciones individuales de palabras, ignorando el orden en que las palabras aparecen. Esto genera una pérdida del contexto y la capacidad de poder entender la semántica del lenguaje, cómo pueden ser expresiones particulares o la polisemia. Por ejemplo, las palabras rey y reina están relacionadas pero este tipo de tokenización no preserva esa relación.

Otra desventaja es la gran dimensionalidad que tiene la representación vectorial de estos algoritmos (mencionado en la sección 1.3), esta puede dar lugar a ineficiencia computacional y desafíos a la hora de almacenar y procesar estos vectores.

## 2.4. Modelando con Otras Arquitecturas

### SVM

El modelo elegido para comparar con Naive Bayes es Support Vector Machines o SVM (support vector machines), este es otro algoritmo con la capacidad de clasificar ejemplos no vistos durante el entrenamiento.

Este algoritmo funciona en base a la existencia de un hiperplano que separa los datos en base a margen máximo. Por lo que, todos los vectores que estén de un lado del hiperplano

pertenecerán a una categoría, y los datos que estén del lado contrario pertenecerán a otra categoría. Se le denomina support vectors o vectores de soporte a todos los datos que queden en las inmediaciones del hiperplano.

Este modelo emplea la utilización de kernels debido a que si se parte de datos complejos una separación lineal de los mismos no es una opción viable.

En la siguiente figura se muestran los resultados utilizando cross validation para diferentes set de parámetros con el modelo SVM. Se puede ver como el mejor set de parámetros es aquel que tiene unigramas, IDF y el kernel sigmoid. Los dos primeros parámetros acompañan a los mejores parámetros del modelo Naive Bayes.

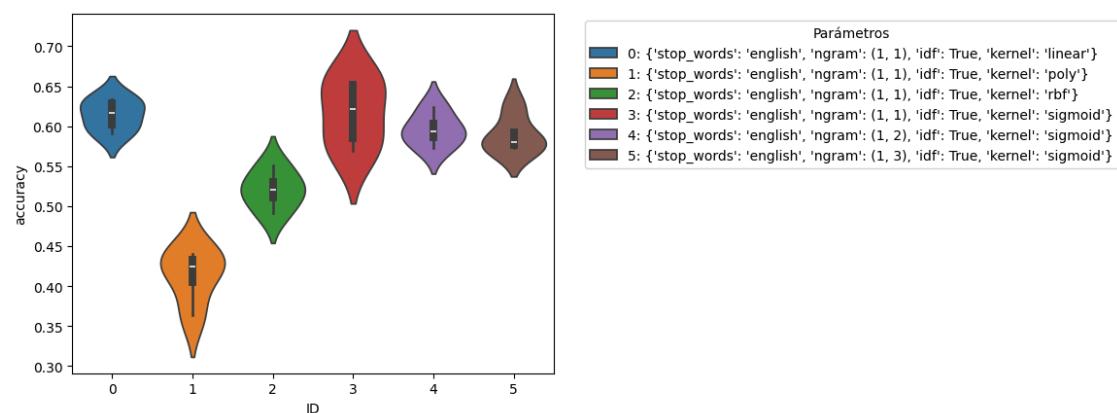


Figura 11: Resultados de la validación cruzada con diferentes conjuntos de parámetros para el modelo SVM.

El desempeño obtenido con este modelo no tuvo una gran diferencia respecto al modelo Naive Bayes, y esto está relacionado también al uso de bag-of-words (BoW) y TF-IDF como tokenizadores debido a la pérdida de semántica y otras desventajas ya mencionadas.

## 2.5. Experimentando con Personajes Adicionales

Para este análisis decidimos utilizar tres personajes que tuvieran una relación de párrafos parecida y que no hubiera una diferencia cómo en el caso de los personajes anteriormente utilizados. Los personajes elegidos fueron *Othello*, *Iago* y *Antony*. En la siguiente gráfica se puede apreciar la cantidad de párrafos por personaje y la similitud entre las mismas.

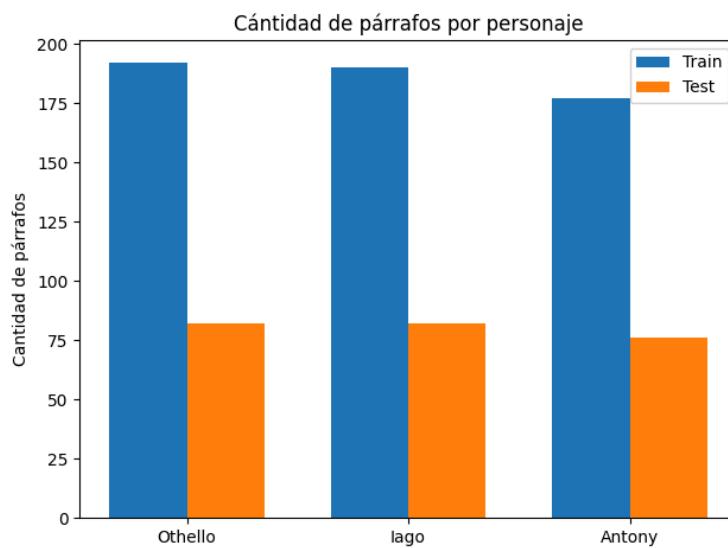


Figura 12: Se muestra la cantidad de párrafos asignados a los 3 personajes elegidos para este análisis, *Othello*, *Iago* y *Antony*.

Si se aplica el modelo de Naive Bayes con diferentes conjuntos de parámetros y utilizando validación cruzada se obtienen los resultados mostrados en la siguiente figura.

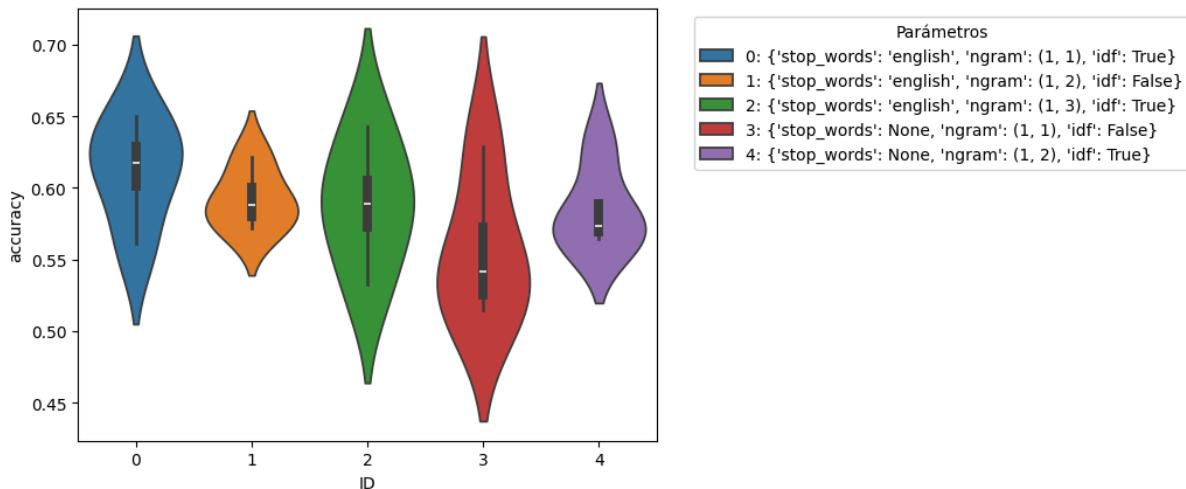


Figura X: Resultados de la validación cruzada del modelo Naive-Bayes con diferentes juegos de parámetros (detallados en la leyenda de la figura) para los personajes, *Othello*, *Iago* y *Antony*.

Estos resultados se diferencian con los anteriores, por ejemplo en que la variación del promedio de los mismos entre los diferentes conjuntos de parámetros es mucho menor y a su vez la distribución de resultados es más homogénea.

Si analizamos los resultados con los parámetros del modelo que mejor se desempeñó para los personajes anteriores obtenemos lo siguiente:

Personaje	Precision	Recall
-----------	-----------	--------

Antony	0,73	0,71
Iago	0,60	0,52
Othello	0,60	0,68
<b>Accuracy</b>	0,64	

Tabla X: Valores de precision, recall y accuracy para los personajes utilizados en el entrenamiento y posterior medida del desempeño del clasificador Naive-Baye con los parámetros `{'stop_words': 'english', 'ngram': (1, 1), 'idf': True}`.

Si comparamos estos resultados con los obtenidos en la sección 2.3 podemos realizar las siguientes observaciones, la accuracy del modelo mejora de 0.58 a 0.63, esto se traduce en que el modelo predice mejor correctamente el personaje al que pertenece el párrafo. Esto se puede deber a que los datos en este modelo están mejor balanceados que en el modelo anterior, teniendo más información en el entrenamiento para poder discernir entre personajes.

Si se toma en cuenta la matriz de confusión para el análisis se ve que en la misma que las predicciones del modelo están más distribuidas y no se encuentran sesgadas hacia un personaje como ocurría anteriormente.

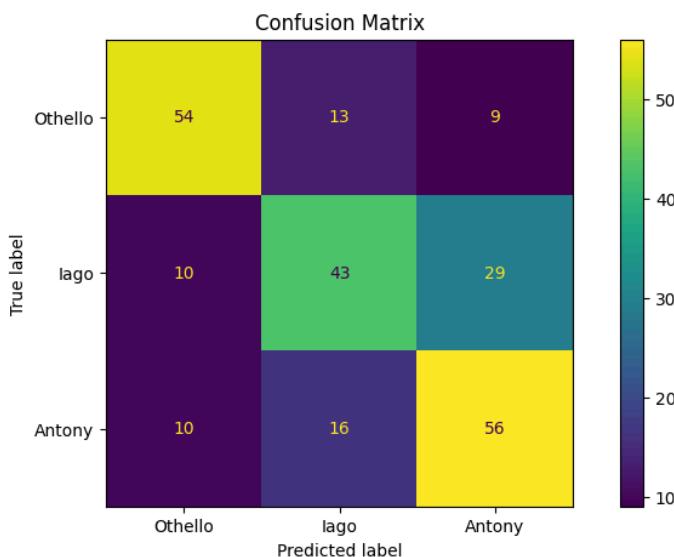


Figura 13: Matriz de confusión para las predicciones del modelo Naive-Bayes con los parámetros `{'stop_words': 'english', 'ngram': (1, 1), 'idf': True}`.

Cuando se trabaja con datasets que se encuentren desbalanceados hacia una o más categorías existen diversas técnicas para poder solucionar este problema. Estas técnicas se basan en realizar un muestreo aleatorio, se puede implementar de dos maneras:

- Sobre-muestreo: Se generan nuevas observaciones para la clase que se encuentra poco representada.

- Sub-muestreo: Se retiran observaciones para la clase que se encuentra más representada.

En cualquiera de los dos casos lo que se intenta es disminuir el sesgo (bias) que puede tener el modelo luego de ser entrenado. Esta es una manera ingenua de hacerlo en el sentido que se ignora completamente las relaciones que pueden llegar a tener las observaciones.

Existen diversas implementaciones, un ejemplo es SMOTE<sup>8</sup>, en este algoritmo se generan nuevas observaciones a partir de la interpolación de las observaciones existentes. La forma en la que el algoritmo genera estas nuevas observaciones sigue una serie de pasos, de manera resumida, se elige una observación dentro de la categoría minoritaria, se encuentran los k vecinos más cercanos, y se elige uno de ellos para realizar la interpolación y generar un nuevo dato.

## 2.6. Técnicas Alternativas de Representaciones de Texto

Como mencionamos anteriormente en este documento, actualmente existen técnicas más complejas y potentes para la representación de textos en vectores. En particular BoW y TF-IDF permiten extraer características que permiten medir similitudes entre textos, pero no logran capturar información de contexto. A continuación enumeramos algunas un conjunto de técnicas posteriores que apuntan a resolver este déficit.

### Matriz de Coocurrencias (Co-occurrences Matrices)

La matriz de coocurrencias es una matriz cuadrada que cuenta las frecuencias de coocurrencias de palabras (o tokens) dentro de una ventana de contexto a lo largo de un corpus. Cada celda de la matriz representa el número de veces que dos palabras (o tokens) aparecen cerca en los textos del corpus.

Formalmente, para un corpus  $C$  de sentencias y un vocabulario  $V$  de tamaño  $N$  (palabras únicas en  $C$ :  $x_1, x_2 \dots x_N$ ), la matriz de coocurrencias  $M$  es una matriz cuadrada de  $N \times N$  en donde  $x_{ij}$  representa la cantidad de veces que las palabras  $x_i$  y  $x_j$  aparecen cerca en las sentencias de  $C$ .

Para fijar ideas, asumamos que las siguientes sentencias representan nuestro corpus de texto:

- Las manzanas son verdes y rojas
- Las manzanas rojas son dulces
- Las naranjas verdes son amargas

Para construir la matriz de coocurrencias primero debemos contar para cada par de palabras en las sentencias, cuántas veces aparecen juntas en cualquier sentencia. Notar que en este caso estamos usando como ventana de contexto la sentencia. Por ejemplo, las

palabras “manzanas” y “rojas” aparecen dos veces, en la primera y en la segunda sentencia, mientras que las palabras “verdes” y “dulces” no aparecen juntas ninguna vez. Siguiendo esta lógica la matriz de coocurrencias queda de la siguiente forma:

Las	manzanas	son	veredes	y	rojas	dulces	naranjas	amargas
manzanas	2	2	1	1	2	1	0	0
son	2	3	2	1	2	1	1	1
veredes	1	2	2	1	1	0	1	1
y	1	1	1	1	1	0	0	0
rojas	2	2	1	1	2	1	0	0
dulces	1	1	0	0	1	1	0	0
naranjas	0	1	1	0	0	0	1	1
amargas	0	1	1	0	0	0	1	1

Notar que la matriz es simétrica ya que en esta representación importa la combinación de palabras dentro de una ventana de contexto y no el orden o la distancia en la que aparecen. Por otro lado, esta representación suele combinarse con PCA o SVD para reducir la dimensionalidad.

### Posibles ventajas

Como mencionamos anteriormente, esta técnica permite capturar cierto grado de información de contexto como las combinaciones de palabras en una ventana de contexto y por consiguiente proveer información más relevante a un clasificador de texto. A su vez, permite reducir la dimensionalidad de las representaciones de texto, haciendo más eficientes tanto el entrenamiento de un modelo como la inferencia.

## Word2Vec

Introducida por *Tomas Mikolov et al.* en *Efficient estimation of word representations in vector space [4]*, parte de la idea de capturar relaciones semánticas entre palabras, representando cada palabra como un vector denso (word embedding), en donde palabras con significados similares se encuentran cerca en dicho espacio vectorial.

Para esto word2vec utiliza vectores de alta dimensionalidad, típicamente 300 o más dimensiones. Estos vectores se aprenden durante el entrenamiento de word2vec para lo cual utiliza una red neuronal poco profunda (shallow neural network). En particular word2vec plantea dos variantes para el entrenamiento:

- **CBOW (Continuous Bag of Words):** El modelo se ajusta a predecir una palabra target (palabra central) a partir de un contexto de palabras que la rodean, con una

ventana determinada. Por ejemplo si el contexto es “Hoy es un <X> soleado” el modelo se entrena para predecir por ejemplo la palabra “soleado”.

- **Skip-gram:** El modelo se ajusta a predecir las palabras de contexto que rodean a una palabra central. Por ejemplo predecir “Hoy”, “es”, “un”, “soleado” a partir de la palabra central “día”.

Por su arquitectura, esta técnica permite capturar el contexto en el que aparece una palabra en una sentencia y las relaciones entre palabras. Esto aporta mucha más información sobre contexto para diferentes tareas de NLP como clasificación.

### **Posibles Ventajas**

Al capturar mayor información de contexto y en particular sobre la forma en que aparecen juntas en una sentencia las palabras del vocabulario, esta representación podría permitirnos capturar mayores diferencias en el uso del lenguaje entre un personaje y otro y de esta forma mejorar los resultados en la clasificación.

Por otro lado, sería bueno entrenar los vectores de word2vec tanto sobre los párrafos de los personajes seleccionados como sobre todos los párrafos del corpus de Shakespeare y comparar los resultados. Es posible que al incorporar más párrafos incorporamos más ejemplos de uso del lenguaje y por consiguiente se pueda aprender representaciones más relevantes para el problema.

### 3. Opcionales: Modelo Fasttext



Fasttext es una biblioteca open-source disponible en Python que incluye diferentes técnicas de representación de texto, así como también entrenar clasificadores de texto basados en estas representaciones. Originalmente fue desarrollada por Facebook y abierta a la comunidad, quien se encarga de su desarrollo.

En el contexto de este trabajo, experimentamos con Fasttext en diferentes grados, buscando entrenar un clasificador más flexible y con una performance mejor sobre nuestro dataset de evaluación:

1. Aprender representaciones de texto a partir de los párrafos de Shakespeare utilizando Fasttext.
2. Entrenar un clasificador de texto en base a estas representaciones.
3. Utilizar representaciones pre-entrenadas sobre corpus en inglés más extensos y complejos.

En las siguientes secciones profundizamos en cada uno de estos experimentos. Por otro lado, todo el código referente a estos experimentos se encuentra en el siguiente [notebook](#).

#### Aprendiendo Representaciones

Para vectorizar textos con Fasttext se puede o bien utilizar un modelo pre-entrenado sobre un corpus general (típicamente modelos grandes y robustos) o aprender representaciones a partir de un corpus específico. En esta sección vamos a utilizar la segunda opción para experimentar con las capacidades de esta biblioteca.

En particular Fasttext permite entrenar un conjunto de embeddings utilizando la función `train_unsupervised`, la cual recibe un archivo con los textos de entrenamiento y aprende una representación vectorial en un espacio denso, también conocido como word embeddings. Para esto ofrece dos técnicas: Skipgrams y CBOW (Continuous-Bag-of-Words). Ambas técnicas son ampliamente aceptadas en el estado del arte y utilizadas por otras técnicas de aprendizaje similares como [GloVe](#).

A continuación se muestra como se pueden aprender estas representaciones con una simple línea de código:

```
import fasttext

model_skipgram = fasttext.train_unsupervised(filename_train,
model="skipgram")
```

Notar que la forma en que se proporcionan los textos de entrenamiento es a partir de un archivo `.txt` el cual contiene los textos y también las etiquetas asociadas (en nuestro caso las etiquetas representan el personaje asociado al párrafo).

Utilizando esta `skipgrams` sobre los textos con el mismo pre-procesamiento utilizado anteriormente (normalización, filtrado stopwords, etc.), obtenemos vectores de largo 100 componentes. Podemos ver en la figura 14 el resultado de analizar con PCA las dos componentes principales de estos vectores para cada personaje.

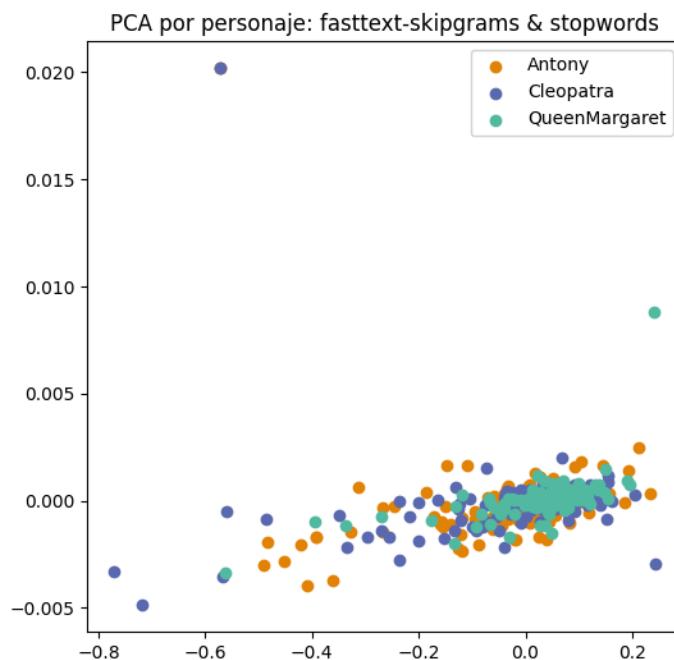


Figura 14 - PCA por personaje utilizando fasttext-skipgrams

Si bien en las representaciones aprendidas no es posible separar los personajes utilizando simplemente dos componentes, es muy interesante ver como se captaron patrones similares con representaciones vectoriales mucho más compactas. Recordemos que en la sección 1 utilizamos vectores de 2600 componentes y aquí estamos usando tan solo 100. Lo que es incluso más interesante aún es el ratio de varianza acumulada en las 10 primeras componentes principales. Recordemos que para las representaciones BoW y TF-IDF de la parte 1 vimos que se requieren al menos las primeras 300 dimensiones para capturar el 80% de la varianza en el dataset. Sin embargo con estos vectores en tan solo con las 2 componentes principales se representa más del 99% de la varianza en los datos (ver figura 15).

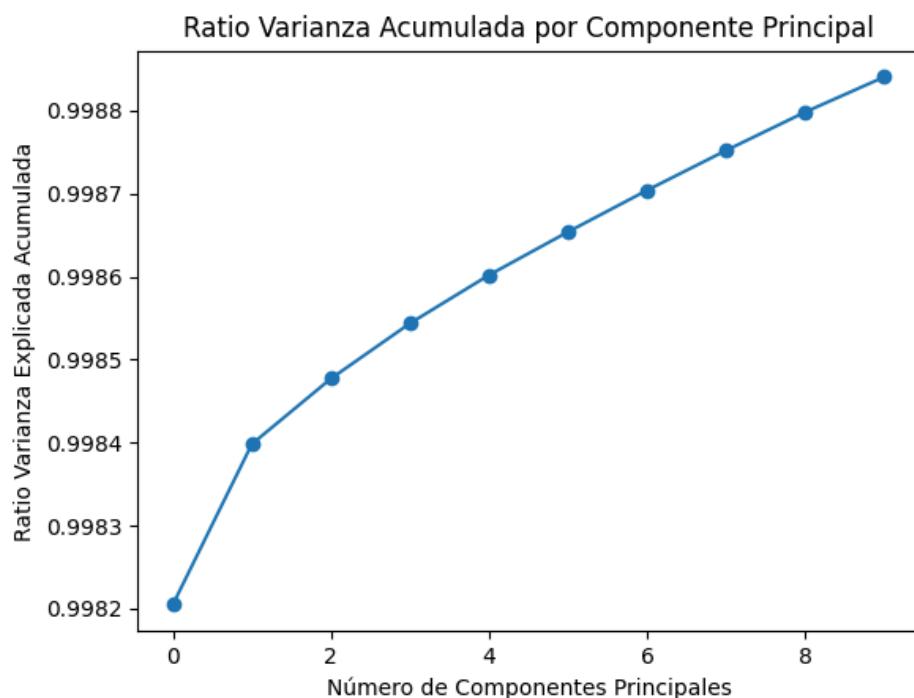


Figura 15 - Ratio Varianza Acumulada por Componente Principal Top 10

A continuación veamos qué sucede si en lugar de utilizar skipgrams utilizamos cbow. En la figura 16 se muestran las dos componentes principales para los mismos ejemplos de entrenamiento. Cambia un poco la distribución de los datos en el plano, pero siguen sin presentarse patrones claros que permitan separar los párrafos.

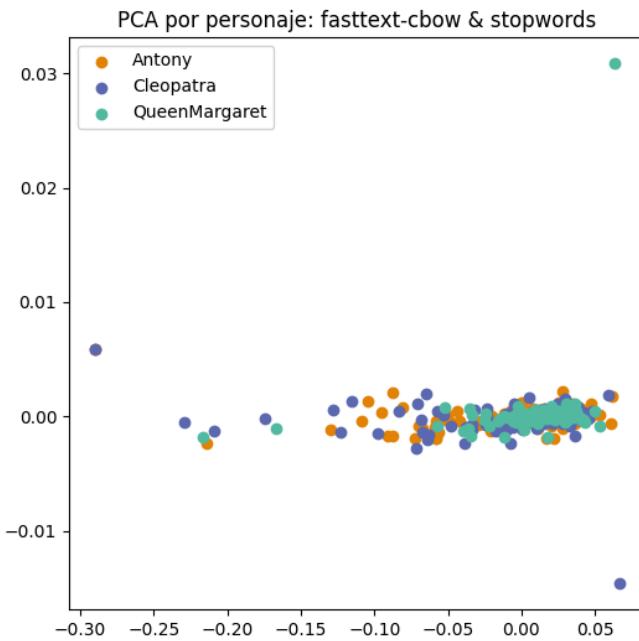


Figura 16 - PCA por personaje utilizando fasttext-cbow

A pesar de los resultados de PCA poco alentadores, probemos cómo nos va entrenando un clasificador de texto utilizando esta biblioteca.

## Entrenando un Clasificador de Texto

Vamos a probar la opción de entrenamiento utilizando nada más que los textos provistos. Internamente esta opción construye las representaciones a partir del texto y utiliza el método de k-nearest neighbors (KNN) para devolver una predicción.

A los efectos de probar la librería entrenamos con un conjunto de hiper parámetros arbitrarios.

```
# Train model
model_classifier = fasttext.train_supervised(
    input=train_file, epoch=2000, lr=0.01, wordNgrams=1, verbose=2,
minCount=1
)
```

Con estos parámetros de experimentación obtenemos los siguientes resultados:

Personaje	Precision	Recall
Antony	0,57	0,65
Cleopatra	0,52	0,49
Queen Margaret	0,79	0,66
<b>Accuracy</b>	0,65	

A continuación se muestra la matriz de confusión:

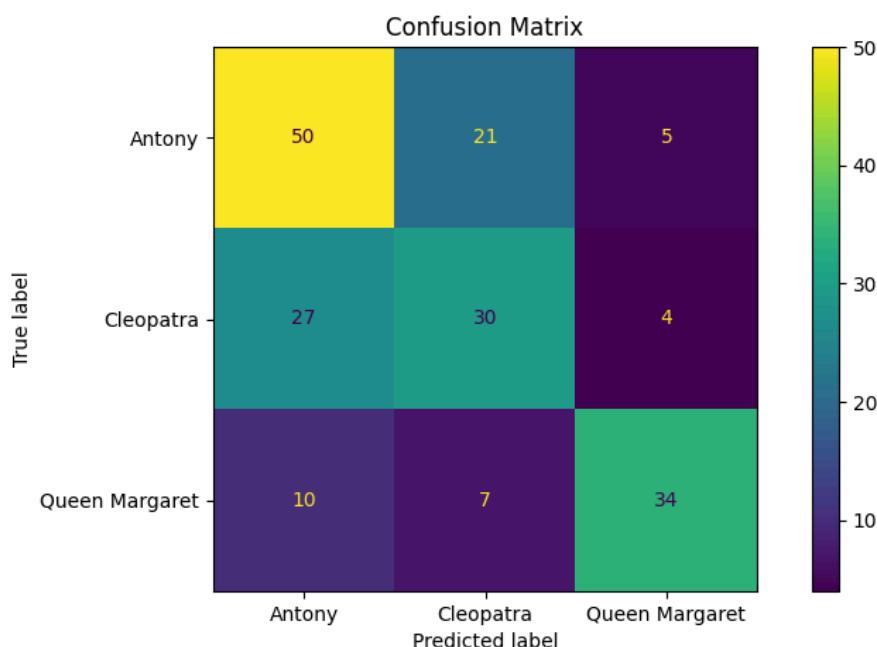


Figura 17 - Matriz de Confusión Fasttext Clasifier

## Usando embeddings pre-entrenados

Unas de las ventajas que ofrece una biblioteca como fasttext es la disponibilidad de conjuntos de embeddings pre-entrenados sobre diferentes idiomas. Esto permite utilizar representaciones de texto más potentes, entrenadas sobre cientos de miles de textos cuidadosamente curados y pre-procesados y entrenar modelos encima de estos vectores.

De entre los 157 vectores [disponibles](#) para diferentes idiomas, vamos a utilizar los vectores para inglés denominados cc-en-300. Estos son vectores de dimensión 300 entrenados sobre [Common Crawl](#) y [Wikipedia](#).

Como se aprecia en la figura 18, de analizar las dos primeras componentes principales con PCA, vemos que estas representaciones tampoco permiten separar fácilmente los personajes.

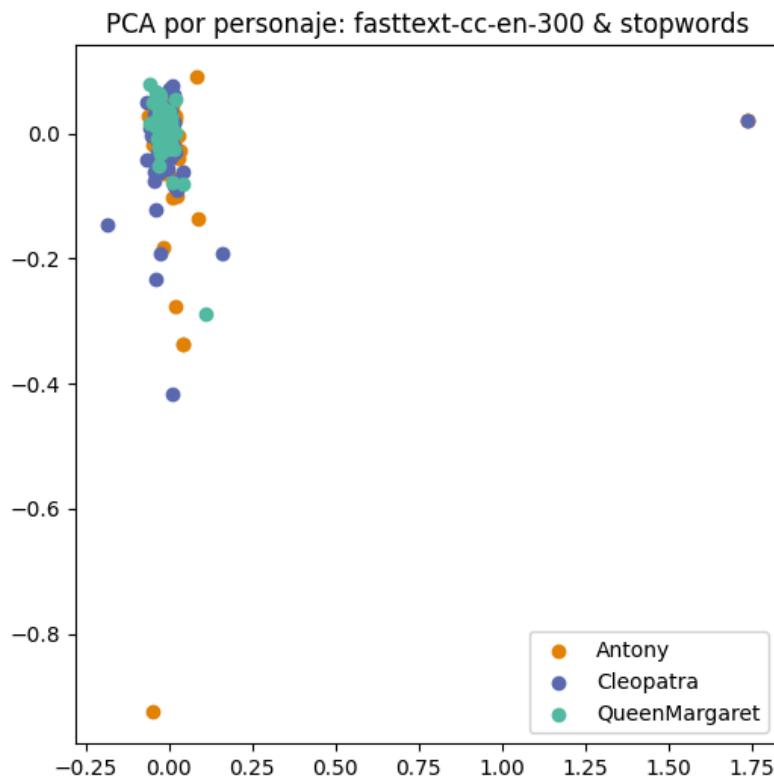


Figura 18 - PCA por personaje utilizando cc-en-300

Desafortunadamente hasta aquí llegamos con FastText ya que no logramos entrenar un modelo utilizando representaciones pre-entrenadas debido a algunos errores para los cuales no encontramos buena documentación.

De todas formas, estos experimentos alcanzaron para sacar en limpio varias conclusiones:

- Las representaciones obtenidas con skipgrams o cbow son mucho más densas y compactas que las obtenidas utilizando BoW o TF-IDF. Esto permite manejar vectores de dimensiones mucho más pequeñas y por lo tanto más eficientes. Esto nos permitiría entrenar un clasificador utilizando más ejemplos, incluso párrafos de otros personajes.
- Además, la información capturada por las componentes de estos vectores es mucho más rica que con las primeras representaciones (análisis de varianza explicada PCA).
- Por último, los resultados entrenando un clasificador, demuestran que este tipo de técnicas incluso sin prestarle mucha atención a los hiper parámetros nos permiten obtener métricas tan buenas como las técnicas utilizadas en la parte 2.

## 4. Conclusiones

A partir de los experimentos realizados en este trabajo, podemos concluir lo siguiente:

- No es posible, o al menos con la combinación de pre-procesamiento y técnicas utilizadas, separar los personajes utilizando únicamente las dos primeras componentes del análisis PCA. Intuitivamente pensamos que se debía a que las mismas no alcanzan a capturar suficiente información de los datos. Sin embargo, el mismo análisis realizado con PCA sobre las representaciones aprendidas con FastText derriba esta teoría.
- A la hora de entrenar modelos es importante que los datos estén balanceados para todas las categorías (sin sesgos). Esto se pudo ver reflejado en el caso del personaje Antony cuando analizamos la performance de los modelos entrenados en comparación a los personajes Cleopatra y Queen Margaret.
- Tanto el modelo Naive Bayes como SVM tuvieron un desempeño parecido siendo el segundo el modelo el que mejor se desempeñó.
- Al utilizar para el entrenamiento personajes con una cantidad de párrafos similar, podemos concluir que el sesgo hacia un determinado personaje desaparece pero el modelo no se desempeña drásticamente mejor.
- El uso de técnicas más avanzadas como Word2Vec podría llegar a producir mejores resultados debido a la capacidad de mantener la semántica del lenguaje, ayudando a poder discernir el lenguaje y modismos utilizados por cada personaje en la clasificación. A su vez, generan representaciones más compactas (vectores de menor dimensión) y más densas, que permiten entrenar modelos de forma más eficiente y trabajar con más datos.

## Referencias

1. Base de Datos pública de la obra de Shakespeare con datos para este laboratorio  
<https://relational-data.org/dataset/Shakespeare>
2. Repositorio oficial del curso Intro-CD con código de referencia para Laboratorio 1  
<https://gitlab.fing.edu.uy/maestria-cdaa/intro-cd/>
3. Repositorio del grupo con código fuente y todos los entregables de esta tarea laboratorio  
<https://github.com/efviodo/mcdaa-intro-cd>
4. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.  
<https://arxiv.org/pdf/1301.3781>
5. Pedregosa, F., Varoquaux, Gael, Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... others. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825–2830.
6. Stanford CS224 Natural Language Processing with Deep Learning class notes  
[https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/lecture\\_notes/cs224n-2017-notes2.pdf](https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/lecture_notes/cs224n-2017-notes2.pdf)  
<https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/slides/cs224n-2019-lecture02-wordvecs2.pdf>
7. Fasttext Experimentation Notebook  
[https://github.com/efviodo/mcdaa-intro-cd/blob/main/Tarea2/notebooks/laboratorio\\_2\\_ev\\_fasttext\\_classifier.ipynb](https://github.com/efviodo/mcdaa-intro-cd/blob/main/Tarea2/notebooks/laboratorio_2_ev_fasttext_classifier.ipynb)
8. N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer (2002) SMOTE: Synthetic Minority Over-sampling Technique