



# MEMORIA RETO ACCENTURE:

¿Sabes quién es la máquina?

Alejando Gómez Gómez  
Pablo Martín Viñuelas  
Rafael José Fernández  
Sara Vicente Arroyo


I METODOLOGÍA			4
1	TRATAMIENTO DE LOS DATOS		5
1.1	Ampliación de la base de datos		5
1.2	Pre-procesamiento de los datos		5
1.3	Representación de los datos		7
2	ESTRATEGIA DE CLASIFICACIÓN		10
2.1	Árbol de Decisión		10
2.2	Naïve Bayes		11
2.3	Red Neuronal		12
II RESULTADOS			15
3	EVALUACIÓN Y VALIDACIÓN		16
3.1	Árbol de decisión		16
3.2	Naïve Bayes		18
3.3	Red neuronal		19
III CONCLUSIONES			22
4	OBSERVACIONES		23
4.1	Dependencia de los resultados y los datos		23
4.2	Comparación de los métodos		23
4.3	Principales dificultades encontradas		24
5	CONOCIMIENTOS ADQUIRIDOS		25
IV REFERENCIAS			26
6	REFERENCIAS		27
V APÉNDICE			28
7	TUTORIAL INTERFAZ GRÁFICA		29

## ABSTRACT

La presente memoria tiene como objetivo servir de guía al código que se adjunta en la carpeta de la solución. En los capítulos iniciales se detalla la metodología seguida para la resolución del problema planteado por la compañía *Accenture*. Más concretamente, en el Capítulo 1 se comenta cómo se ha ampliado la base de datos proporcionada por la empresa, así como el subsecuente procesamiento necesario para la aplicación de los distintos algoritmos. El Capítulo 2 presenta brevemente cada uno de los tres procedimientos de clasificación ideados por el grupo. En el Capítulo 3 se evalúa la respuesta de los tres modelos al recibir datos nuevos, para seguidamente discutir el rendimiento de cada uno utilizando métricas varias. Los últimos capítulos están dedicados a conclusiones. En el Capítulo 4 se expone toda una serie de comentarios relevantes, entre los que destaca la comparativa de los resultados obtenidos mediante los distintos métodos. Finalmente, en el Capítulo 5 se discuten los conocimientos adquiridos por los miembros del grupo a lo largo del proceso de resolución del problema. Además, en el Apéndice se presenta un tutorial de uso de una sencilla interfaz gráfica, cuyo propósito es permitir a un usuario escribir manualmente el texto que desee y comparar la decisión de cada uno de los algoritmos implementados.



# METODOLOGÍA



# 1. TRATAMIENTO DE LOS DATOS

## 1.1. AMPLIACIÓN DE LA BASE DE DATOS

El primer problema que se nos planteó fue el de obtener una base razonablemente representativa de textos, tanto de procedencia humana como generados por inteligencia artificial generativa. Ello se debe a que los principales algoritmos que introduciremos en las posteriores secciones tienen una fidelidad que depende fuertemente de dicha cantidad.

La búsqueda de un corpus de textos en español resultó complicada, puesto que en la mayoría de los casos el idioma predominante en el que se encuentran es el inglés. En otros casos, las que sí estaban en español eran de acceso restringido, usualmente propiedad de instituciones lingüísticas cuyo acceso requería suscripción. Finalmente decidimos optar por la base de datos proporcionada en este [enlace](#) a *GitHub*.

Resumidamente, en él se nos proporciona una serie de ficheros con extensión *.txt* que encadenan líneas de distintos tipos de textos humanos. Dada la enorme cantidad de información, únicamente utilizamos aquellos procedentes de los ficheros *.txt Source* del *GitHub* proceden de *TED*, *ParaCrawl* y *Global Voices* (para más información de dichas fuentes, el propio autor referencia los links a sus webs).

Los extractos de textos de inteligencia generativa que alimentamos a nuestros modelos han sido creados todos por **ChatGPT-3.5**, principalmente debido a la gran calidad que posee de procesamiento de lenguaje natural. En particular, en la propia *prompt* se le ordenó escribir diversos textos en un fichero de formato CSV (*.csv*), haciendo especial hincapié en que estos fuesen de contenido variado, diversa complejidad gramatical y sintáctica, y explorasen el estilo propio de las distintas clases de textos (narrativos, periodísticos, científicos, etc.).

Debido a la falta de corpus de textos creados por inteligencia generativa, y el tiempo que conlleva manufacturarlos nosotros mismos, nos hemos visto obligados a reducirnos a un conjunto mucho menor del que disponíamos de textos humanos. Una posible fuente de mejora sería la implementación de una API de *ChatGPT* que nos permitiera automatizar la tarea. Esto último no se pudo llevar a cabo debido a que esta funcionalidad está restringida a usuarios de pago suscritos.

## 1.2. PRE-PROCESAMIENTO DE LOS DATOS

Después de establecer una base de datos robusta, el siguiente paso consistió en estandarizar y homogeneizar todos los ficheros de datos en un mismo documento unificado, de modo que su lectura y uso por los diversos modelos resultara lo más eficiente y accesible posible.

Para ello, usando las funciones internas de las librerías **Numpy** y **Pandas** se procedió a la fusión de múltiples ficheros de texto con extensión *.txt* en un solo archivo de formato CSV. En primer lugar, escribimos una sencilla función en Python que usamos para convertir todos los *.txt* a *.csv* que mostramos a continuación.

```

1 def convert_txt_to_csv(input_file, output_file):
2     with open(input_file, 'r', encoding='utf-8') as txt_file:
3         lines = txt_file.readlines()
4
5     # Aquí procesamos cada línea para extraer las frases entre comillas
6     phrases = [line.strip().strip('"') for line in lines]
7
8     # Finalmente, escribimos las frases en un archivo CSV
9     with open(output_file, 'w', newline='', encoding='utf-8') as csv_file:
10         writer = csv.writer(csv_file)
11         for phrase in phrases:
12             writer.writerow([phrase]) # Cada frase en su propia fila

```

Durante la etapa de preparación de datos, nos enfrentamos al primer problema ya mencionado. Identificamos una disparidad notable en el volumen de datos disponibles para el entrenamiento: los datos generados por algoritmos de inteligencia artificial sumaban únicamente 1071 instancias únicas, en contraste con la abundante colección de más de 1.3 millones de frases aportadas por colaboradores humanos. Para garantizar una distribución equitativa y representativa en nuestro conjunto de entrenamiento, optamos por una estrategia de muestreo aleatorio. Esto implicó seleccionar al azar 1071 frases de la reserva generada por humanos, una cantidad que igualara a la proporcionada por la inteligencia generativa. Finalmente, esta selección se complementó con los 120 ejemplos adicionales provistos por Accenture, que utilizamos para evaluar cada modelo una vez entrenado.

Al avanzar en la preparación de los datos, surgió otro reto: detectamos que los conjuntos de datos seleccionados incluían elementos que no correspondían al tipo de dato `string`, una inconsistencia que podría provocar errores en procesos posteriores. Se llevó a cabo un proceso de limpieza para eliminar estos desajustes. Adicionalmente, en un esfuerzo por reforzar la homogeneidad de nuestros datos, decidimos restringir el conjunto de cadenas humanas a aquellas cuya longitud se encontraba en el rango de 130 a 190 caracteres. Esta decisión se fundamentó en que el corpus de texto generado por inteligencia artificial se mantenía consistentemente dentro de este espectro, estableciendo así un criterio uniforme para la comparación de los datos.

```

1 import pandas as pd
2 import numpy as np
3
4 dfIA = pd.read_csv('IA.csv')[0:1071]
5 dfHUMAN = pd.read_csv('Human.csv')
6 dfHUMAN1 = pd.read_csv('Human1.csv')
7 dfHUMAN2 = pd.read_csv('Human2.csv')
8 dfHUMAN = pd.concat([dfHUMAN, dfHUMAN1, dfHUMAN2], ignore_index=True)
9
10 df_HUMAN = dfHUMAN[dfHUMAN['Frases'].apply(lambda x: isinstance(x, str) and
11     130 <= len(x) <= 190)]
12 dfIA = dfIA[dfIA['Frases'].apply(lambda x: isinstance(x, str))]
13 # Las frases de dfIA tienen una longitud de entre 130 y 190 caracteres, por eso
14   restringo los de df_HUMAN#
15 dfIA.insert(1, "Tipo texto", "IA")
16 df_HUMAN.insert(1, "Tipo texto", "Humano")
17 df_HUMAN = df_HUMAN.sample(frac=1, random_state=69).reset_index(drop=True)
18
19 ##Mezclo todo
20 df = pd.concat([dfIA, df_HUMAN[0:1071]], ignore_index=True)
21 df = df.sample(frac=1, random_state=12).reset_index(drop=True)
22 df.to_csv('TRAIN.csv', index=False)

```

Con el fin de optimizar la compactificación de nuestros datos, procedimos a ampliar el archivo CSV con una columna adicional que etiqueta si la información fue generada por humanos o producida artificialmente. El conjunto de datos resultante se encuentra accesible dentro de la carpeta de recursos de este trabajo, bajo el nombre *TRAIN.csv*.

Durante la revisión del archivo, identificamos la presencia de caracteres no deseados, como elementos de código *HTML*, caracteres especiales y saltos de línea, que podrían complicar el proceso de *tokenización*. Para rectificar esto, aplicamos expresiones regulares mediante la biblioteca **Regular expressions (re)** de Python, lo que nos permitió reemplazar o eliminar estos caracteres de manera efectiva.

Además, nos percatamos de la frecuencia excesiva de ciertas palabras comunes, conocidas como *stopwords*, que no contribuyen con información relevante al análisis de los datos, sino que por el contrario hacen que los análisis tomen más tiempo a la par que aumentan de forma innecesaria la complejidad de los algoritmos. Utilizando la biblioteca **Natural Language Toolkit (nlk)**, implementamos un filtrado para excluir estas palabras de nuestro conjunto de datos. A continuación, se detalla la función empleada para ejecutar este proceso de limpieza.

```

1 import nltk
2 import re
3 import numpy as np
4 nltk.download('stopwords')
5
6 def normalize_document(doc, stopwords=nltk.corpus.stopwords.words('spanish')):
7
8     #remove html tags (Los textos tienen etiquetas html que hacen la
9     #tokenizacion mas dificil)
10    doc = re.sub(r'<[^>]+>', '', doc)
11    #remove withespaces and break lines
12    doc = ' '.join(doc.strip().split())
13    #remove special characters
14    doc = doc.lower()
15    # tokenize document
16    tokens = nltk.WordPunctTokenizer().tokenize(doc)
17    # filter stopwords out of document
18    filtered_tokens = [token for token in tokens if token not in stopwords]
19    # re-create document from filtered tokens
20    doc = ' '.join(filtered_tokens)
21
22    return doc
23
24 normalize_corpus = np.vectorize(normalize_document)

```

## 1.3. REPRESENTACIÓN DE LOS DATOS

### 1.3.1. Representación del texto en el árbol y naive bayes

Para poder entrenar estos dos modelos, tuvimos que transformar cada texto en un vector de variables descriptivas. De esta forma, cada texto de nuestro *dataset* fue procesado para extraer las palabras relevantes que en él aparecen, para después considerarlas todas como un conjunto. Este conjunto lo llamamos *feature words*. La forma en la que vamos a representar el conjunto de los datos es una matriz, donde cada fila representa un texto del *dataset* y cada columna es una de las *feature words*.

A partir de este punto, decidimos considerar de manera paralela dos aproximaciones diferentes, la representación como *bolsa de palabras* y la representación como *Frecuencia de Término-Frecuencia Inversa de Documento (TF-IDF)*.

En la representación como *bolsa de palabras*, cada texto fue transformado en un vector de palabras con su frecuencia de aparición en el texto. Las palabras que no aparecen en dicho texto, pero sí en otros, tienen frecuencia cero. El orden de aparición deja de ser relevante.

Para llevar a cabo esta representación utilizamos la librería **Scikit-learn (sklearn)**, concretamente la función `Count Vectorizer`, que transforma el texto en variables, donde cada variable es una palabra, y cuenta las veces que aparece dicha palabra.

```
1 from sklearn.feature_extraction.text import CountVectorizer
2
3 cv = CountVectorizer()
4 cv_matrix = cv.fit_transform(train_corpus)
5 cv_matrix = cv_matrix.toarray()
```

De esta forma conseguimos un vocabulario de palabras con el que trabajar. Este vocabulario lo obtuvimos utilizando la función `CountVectorizer.get_feature_names_out`.

```
1 #Sacamos las palabras propias del corpus
2 vocab = cv.get_feature_names_out()
```

La otra representación que hemos utilizado ha sido *TF-IDF*, ya que el principal problema de la *bolsa de palabras* es que la frecuencia absoluta de un término puede no ser útil, ya que será muy alta en palabras comunes en general, no necesariamente *stopwords*, pero sí otras comunes. Estas palabras no tienen tanto poder discriminante como otras, comunes en un texto concreto pero no en el corpus, por lo que no debemos darles tanta importancia. El *TF-IDF* cuenta el número de veces que una palabra aparece en un texto, dividido entre el logaritmo del porcentaje de textos que contienen dicho término. Para poder generar estos pesos utilizamos de nuevo la librería **Scikit-learn (sklearn)** pero ahora la función `TfidfVectorizer`, y hacemos lo mismo que con la bolsa de palabras.

```
1 from sklearn.feature_extraction.text import TfidfVectorizer
2
3 tv = TfidfVectorizer()
4 tv_matrix = tv.fit_transform(norm_corpus)
```

Y hacemos lo mismo que con la bolsa de palabras.

```
1 vocabtv = tv.get_feature_names_out()
```

### 1.3.2. Representación del texto en la red neuronal

En esta última aproximación al problema utilizamos la técnica de representación del lenguaje natural conocida como *Word Embedding*, es decir, representamos cada palabra como un elemento de un espacio vectorial de dimensionalidad reducida. Elegimos esta técnica por sus potentes propiedades, entre las que destaca su capacidad de preservar las relaciones semánticas y sintácticas entre palabras mediante la distancia entre sus vectores



asociados. En concreto, utilizamos un *Word Embedding* en español pre-entrenado mediante el algoritmo *Word2Vec*, encontrado en un repositorio de *GitHub* al que se puede acceder a través del siguiente **enlace**. El espacio vectorial que representa es 300-dimensional.

Primero cargamos el *Word Embedding*, con extensión de archivo comprimido *.bz2*, y creamos el diccionario `embedding_index`, que asocia el vector correspondiente a cada palabra.

```
1 import bz2
2 import numpy as np
3
4 # Ubicación del Word-Embedding en español utilizado.
5 file_path = './SBW-vectors-300-min5.txt.bz2'
6
7 # Inicialización del diccionario para el embedding index.
8 embedding_index = {}
9
10 with bz2.open(file_path, 'rt', encoding='utf-8') as f:
11     for line in f:
12         values = line.split()
13         word = values[0] # la palabra es el primer valor en cada línea
14         vector = np.asarray(values[1:], dtype='float32') # el resto de la
15                     línea es el vector
16         embedding_index[word] = vector
```

Como se detallará en la Sección 2.3, la primera capa de la red neuronal que hemos implementado es una del tipo `Embedding` de la librería **Keras**. La entrada para esta capa debe ser una lista de índices, por lo que realizamos un paso adicional al preprocesamiento de los textos del *dataset*. Utilizamos un objeto de la clase `Tokenizer` de la librería **Keras**, para asignar números enteros distintos a cada palabra del corpus y convertir los textos en listas de números de una longitud fijada `MAX_TEXT_LENGTH = 40`. La elección de ese número máximo de palabras nos pareció razonable teniendo en cuenta que el número de caracteres de los textos oscilaba entre 130 y 190.

```
1 from keras.preprocessing.text import Tokenizer
2 from keras.utils import pad_sequences
3
4 # Tokenización del texto
5 tokenizer = Tokenizer()
6 tokenizer.fit_on_texts(df.Frases)
7 vocab_size = len(tokenizer.word_index)
8 sequences = tokenizer.texts_to_sequences(df.Frases)
9 data = pad_sequences(sequences, maxlen=MAX_TEXT_LENGTH)
10
11 # Inicialización de la matriz del embedding
12 embedding_matrix = np.zeros((vocab_size + 1, EMBEDDING_DIM))
13
14 for word, i in tokenizer.word_index.items():
15     embedding_vector = embedding_index.get(word)
16     if i < vocab_size:
17         if embedding_vector is not None:
18             # Palabras no encontradas en el embedding index serán todo ceros
19             embedding_matrix[i] = embedding_vector
```

Por último, para implementar una red neuronal de clasificación binaria precisábamos que las etiquetas o *targets* tuvieran valores numéricos. Por ello, asignamos el valor 1 a los textos humanos y el valor 0 a aquellos generados por inteligencia artificial.

```
1 df['Tipo texto'] = df['Tipo texto'].map({'Humano': 1, 'IA': 0})
```

## 2. ESTRATEGIA DE CLASIFICACIÓN

Como solución al problema de identificación de textos no humanos proponemos tres algoritmos, cada uno de ellos basado en un marco teórico distinto. Con ello, pretendemos atacar el problema desde distintas perspectivas, ya que de esta manera cabría esperar que los puntos críticos donde uno de los métodos fallara no coincidieran con los de los otros dos. En la Sección 4.2 se compararán los resultados obtenidos por los tres métodos, lo cual proporcionará mayor robustez a la hora de la toma de decisión.

Además, en todos los casos se ha empleado nuestro propio *dataset* como conjunto de entrenamiento y validación. En el Capítulo 3 se validarán los modelos con el conjunto de frases proporcionadas por Accenture para así asegurarnos de que nuestros propios datos no han generado sesgo.

### 2.1. ÁRBOL DE DECISIÓN

Los árboles de decisión son una técnica de aprendizaje supervisado. Se suelen utilizar en problemas de clasificación y por eso la hemos elegido. A partir de un conjunto de ejemplos de entrenamiento, se construye un árbol de decisión donde cada nodo pregunta por el valor de una variable. Así conseguimos un árbol para clasificar los nuevos textos.

La principal ventaja de este método es que es muy interpretable y puede ser analizado por un experto en el campo para entender y explicar la clasificación, siempre y cuando el árbol resultante no sea demasiado grande.

Para realizar esta parte, utilizamos de nuevo la librería **Scikit-learn (sklearn)**, en concreto `tree.DecisionTreeClassifier`. Nosotros hemos utilizado los valores por defecto porque parecían funcionar adecuadamente. Para generar el modelo utilizamos la función `fit()` y para predecir los resultados una vez entrenados usamos `predict`.

```
1 from sklearn import tree
2 import numpy as np
3
4 # Empezamos con la bolsa de palabras
5 cv_matrix_train = cv.fit_transform(train_corpus)
6 cv_matrix_test = cv.transform(test_corpus)
7
8 # Creamos el clasificador con los valores por defecto
9 tree_classifier = tree.DecisionTreeClassifier()
10 tree_classifier.fit(cv_matrix_train, source_train)
11
12
13 tree_train_predictions = tree_classifier.predict(cv_matrix_train)
14 tree_test_predictions = tree_classifier.predict(cv_matrix_test)
```

Para poder observar las palabras que utiliza el árbol para clasificar hemos utilizado la función `print_top20_features_in_trees`.

```

1 def print_top20_features_in_trees(vectorizer, clf):
2     """Prints features with the highest coefficient values"""
3     feature_names = vectorizer.get_feature_names_out()
4
5     top20 = np.argsort(clf.feature_importances_) [-20:]
6     reversed_top = top20[::-1]
7     print("Top 20 features in the tree\n")
8     print("%s" % ( " / ".join(feature_names[j] for j in reversed_top)))

```

Para visualizar el árbol de decisión que está utilizando nuestro modelo, utilizamos la función `tree.plot_tree` de la librería **Scikit-learn (sklearn)**. La Figura 2.1 muestra la representación obtenida.

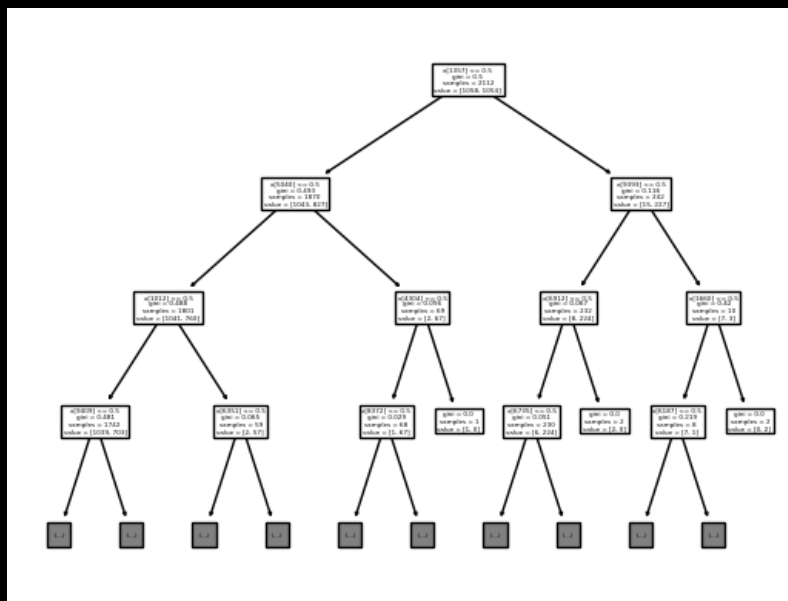


Figura 2.1: Árbol de decisión obtenido.

En este caso, probamos tanto la representación con *bolsa de palabras* como la representación *TF-IDF*, para luego poder compararlas.

## 2.2. NAÏVE BAYES

El clasificador Naïve Bayes es un clasificador probabilístico muy simple y muy poderoso. El clasificador asume que todas las variables son independientes entre sí. En este caso, que la ocurrencia de una palabra en un texto es independiente del resto de palabras. Pese a esta gran simplificación el clasificador funciona muy bien.

El clasificador aprende cómo de frecuente es cada clase y cómo de frecuente es que una palabra ocurra en cada clase. Cuando tiene que estimar un texto nuevo, estima la probabilidad de que pertenezca cada una de las clases y elige la más alta. Hay diferentes tipo de clasificadores Naïve Bayes, pero nosotros vamos a utilizar el NB multinomial.

La razón de esta decisión es que este es el clasificador que se utiliza cuando las variables son enteras (como el número de apariciones de cada palabra en el documento), aunque también acepta datos con decimales (como el valor *TF-IDF*).

Para trabajar con este modelo vamos a utilizar la función `MultinomialNB` que se encuentra en la librería **Scikit-learn (sklearn)**. Con la función `fit` entrenamos el modelo, y

luego utilizamos la función `predict` para generar respuestas nuevas.

```

1 mnb_classifier = MultinomialNB()
2
3
4 cv_matrix_test = cv.transform(test_corpus)
5
6 mnb_classifier.fit(cv_matrix_train, source_train)
7
8 mnb_train_predictions = mnb_classifier.predict(cv_matrix_train)
9 mnb_test_predictions = mnb_classifier.predict(cv_matrix_test)

```

Para poder observar las palabras que más ha tenido en cuenta a la hora de tomar la decisión, utilizamos la función `print_top20_features_per_class_in_NB`.

```

1 def print_top20_features_per_class_in_NB(vectorizer, clf, class_labels):
2     """Prints features with the highest coefficient values, per class"""
3     feature_names = vectorizer.get_feature_names_out()
4     print("Top 20 features per class\n")
5     for i, class_label in enumerate(class_labels):
6         top20 = np.argsort(clf.feature_log_prob_[i])[-20:]
7         reversed_top = top20[::-1]
8
9         print("%s: %s" % (class_label,
10             " / ".join(feature_names[j] for j in reversed_top)), '\n')

```

También en este caso, probamos tanto la representación con *bolsa de palabras* como la representación *TF-IDF*, para luego poder compararlas.

## 2.3. RED NEURONAL

Hemos implementado una red neuronal sencilla haciendo uso de las librerías **Keras** y **Tensorflow**. Como era de esperar, la principal dificultad encontrada fue la elección de los hiperparámetros, además del tipo y número de capas. Como mencionamos en la Sección 1.3.2, la primera capa del modelo es una de tipo *Embedding*. Ajustamos su parámetro `weights=[embedding_matrix]` para inicializar sus pesos según el *Word Embedding* español pre-entrenado utilizado. También ajustamos su parámetro `trainable=False` para mejorar la eficiencia durante el entrenamiento.

Como nos enfrentábamos a un problema de procesamiento del lenguaje natural, decidimos utilizar también una capa recurrente *Long Short-Term Memory (LSTM)*. De esta manera, nuestro modelo podría tratar los textos como secuencias de palabras, donde el orden es relevante, y *recordar* información importante tanto a corto como a largo plazo.

Añadimos también una capa densa interna de 50 neuronas con función de activación rectificadora (*ReLU*). Para combatir el sobreaprendizaje, incluimos posteriormente también dos capas de *Dropout* con probabilidad 0,2, antes y después de la capa densa mencionada.

Finalmente, añadimos una capa densa de salida, con una única neurona y función de activación logística o `sigmoid`. El valor de salida representa, por tanto, la probabilidad de que el texto de entrada haya sido escrito por un humano. Es decir, dado un cierto texto, valores cercanos a 1 representan una alta confianza de la red en que este ha sido redactado por una persona, mientras que valores cercanos a 0 representan lo contrario.

El código resultante se muestra a continuación.

```

1 from keras.models import Sequential
2 from keras.layers import LSTM, Dense, Embedding, Dropout
3 from keras.optimizers import Adam
4
5 # Creación del modelo de red neuronal
6 model = Sequential([
7     Embedding(input_dim=vocab_size + 1,
8               output_dim=EMBEDDING_DIM,
9               weights=[embedding_matrix],
10              input_length=MAX_TEXT_LENGTH,
11              trainable=False),
12     LSTM(EMBEDDING_DIM, batch_input_shape=(None, MAX_TEXT_LENGTH, EMBEDDING_DIM
13       )),
14     Dropout(0.2),
15     Dense(50, activation='relu'),
16     Dropout(0.2),
17     Dense(1, activation='sigmoid')
18 ])
19 # Compilación del modelo
20 model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.01),
               metrics=['accuracy'])

```

Se puede observar que la función de pérdida utilizada para la compilación es la entropía cruzada binaria, la más recomendada para un problema de clasificación binaria con una única neurona de salida y activación logística. Además, elegimos el algoritmo de descenso de gradiente estocástico **Adam** con tasa de aprendizaje 0,01. La optimización de este hiperparámetro fue la más relevante. Conseguimos una mejora de la exactitud (*accuracy*) sobre datos *nuevos* (no utilizados durante el entrenamiento) del 75%-77% al 88%-91% al aumentarlo de 0,001 a 0,01. En la Figura 2.2 se muestra la especificación del modelo de red neuronal empleado.

Model: "sequential"		
Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 40, 300)	3031500
lstm (LSTM)	(None, 300)	721200
dropout (Dropout)	(None, 300)	0
dense (Dense)	(None, 50)	15050
dropout_1 (Dropout)	(None, 50)	0
dense_1 (Dense)	(None, 1)	51
Total params: 3,767,801		
Trainable params: 736,301		
Non-trainable params: 3,031,500		

Figura 2.2: Especificación del modelo de red neuronal utilizado.

Por último, lo entrenamos con el *dataset TRAIN.csv* preprocesado según lo explicado en la Sección 1.3.2. Utilizamos el 80% de los textos para entrenamiento y el 20% para validación.

Probamos diversas cantidades distintas de épocas de aprendizaje, y concluimos que 20 son suficientes para alcanzar el máximo de exactitud en entrenamiento y validación. Por tanto, no utilizamos un valor mayor para evitar la aparición de sobreaprendizaje.

```
1 from sklearn.model_selection import train_test_split
2
3 frase_train, frase_test, tipo_train, tipo_test = train_test_split(data, df['
    Tipo texto'], test_size=0.2, random_state=RANDOM_STATE)
4
5 history = model.fit(frase_train, tipo_train, epochs=20, batch_size=64,
    validation_data=(frase_test, tipo_test))
```



# RESULTADOS



### 3. EVALUACIÓN Y VALIDACIÓN

Para el proceso de evaluación tomamos la decisión de usar como conjunto de test de los algoritmos aquellos textos que Accenture proporcionó para el concurso. Esta decisión se fundamenta en dos razones: la primera, que son el conjunto de datos que más similitud tendrán con el tipo de textos que la propia empresa empleará para ensayar con los modelos; la segunda se justifica por permitir cuantificar de manera imparcial su precisión, es decir, proporcionan un conjunto de textos de procedencia y generación independiente de los que usamos para el entrenamiento y validación por lo que no cabría esperar correlaciones entre ellos.

De manera paralela, a medida que desarrollábamos nuestros modelos, dividimos el *dataset* en dos conjuntos: *train* y *test*, y hemos utilizado este último para realizar comprobaciones sobre la *accuracy* de nuestros modelos. Hemos decidido no incluirlo en la memoria porque considerábamos más relevante los resultados que hemos obtenido del *dataset* de Accenture. Los hemos dejado en el `Jupyter Notebook`, pero en general hemos conseguido una *accuracy* de más del 90 % en casi todos los modelos, y cercana al 100 % en varios de ellos.

#### 3.1. ÁRBOL DE DECISIÓN

Con el objetivo de visualizar y evaluar el desempeño de los modelos de clasificación, recurrimos al uso de la matriz de confusión. Este instrumento nos permitió una representación clara de la efectividad de los modelos. Además, para cuantificar la precisión de las predicciones, se empleó la función `np.mean` de la biblioteca **NumPy**, que proporcionó un cálculo directo del porcentaje de aciertos en cada prueba.

Mediante el uso de la función `predict`, se logró alcanzar una tasa de éxito aproximada del 75 % en la clasificación, tanto utilizando la técnica de Bolsa de Palabras como la de TF-IDF, para ambos modelos de clasificación.

```
1 np.mean(tree_test_predictions == type_test_accenture)
```

Asimismo, se utilizaron matrices de confusión para poder detectar falsos positivos y falsos negativos en nuestro modelo. Para ello se definió una función como mostramos a continuación.

```
1 def debug_print(predictions, target, target_names):
2     print(classification_report(target, predictions, target_names=target_names)
3         )
4     conf_matrix = confusion_matrix(target, predictions)
5     conf_matrix_df = pd.DataFrame(conf_matrix, index=target_names, columns=
6         target_names)
7     plt.figure(figsize=(15, 10))
8     sn.heatmap(conf_matrix_df, annot=True, vmin=0, vmax=conf_matrix.max(), fmt=
9         'd', cmap="YlGnBu")
10    plt.yticks(rotation=0)
11    plt.xticks(rotation=90)
```

En las figuras 3.1 y 3.2 observamos las matrices de confusión para el método de bolsa de



palabras y el método de TF-IDF. El eje de ordenadas representa el tipo real de las frases y el eje de abscisas la predicción hecha por el árbol.

Además, para poder entender más los falsos negativos y falsos positivos del modelo se definió una función para leer las predicciones erróneas como mostramos a continuación.

```
1 def get_wrong_predictions(predictions, target, data):
2     wrong = (predictions != target)
3     return data[wrong], target[wrong]
4
5 original_values = ['Humano', 'IA']
6
7 #Vamos a sacar algunas predicciones en las que nuestro modelo se ha equivocado.
8 d, t = get_wrong_predictions(tree_test_predictions, source_test, sentence_test)
9
10 d1, d0 = d[t=='Humano'], d[t=='IA']
11
12 if(len(d1)>0): print("Se esperaba: ", original_values[1], " y se predijo: ",
13                    original_values[0], " para:\n ", d1[d1.index[0]])
14 print("\n")
15 if(len(d0)>0): print("Se esperaba: ", original_values[0], " y se predijo: ",
16                    original_values[1], " para:\n ", d0[d0.index[0]])
```

Entre algunos ejemplos en los que se obtuvieron predicciones erróneas para los datos de Accenture ejemplificamos para el método de bolsa de palabras:

- Se esperaba IA y se predijo Humano para “El campo estaba en un valle rodeado de montañas nevadas. Era muy colorido porque había plantaciones de muchos tipos de flores y, por eso, olía muy bien. A la izquierda del camino que cruzaba este extenso lugar había una cabaña que tenía una chimenea que echaba humo. Al lado de la cabaña había dos

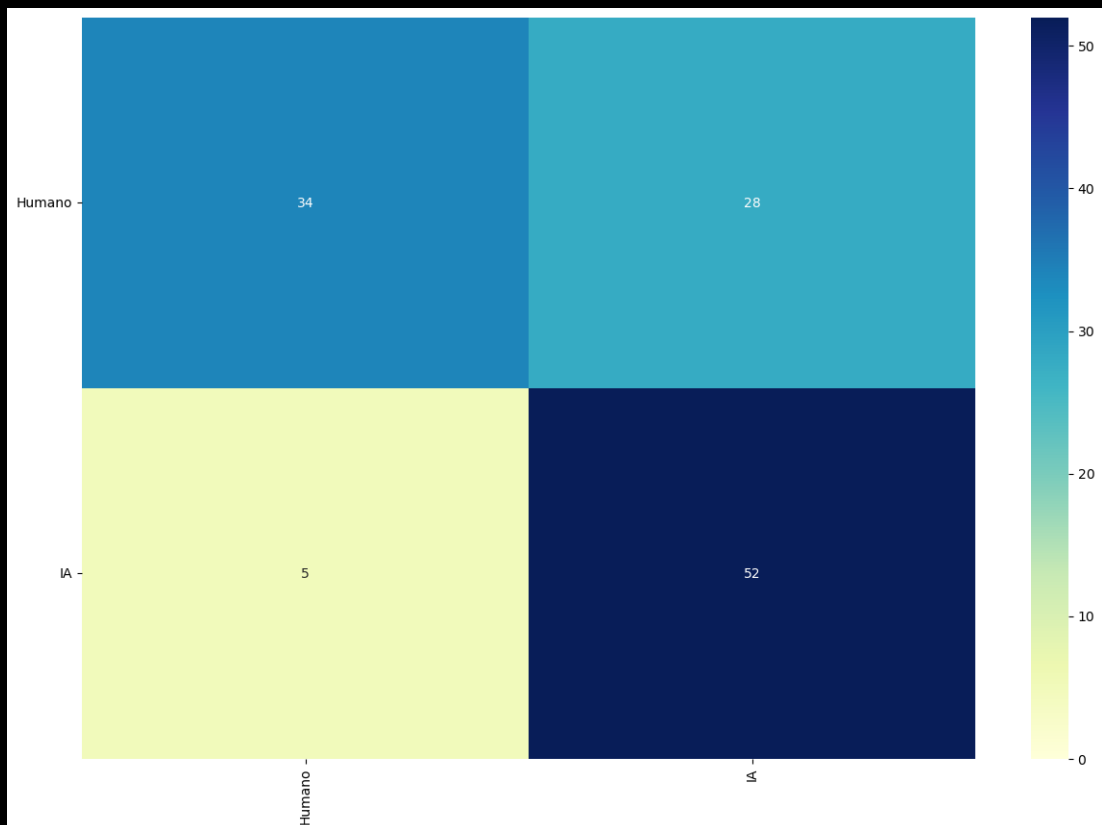


Figura 3.1: Matriz de confusión para el árbol de decisión con el método de bolsa de palabras.

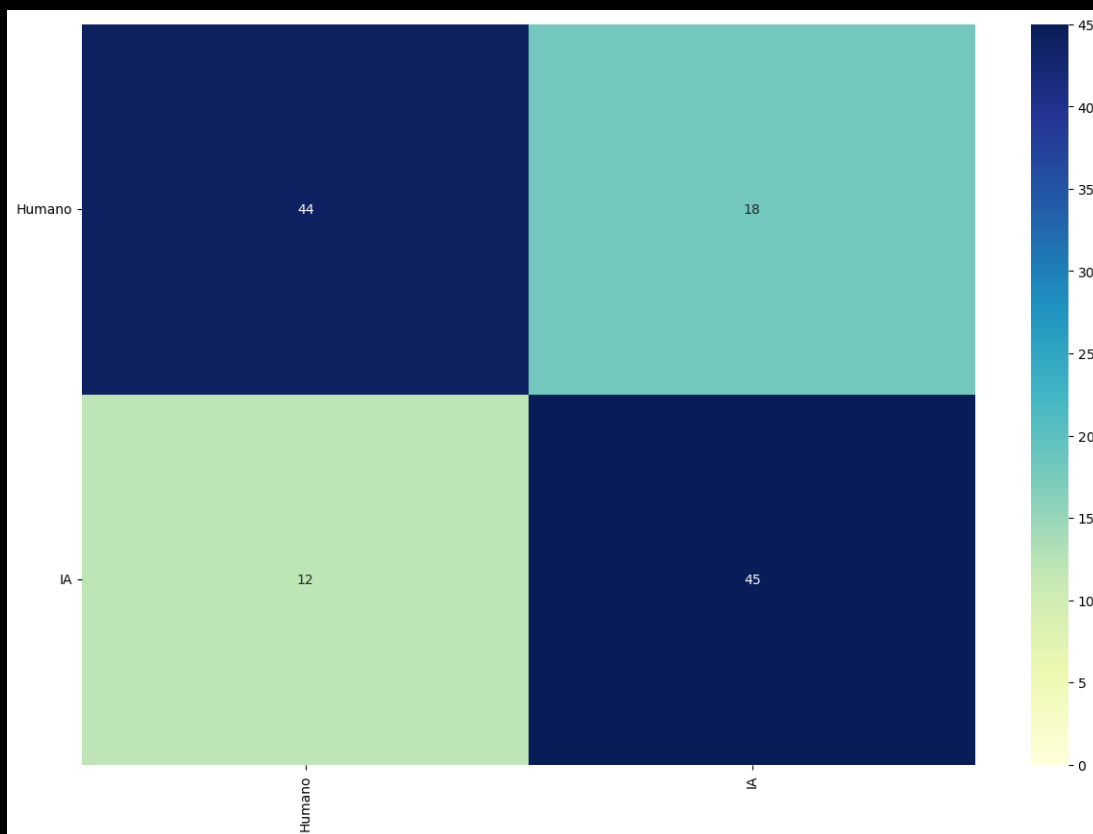


Figura 3.2: Matriz de confusión para el árbol de decisión con el método TF-IDF.

caballos que tomaban agua de un pequeño arroyo. No había personas fuera de la casa, pero la puerta estaba abierta”

- Se esperaba Humano y se predijo IA para “El último habitante de un pequeño pueblo en el oeste de Estados Unidos era un anciano que había vivido allí durante décadas. La población había disminuido gradualmente a medida que los jóvenes se marchaban en busca de oportunidades en la ciudad. Pero cuando llegó un desconocido con una oferta para comprar la tierra, el anciano se aferró a su hogar y a los recuerdos que contenía. A medida que luchaba contra los intentos de desalojo, comenzó a desentrañar la historia oculta de su pueblo y a entender por qué era tan importante que nunca se rindiera”

### 3.2. NAÏVE BAYES

De manera completamente análoga a la sección anterior se obtuvieron los porcentajes de acierto en predicciones con el uso de la función `np.mean` y las matrices de confusión para el método de Naïve Bayes para ambas representaciones de bolsa de palabras y TF-IDF.

Para ambos métodos, el porcentaje de acierto es superior al 83 % y en las figuras 3.3 y 3.4 podemos ver las matrices de confusión ejecutadas con la función `debug_print`.

Entre algunos de las predicciones erróneas para los datos de Accenture destacamos las siguientes:

- Se esperaba IA y se predijo Humano para “El cazador había perdido todas sus herramientas de caza, pero se dio cuenta de que no las necesitaba porque podía alimentarse de las frutas que crecían en los árboles. Se puso muy contento con esta determinación y decidió que,

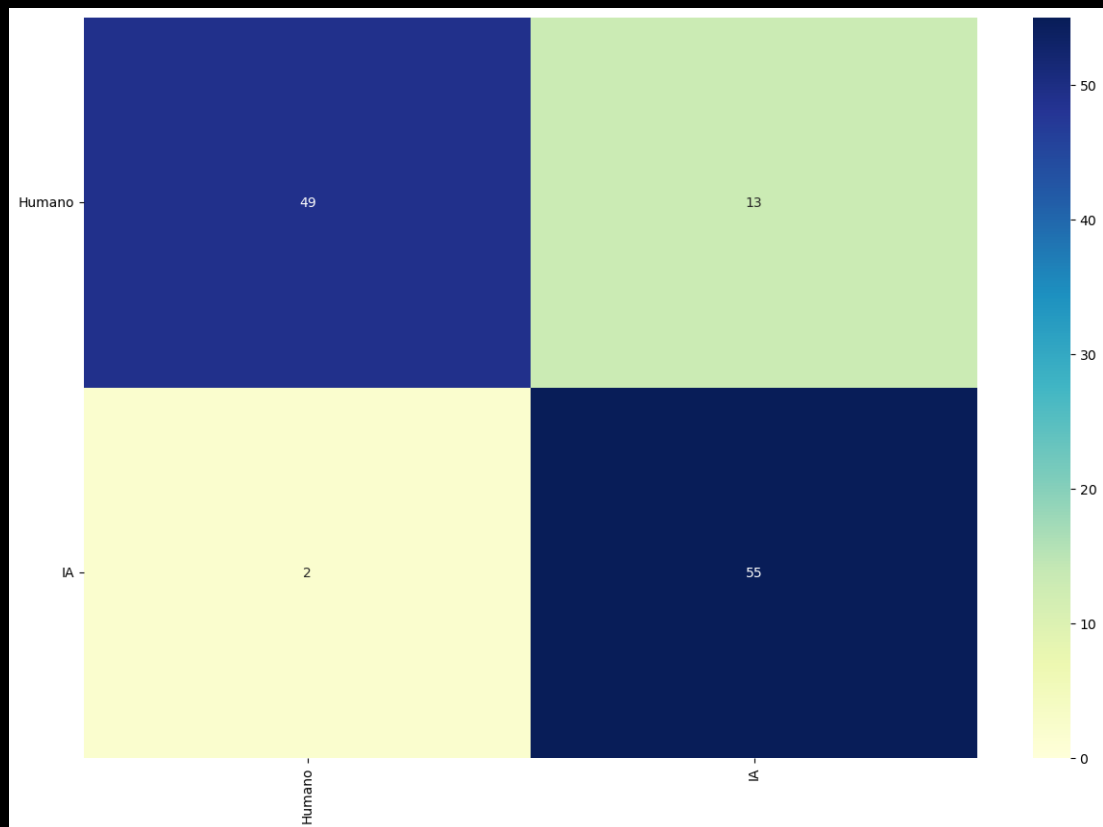


Figura 3.3: Matriz de confusión para Naïve Bayes con el método de bolsa de palabras.

cuando llegara a la ciudad, les diría a sus amigos que no practicaría esta actividad nunca más, porque había aprendido a respetar a la naturaleza.”

- Se esperaba Humano y se predijo IA para “En una ciudad futurista donde la tecnología había avanzado más allá de la imaginación, un joven científico trabajaba incansablemente en su laboratorio subterráneo. Había descubierto una forma revolucionaria de viajar en el tiempo, pero su invención era peligrosa y planteaba preguntas éticas fundamentales. Mientras los poderes gubernamentales y corporativos intentaban apoderarse de su descubrimiento, él se debatía entre la promesa de cambiar el pasado y el temor de alterar el curso de la historia de la humanidad de manera irreversible.”

### 3.3. RED NEURONAL

La red neuronal es el algoritmo que mayor precisión alcanza de entre todos los métodos, oscilando ésta en torno al 91% sobre datos nuevos. Se hizo uso de otras métricas de medición de rendimiento como pueden ser la matriz de confusión y el valor-F1. La matriz de confusión de la red que hemos entrenado se muestra en la Figura 3.5. A simple vista es claro la excelente labor de clasificación que lleva a cabo, puesto que los errores que comete son despreciables frente a los aciertos que se observan en la diagonal principal.

Además, empleando la función `f1_score` del paquete `sklearn.metrics` se obtuvo que el valor-F1 de nuestra red coincide exactamente con la precisión calculada, es decir, que se encuentra entorno al 91%. Nótese que no debe suceder necesariamente que ambas estadísticas de rendimiento coincidan.

Una desventaja de este método frente a los otros presentados es que, pese a la eficacia de

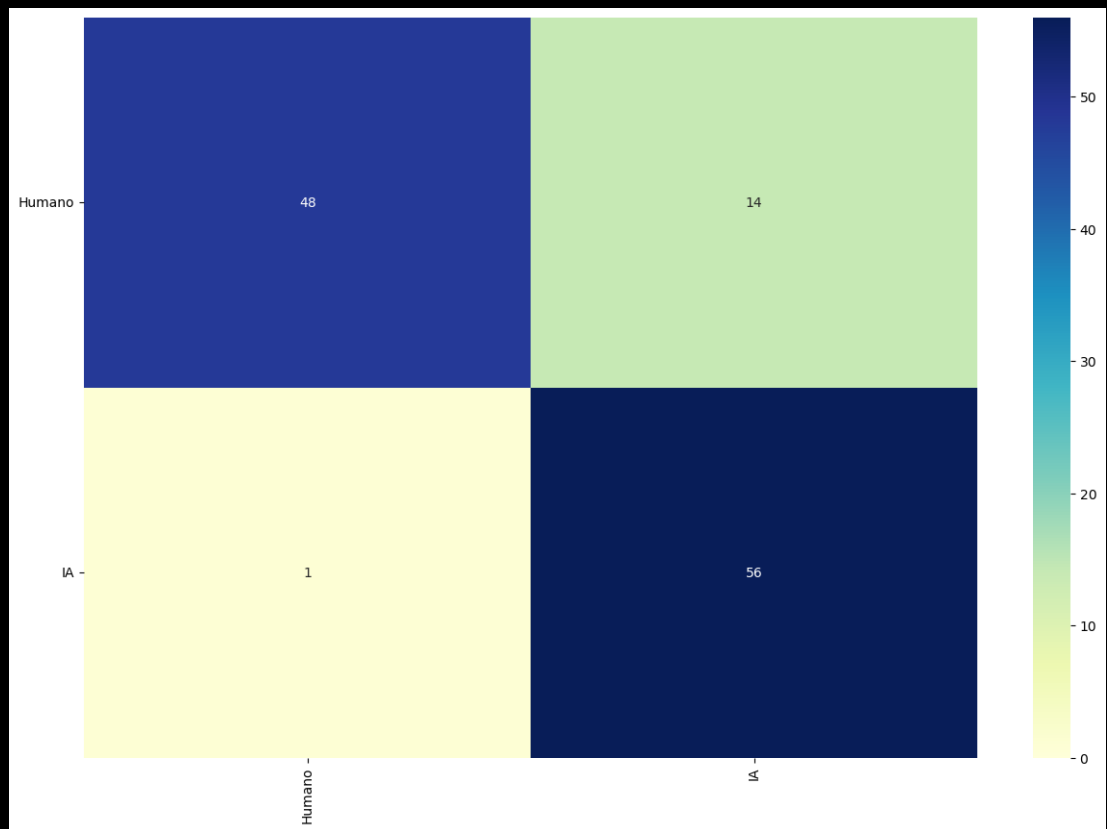


Figura 3.4: Matriz de confusión para Naïve-Bayes con el método TF-IDF.

su labor, no podemos ahondar en los patrones y características de los textos que la red emplea para clasificar. Trabajamos con ella como si de una caja negra se tratase, por lo que no existe realmente una justificación del porqué de su excelente funcionamiento más allá de los resultados que proporciona.

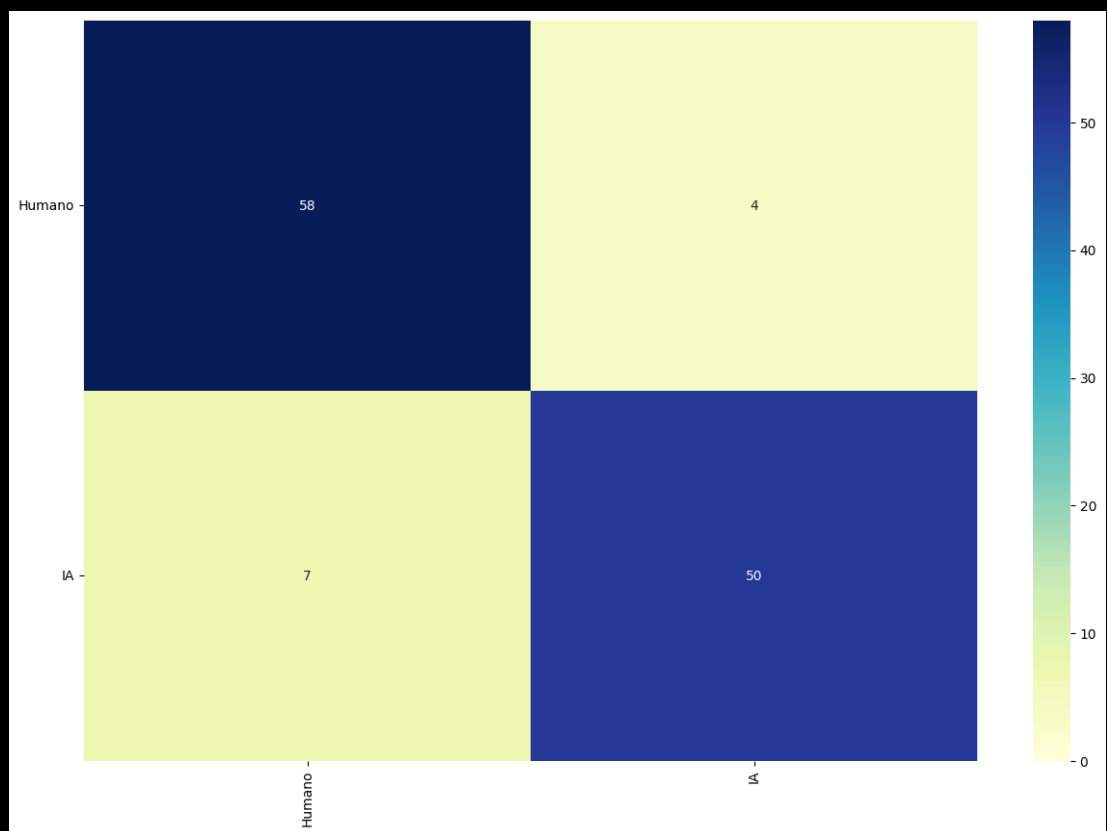


Figura 3.5: Matriz de confusión para la red neuronal entrenada sobre los datos de Accenture.



# CONCLUSIONES



## 4. OBSERVACIONES

### 4.1. DEPENDENCIA DE LOS RESULTADOS Y LOS DATOS

Es importante recalcar que todos los resultados de la sección anterior dependen fuertemente del subconjunto de textos humanos que tuvimos que aleatoriamente seleccionar de entre los millones que poseía el corpus. Principalmente esto se debe a la extensión de los extractos, ya que algunos tenían una longitud de unas pocas palabras, mientras que otros estaban comprendidos por un par de oraciones; por otra parte, los temas de los que trataban no eran independientes entre sí, luego cabe la posibilidad de que la mezcla aleatoria haya seleccionado una mayor proporción de textos de una longitud o temática o estilo dados. Esto repercute gravemente en cualquier algoritmo de clasificación como los que estamos tratando, ya que podrían interpretar tales sesgos como rasgos característicos de textos generados por humanos, cuando realmente han sido resultado de un proceso de selección desafortunado.

Es por ello, que frente a la falta de tiempo no hemos tenido ocasión de repetir los resultados para distintos muestreos del gigantesco dataset humano. La repetibilidad de unas mismas estadísticas de precisión, independientemente de la permutación elegida nos aseguraría que este problema no tiene efecto significativo en los resultados. En caso de que se observe una reducción o aumento considerables en las métricas, sería oportuno realizar un barrido de distintos *set* de *train* para así optimizar aquellos que supongan una representación más fidedigna de textos humanos.

Otra cuestión relevante, es la sorprendente potencia de la red neuronal y del algoritmo de Naïve Bayes para detectar textos generados artificialmente si consideramos su reducido tamaño en cuanto a parámetros se refiere. Ello nos lleva a postular que, en caso de conseguir incrementar considerablemente el conjunto de entrenamiento de los textos generados por inteligencia artificial, de manera que su tamaño sea comparable con el del corpus humano que poseemos, podríamos lograr incluso mejores resultados.

### 4.2. COMPARACIÓN DE LOS MÉTODOS

Una vez presentados los resultados de cada algoritmo, el ranking al que llegamos en términos de rendimiento de clasificación es, de mejor a peor:

1. Red neuronal
2. Naïve Bayes
3. Árbol de decisión

No obstante, no sólo estamos interesados en un único método de clasificación universal, sino que opinamos que alimentar el texto prueba a distintos algoritmos proporciona una mayor validez y credibilidad. Es por ello que creemos conveniente presentar todas las soluciones que se nos ocurrieron, sin necesidad de excluir ninguna, ya que cada una de ellas arroja distintas perspectivas desde la que atacar el problema. Por ejemplo, si bien es cierto que la red neuronal es el mecanismo de clasificación más potente, no nos proporciona información ninguna de cómo procede a dicha clasificación; por el contrario, el árbol de clasificación, de peor eficiencia, permite obtener un esquema de las decisiones que toma el algoritmo.

Estos motivos fueron los que justificaron la creación de la interfaz, ya que de esta manera podemos acceder rápidamente a los modelos, pudiendo así compararlos. Por ejemplo, si un modelo falla en un texto, ¿ocurrirá lo mismo con los otros?; si un modelo tiende a clasificar como IA textos más cortos, ¿ocurrirá lo mismo con los otros? En caso afirmativo, por ejemplo, confirmaría un sesgo en los datos. Es por ello, que esta herramienta permite investigar de forma más eficiente en las características de cada algoritmo con el fin de ponerlas al límite.

### 4.3. PRINCIPALES DIFICULTADES ENCONTRADAS

Al igual que sucede en la mayoría de situaciones con este tipo de problemas, el primer obstáculo que se encuentra es el de una base extensa y variada de datos. Además de la escasez existente de tales corpus en el idioma castellano, se le debe sumar la extrema diversidad en cuanto a contenido, extensión y estilo en cuanto a los textos se refiere. En el caso de la ampliación de textos generados por IA, se tuvo que realizar a mano por los miembros del grupo, lo cual explica su escasez en comparación con los humanos; este es uno de los puntos que debido a los recursos que disponíamos no pudimos solucionar.

Posteriormente, nos tuvimos que enfrentar a la manera de codificar estos datos según las especificaciones de cada algoritmo. Por ejemplo, tuvimos que solucionar el eliminar caracteres extraños de los textos en el caso de los árboles de decisión, o el encontrar un *embedding* suficientemente bueno para alimentar los datos de entrenamiento a la red neuronal. Además, a la hora de crear el programa de la interfaz, hubo que fusionar todos los modelos en un mismo fichero, lo cual supuso otro reto.

Otro aspecto ajeno a la propia naturaleza del problema es el hecho de que cada uno de los integrantes del equipo utiliza un ordenador con un sistema operativo diferente, lo que nos ha complicado el compartir información por la diferencia de formato en algunos aspectos; por ejemplo, esto sucedía con la ruta de los archivos, ya que era imposible garantizar que todos estructurasen sus carpetas locales de la misma manera. Es por esto que nos hemos terminado decantando por utilizar herramientas online como *Google Collaboratory*. Una desventaja de esta decisión fue que esta aplicación no funciona si más de una persona accede de forma simultánea a un mismo fichero y lo corre, lo cual ralentizaba la cooperación en directo.



## 5. CONOCIMIENTOS ADQUIRIDOS

Durante el desarrollo del proyecto, el equipo ha experimentado una mejora de habilidades técnicas y teóricas. El dominio del lenguaje de programación Python se ha fortalecido significativamente, particularmente en lo que respecta a la gestión y evaluación de extensas bases de datos. Paralelamente, hemos consolidado nuestra comprensión en el campo de la inteligencia artificial.

La familiarización con herramientas de desarrollo colaborativo y computación en la nube, como Google Colab, ha sido otro de las competencias adquiridas, proporcionando una plataforma eficiente para la implementación de nuestros modelos. Además, el proyecto ha sido especialmente enriquecedor para algunos miembros del equipo que, hasta ahora, no habían interactuado con interfaces gráficas, dándoles de una buena experiencia para futuros desafíos tecnológicos.

Considerando todo lo expuesto en esta memoria, creemos haber obtenido una solución satisfactoria al problema.

Tras alimentar esta misma memoria a través del modelo de red neuronal entrenado, obtenemos que el 100 % de los extractos de texto analizados son clasificados como tipo “ Humano”. Con ello, el tribunal puede estar seguro a un 91 % de *accuracy* de que no hemos usado *ChatGPT* para redactarla. Pero quién sabe, igual desde el principio esta memoria ha sido escrita por una máquina.



# REFERENCIAS



## 6. REFERENCIAS

1. Traditional Methods for Text Data
2. NLP Town
3. Spanish Billion Words Corpus and Embeddings: CRISTIAN CARDELLINO (March 2016)
4. Compilation of Large Spanish Unannotated Corpora: JOSÉ CAÑETE, (May 2019)



# APÉNDICE



## 7. TUTORIAL INTERFAZ GRÁFICA

En la carpeta de archivos se incluye un fichero llamado *Interfaz.ipynb*, en el cual, al ejecutar la última celda, nos muestra un cuadro con el que podremos interactuar. En él, podremos seleccionar una de las tres estrategias de clasificación de textos, junto con las dos formas de representación que hemos implementado (lo que da un total de cinco formas), así como introducir un extracto manualmente en la pantalla.

Comprende de dos botones únicamente, el primero que tras seleccionarlo despliega una lista con los métodos como se muestra en la Figura 7.1; y el segundo, que debe presionarse una vez que previamente hayamos introducido el texto y elegido el método de clasificación.

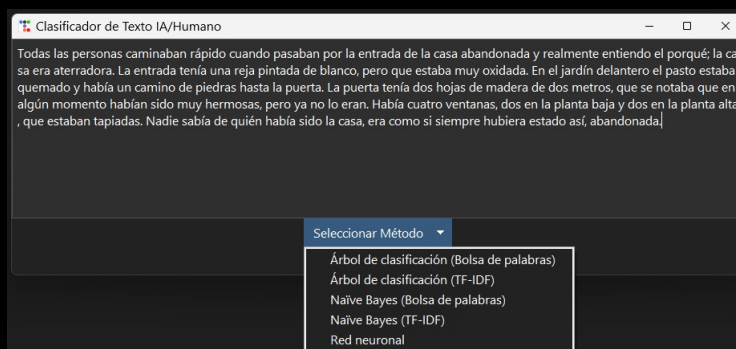


Figura 7.1.

El programa entonces ejecutará una ventana que nos indicará la decisión final de si el texto proporcionado es posiblemente humano o generado artificialmente (ejemplo en la Figura 7.2). La motivación de esta interfaz es facilitar un uso rápido de los algoritmos sin la

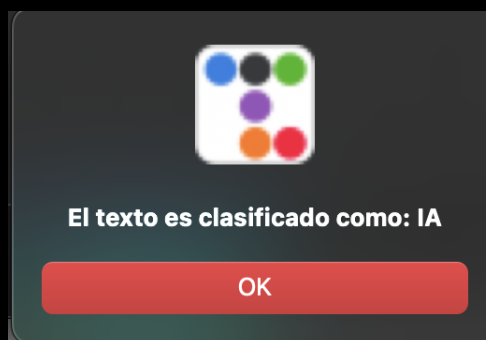


Figura 7.2.

necesidad de correr todos los archivos adjuntos, ni tener que entrenar los modelos de nuevo. Además permite comparar la precisión de cada método y así comprobar los comentarios de la sección de resultados.