



INSTITUTO POLITÉCNICO NACIONAL



ESCUELA SUPERIOR DE CÓMPUTO

CLASIFICACIÓN DE TEXTOS

PROFESOR:

ITURIEL ENRIQUE FLORES ESTRADA

ALUMNO:

ROSALES ONOFRE TANIA

7BM1

FECHA DE ENTREGA: 19 DE ABRIL DE 2023

ÍNDICE

CLASIFICACIÓN DE TEXTOS	1
Introducción	3
Extracción De Datos Y Clases	3
Preprocesamiento.....	3
1. Limpieza De Etiquetas Html	3
2.Normalizar texto	4
Stop words.....	4
3.LEMATIZAR/STEMMING/Stop words	4
Problema al aplicar	5
DEF NORMALIZAR()	5
Resultados función	5
4.Tf-idf	5
entrenamiento	6
Division De Conjuntos De Entrenamiento.....	6
Aplicar Oversampling Al Conjunto De Entrenamiento	6
Aplicación De Modelos	7
Regresión Logística.....	7
resultados	7
Bayes Ingenuo	8
Resultados	8
KNN	9
Red Neuronal Con Capa De “Incrustación”	10
CAPA DE INCRUSTACIÓN DE WOR2VEC	10
Capa De Incrustación Con Tensorflow	13
Resultado.....	14
Comparación De Modelos	14
Conclusiones.....	15

INTRODUCCIÓN

En esta practica se buscó hacer una clasificación mediante 4 modelos: Bayes ingenuo, Regresión logística, Vecinos cercanos -KNN, Red Neuronal con capa de “incrustación” entrenada con el cuerpo de documentos con 2000 observaciones con 4 clases.

Los documentos a utilizar son noticias de la razón y la jornada donde se extrajeron por medio de webscrappin.

EXTRACCIÓN DE DATOS Y CLASES

Para la extracción de datos se hizo web scrapping de noticias de la razón y de noticias de la jornada, al ser extraídos de manera manualmente no se tiene un gran conjunto de datos como lo requiere uno de los objetivos de la practica ya que principalmente al ser de noticieros se encuentra diferencia de clases para ello se tuvo que hacer un análisis de cada una de las categorías de ambos noticieros donde se determinaron 4 clases similares que ayudaría a tener más observaciones en el conjunto de datos .

Para ello primero lo que se hizo fue concatenar ambos archivos CSV que se hizo en el archivo **concatener_files.py** ,donde se hizo una función **concatener_files** donde se hizo la eliminación 11 categorías que no iban a ser de utilidad para esta clasificación además de que se contaban pocas observaciones de las mismas. Para así poder mandarla a llamar en **clases.py** y verificar el total de observaciones de cada una de las clases para así también guardar un archivo llamado **noticias_clases.csv**

Debido al procedimiento de la extracción de datos se eliminaron filas donde se encontraban datos vacíos que aún así contando con categoría no tenia contenido de la noticia para ello se utilizó la siguiente función:

```
df['Contenido'] = df['Contenido'].replace('[\[\]]', '',
regex=True).replace(' ', np.nan)
df = df.dropna(subset=['Contenido'], how='any')
```

PREPROCESAMIENTO

1. LIMPIEZA DE ETIQUETAS HTML

Al ser extraído por medio de web scrapping se obtuvieron etiquetas de html en los datos como los muestra la *Imagen 1*:

```
[<p><em>Ciudad de México.</em>
```

1.Imagen “Ejemplo de etiquetas”

Para poder eliminar estas etiquetas se realizó una función llamada **def clean_html()** que se encuentra en el archivo **preprocesamiento.py** donde se realizo con expresiones regulares los patrones tenían las etiquetas de html y quedo la función como se muestra en la siguiente parte del código:

```
def clean_html(text):

    if text is None or not isinstance(text, str) or not text.strip():
        return ""

    # Eliminar etiqueta que está al principio
    cleaned_text = re.sub('<p><em>', '', text)
    # Eliminar al final etiqueta
    cleaned_text = re.sub('</em>$', '', cleaned_text)
    # Eliminar todas las demás etiquetas HTML
    cleaned_text = re.sub('<[^>]*>', '', cleaned_text)
    cleaned_text = re.sub('\d', '', cleaned_text)
    cleaned_text = re.sub('</p>', '', cleaned_text)
    return cleaned_text
```

2.NORMALIZAR TEXTO

Después de haber eliminado cada una de las etiquetas de html se procedió a normalizar el texto para esto se creo la función llamada **def normalizar(n_text)** donde se eliminaron los signos de puntuación, los acentos esto debido a que no aporta nada en el texto

La eliminación de los signos de puntuación se hizo mediante expresiones regulares mientras que para los acentos se utilizó la *biblioteca Unicode*

Después de haber eliminado, se convirtió el texto en minúsculas esto para el momento de que se quiten las stop words sea más fácil de localizarlas en el texto

STOP WORDS

Para la eliminación de las stop words se utilizo la biblioteca de spacy de las que ya se tienen preestablecidas pero también al notar que en frecuencia de palabras se tenían palabras que se consideraban que no eran tan útiles para la clasificación del texto.

Para ello se creo la siguiente lista y se le asigno al diccionario de palabras que ya tiene spacy:

```
stop_words=['mil','mujer','ciento']
```

3.LEMATIZAR/STEMMING/STOP WORDS

```
# Lematizar y Stemming
tokens = [stemmer.stem(token.lemma_) if token.lemma_ != '-PRON-' else
stemmer.stem(token.text) for token in clean_text if not token.is_stop and
not token.is_punct and not token.is_space]
#Eliminamos las terminaciones el
```

En esta función se aplico lo que primero stemming y después se lematizo esto ya que reduce la complejidad del texto al eliminar sufijos y prefijos, lo que puede hacer que la lematización sea más precisa y eficiente, además de que al vectorizar los tokens ya lematizados y aplicados con stemming pueden ayudar a agrupar palabras relacionadas en la misma raíz.

Al aplicar stemming se tuvieron algunos problemas ya que cambiaba el sufijo como por ejemplo se muestra en la 2.

```
'meter el  
ombrill'
```

Al aparecer esta separación de el al utilizar la función de frecuencia de palabras nos aparece como si fuera una de las más utiliza “el” entonces para hice otra limpieza de datos que no encontraba ya después de hacer la aplicación porque son datos no relevantes para poder realizar clasificación, además de él se encontró también otra palabra que no era relevante.

```
DEF NORMALIZAR()
```

RESULTADOS FUNCIÓN

[illegible]

4.TF-IDF

En el archivo *modelo.py* aplicamos a nuestros datos la función normalizar mencionada anteriormente para obtener los tokens

```
vectorizer = TfidfVectorizer(min_df=5, max_df=0.8, sublinear_tf=True,
use_idf=True)
tfidf_matrix = vectorizer.fit_transform(df['Documento_Normalizado'])
```

Se utilizó la librería de `sklearn.feature_extraction.text`, esto se le aplicará a la columna donde ya está normalizado el texto, se estableció un número mínimo de documentos en los que debe aparecer cada palabra para poder incluirlo, una frecuencia máxima de 80 para que aparezca una palabra para que pueda ser incluida en la matriz de términos frecuentes. También se ajusta después la vectorización

ENTRENAMIENTO

DIVISION DE CONJUNTOS DE ENTRENAMIENTO

```
# Divide los datos en conjunto de entrenamiento y conjunto de prueba
X_train, X_test, y_train, y_test = train_test_split(tfidf_matrix,
df['Categoria'], test_size=0.2, random_state=42)
```

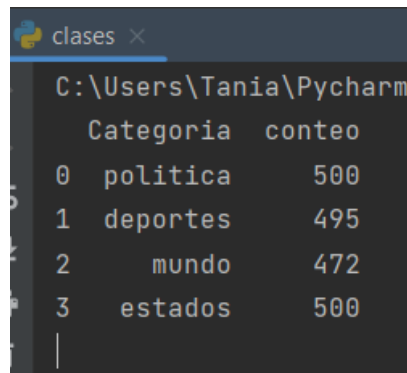
Se dividió el conjunto de entrenamiento en 80% y de prueba 20%

APLICAR OVERSAMPLING AL CONJUNTO DE ENTRENAMIENTO

En este caso al tener una clara desigualdad en la cantidad de observaciones por clase como se muestra en la *4-Imagen "Clases y observaciones"*. Lo cual puede provocar problemas en el proceso de entrenamiento. En particular es probable que el modelo tenga la tendencia a predecir la clase mayoritaria con más frecuencia que las otras, lo que resulta en baja precisión para las clases minoritarias. De hecho antes de realizar el oversampling se podía observar que se predecía más a la clase de estados y política.

Para poder solucionar este problema. Se recurre a la técnica de oversampling (sobremuestreo), para igualar el número de observaciones por clase.

En este caso investigando se me hizo adecuado utilizar SMOTE que lo que hace es generar nuevas muestras y así se puede evitar el riesgo de sobreajustes que puede surgir al simplemente duplicar las observaciones existentes que también es otra técnica de oversampling. Además de que SMOTE puede mejorar la capacidad del modelo para reconocer patrones en las clases minoritarias, lo cual lleva a una mejor precisión general del modelo de conjunto de datos balanceados.



A screenshot of a Jupyter Notebook window titled 'clases'. It displays a table with two columns: 'Categoria' and 'conteo'. The table lists four categories: 'politica' (500), 'deportes' (495), 'mundo' (472), and 'estados' (500).

	Categoria	conteo
0	politica	500
1	deportes	495
2	mundo	472
3	estados	500

4.Imagen"Clases y observaciones"

En este caso se utilizó una biblioteca para la aplicación de esta técnica que fue llamada por la función SMOTE

```
# Aplicar oversampling al conjunto de entrenamiento
sm = SMOTE(random_state=42)
X_train_res, y_train_res = sm.fit_resample(X_train, y_train)
```

APLICACIÓN DE MODELOS

REGRESIÓN LOGÍSTICA

La aplicación de este modelo se encuentra en **modelo.py**

RESULTADOS

```
Precisión Regresión Logística: 0.8954191033138402
Precisión Regresión Logística en el conjunto de prueba: 0.9213197969543148
```

	precision	recall	f1-score	support
deportes	1.00	0.96	0.98	96
estados	0.90	0.90	0.90	106
mundo	0.94	0.96	0.95	96
politica	0.85	0.88	0.86	96
accuracy			0.92	394
macro avg	0.92	0.92	0.92	394
weighted avg	0.92	0.92	0.92	394

5.Imagen "Resultados Regresión Logística"

Los resultados de la regresión logística que se muestran en la *5.Imagen "Resultados Regresión logística"* son bastante buenos. La precisión en el conjunto de entrenamiento es del 89.54%, lo que significa que el modelo clasifica correctamente el 89.54% de las noticias en su categoría correspondiente. En el conjunto de prueba,

la precisión aumenta a 92.13%, lo que significa que el modelo clasifica correctamente el 92.13% de las noticias en su categoría correspondiente.

La precisión para cada categoría es también bastante alta, con un valor máximo de 100% para la categoría "deportes". El valor mínimo es del 85% para la categoría "politica", lo que significa que el modelo tiene dificultades para distinguir entre noticias de política y otras categorías. El promedio ponderado de precisión, que tiene en cuenta el desequilibrio en el número de noticias en cada categoría, es del 92%, lo que indica que el modelo es efectivo para clasificar noticias en general.

BAYES INGENUO

La aplicación de este modelo se encuentra en **modelo.py**

Como se menciono anteriormente para la realización de este modelo se realizo un oversampling y en este caso se realizo por medio de la función que proporciona sklearn que **MultinomialNB()**, esto debido a que investigando se encontró que es efectivo para clasificación de texto.

```
# Naive Bayes con el conjunto de entrenamiento balanceado

clf_nb = MultinomialNB()
clf_nb.fit(X_train_res, y_train_res)
k = 5 # número de pliegues en k-fold cross-validation
kf = KFold(n_splits=k)
scores = cross_val_score(clf_nb, X_train_res, y_train_res, cv=kf)
print("Precisión Naive Bayes:", np.mean(scores))

# Evaluar el modelo en el conjunto de prueba
y_pred_nb = clf_nb.predict(X_test)
precision_nb = accuracy_score(y_test, y_pred_nb)
print("Precisión Naive Bayes en el conjunto de prueba:", precision_nb)
print(classification_report(y_test, y_pred_nb))
```

RESULTADOS


```
Precisión Naive Bayes: 0.8700665061346176
Precisión Naive Bayes en el conjunto de prueba: 0.883248730964467
```

	precision	recall	f1-score	support
deportes	1.00	0.92	0.96	96
estados	0.87	0.83	0.85	106
mundo	0.94	0.95	0.94	96
politica	0.75	0.84	0.79	96
accuracy			0.88	394
macro avg	0.89	0.88	0.89	394
weighted avg	0.89	0.88	0.89	394

6.Imagen "Resultados Bayes Ingenuo"

La precisión promedio de la validación como se muestra en 6.Imagen "Resultados Bayes Ingenuo" cruzada k-fold fue del 87%, lo que significa que el modelo acertó correctamente en el 87% de las clasificaciones.

La precisión en el conjunto de prueba fue del 88%, lo que indica que el modelo pudo generalizar bien a datos que no había visto antes.

En cuanto a las métricas de precisión, recall y f1-score por categoría, se puede ver que el modelo tuvo un desempeño muy bueno en la categoría de deportes, con una precisión del 100% y un recall del 92%. También tuvo un buen desempeño en la categoría de mundo, con una precisión del 94% y un recall del 95%.

En la categoría de estados, el modelo tuvo una precisión del 87% y un recall del 83%, lo que indica que tuvo dificultades para distinguir entre las noticias de esta categoría y las de la categoría de política. En la categoría de política, el modelo tuvo una precisión del 75% y un recall del 84%, lo que indica que tuvo dificultades para clasificar correctamente las noticias de esta categoría.

En general, se puede decir que el modelo tuvo un rendimiento bastante bueno en la tarea de clasificación de noticias por categoría, aunque puede haber oportunidades para mejorar en la clasificación de las categorías de estados y política.

KNN

La aplicación de este modelo se encuentra en **modelo.py**

```
Precisión KNN: 0.6794786530596644
Precisión KNN en el conjunto de prueba: 0.7030456852791879
```

	precision	recall	f1-score	support
deportes	1.00	0.81	0.90	96
estados	0.87	0.69	0.77	106
mundo	0.48	0.99	0.65	96
politica	0.89	0.32	0.47	96
accuracy			0.70	394
macro avg	0.81	0.70	0.70	394
weighted avg	0.81	0.70	0.70	394

7.Imagen "Resultados KNN"

Estos resultados que se muestran en la 7.Imagen "Resultados KNN" tiene una precisión del 67.94% en la clasificación de noticias por categoría. Además, la precisión del modelo en el conjunto de prueba es del 70.30%, lo que sugiere que el modelo está generalizando bien a datos nuevos. Sin embargo, se puede observar que la precisión en la categoría de "mundo" es baja en comparación con las otras categorías.

RED NEURONAL CON CAPA DE "INCRUSTACIÓN"

CAPA DE INCRUSTACIÓN DE WOR2VEC

La aplicación de este modelo se realiza en el archivo **embading.py**

Para la creación de capa de incrustación por medio de wor2vec, para realizar este modelo se utilizaron funciones que proporcionan la biblioteca de gensim y tensorflow.

Así mismo se retoma la función creada en el archivo preprocesamiento def normalizar() para tener el texto limpio.

Se crea y se entrena el modelo de wor2vec utilizando ya las noticias normalizadas, para poder crear vectores de palabras que representen el contenido de las noticias

```
# Entrenamos el modelo Word2Vec
model = Word2Vec(sentences, vector_size=400, window=5, min_count=1,
workers=4)
```

En este caso se la dimensión de vectores que se generara por cada palabra será de 400, que será la entrada de capa de entrada, window que se utilizará para predecir la siguiente palabra por ejemplo el modelo utilizará las 5 palabras anteriores y las 5 palabras posteriores a la palabra objetivo para predecirla en este caso la frecuencia minia que debe tener una palabra será de 1

Después se creará el mapa de característica con un tamaño de 400, y para realizar el hot -encoding se realizo lo siguiente :

```
#Convertimos etiquetas en formato numero
mlb = MultiLabelBinarizer()
#Transformas cada etiqueta de categoria en una lista de una sola etiqueta
y = mlb.fit_transform(df['Categoria'].apply(lambda x: [x]))
#Se obtiene la etiqueta de categoria numerica correspondiente a cada uno
de los ejemplos de entrenamiento
y = np.argmax(y, axis=1)
```

Ya después teniendo el mapa de características en X y en y las etiquetas en numero entero se realizara el k-fold cross validation:

En cada iteración del bucle, se realiza lo siguiente:

- Se dividen los datos en conjuntos de entrenamiento y prueba utilizando los índices generados por la función `skf.split()`.
- El conjunto de entrenamiento se equilibra con el método de sobremuestreo SMOTE.
- Los datos de la etiqueta se convierten de números enteros a vectores binarios utilizando la función `to_categorical` de Keras. Esto es necesario para usar la función de pérdida `'categorical_crossentropy'` en la compilación del modelo.
- Se crea el modelo de clasificación utilizando la clase `Sequential` de Keras y se compila con el optimizador `'adam'`, la función de pérdida `'categorical_crossentropy'` y la métrica de evaluación `'accuracy'`.
- Se ajusta el modelo al conjunto de entrenamiento equilibrado y se valida con el conjunto de prueba no equilibrado.
- El rendimiento del modelo se evalúa en el conjunto de prueba y se almacenan los resultados de la pérdida y la precisión en las listas `loss_scores` y `accuracy_scores`.
- Los resultados promedio se calculan a partir de las listas `loss_scores` y `accuracy_scores` y se imprimen en pantalla.

```
# Definir el número de folds
k = 5

# Crear el objeto StratifiedKFold
skf = StratifiedKFold(n_splits=k, shuffle=True, random_state=42)

# Crear listas para almacenar los resultados
loss_scores = []
accuracy_scores = []

# Iterar sobre los k-folds
for train_index, test_index in skf.split(X, y):
    # Dividir los datos en conjuntos de entrenamiento y prueba
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Aplicar oversampling al conjunto de entrenamiento
    sm = SMOTE(sampling_strategy='auto', random_state=42)
    X_train_res, y_train_res = sm.fit_resample(X_train, y_train)
```

```

# Crear numero entero en un vector binario de tamaño 4
y_train_res = to_categorical(y_train_res, num_classes=4)
y_test = to_categorical(y_test, num_classes=4)

# Crear el modelo de clasificación
model = Sequential()
model.add(Dense(units=16, activation='relu', input_shape=(400,)))
model.add(Dense(units=4, activation='softmax'))

# Compilar el modelo
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Entrenar el modelo
model.fit(X_train_res, y_train_res, epochs=14, batch_size=32,
validation_data=(X_test, y_test))

# Evaluar el rendimiento del modelo
loss, accuracy = model.evaluate(X_test, y_test)

# Almacenar los resultados en las listas
loss_scores.append(loss)
accuracy_scores.append(accuracy)

# Calcular los resultados promedio
avg_loss = np.mean(loss_scores)
avg_accuracy = np.mean(accuracy_scores)

# Imprimir los resultados promedio
print('Loss promedio:', avg_loss)
print('Accuracy promedio:', avg_accuracy)

```

RESULTADO

```

Epoch 7/14
50/50 [=====] - 0s 3ms/step - loss: 1.0053 - accuracy: 0.5337 - val_loss: 0.9744 - val_accuracy: 0.5394
Epoch 8/14
50/50 [=====] - 0s 4ms/step - loss: 0.9895 - accuracy: 0.5369 - val_loss: 0.9614 - val_accuracy: 0.5598
Epoch 9/14
50/50 [=====] - 0s 3ms/step - loss: 0.9752 - accuracy: 0.5606 - val_loss: 0.9497 - val_accuracy: 0.5649
Epoch 10/14
50/50 [=====] - 0s 4ms/step - loss: 0.9698 - accuracy: 0.5412 - val_loss: 0.9390 - val_accuracy: 0.5852
Epoch 11/14
50/50 [=====] - 0s 3ms/step - loss: 0.9603 - accuracy: 0.5644 - val_loss: 0.9333 - val_accuracy: 0.5725
Epoch 12/14
50/50 [=====] - 0s 3ms/step - loss: 0.9520 - accuracy: 0.5686 - val_loss: 0.9220 - val_accuracy: 0.5954
Epoch 13/14
50/50 [=====] - 0s 3ms/step - loss: 0.9402 - accuracy: 0.5688 - val_loss: 0.9337 - val_accuracy: 0.5623
Epoch 14/14
50/50 [=====] - 0s 3ms/step - loss: 0.9373 - accuracy: 0.5819 - val_loss: 0.9129 - val_accuracy: 0.5827
13/13 [=====] - 0s 4ms/step - loss: 0.9129 - accuracy: 0.5827
Loss promedio: 0.9648025496482849
Accuracy promedio: 0.5719468832015991

Process finished with exit code 0

```

8.Imagen "Resultados con wor2vec"

En este caso una de las cosas que se puede observar como se muestra en 8.Imagen “Resultados con word2vec” principalmente es que no es tan efectivo la precisión,esto debido a que es una capa pre-entrenada.

CAPA DE INCRUSTACIÓN CON TENSORFLOW

La aplicación de este modelo se realiza en el archivo **capa_embading.py**

Se realiza la misma estructura de leer las noticias y aplicar la funció de normalización

Primero obtenemos el numero de clases en este caso son 4 clases en donde aplicamos hot-econding en cada una de las etiquetas

Creamos un diccionario de palabras que va a ser del mismo tamaño del documento ya normalizado:

```
word_dict = {}
for sentence in df['Documento_Normalizado']:
    for word in sentence:
        if word not in word_dict:
            word_dict[word] = len(word_dict) + 1
```

Obtenemos la longitud máxima de de la secuencia para poder así proseguir con hacer la creación de la matriz de características que va a hacer la entrada de la capa de embedding.

En la función se puede notar que cada palabra de la oración itera y se utiliza en el diccionario de palabras creadas anteriormente donde se obtendrá el índice y se agrega a la lista seq,después de recorrer la lista seq se agrega a una lista X y así el resultado será una lista de lista donde cada sublista es una secuencia de índices de palabras como se muestra:

```
X = []
for sentence in df['Documento_Normalizado']:
    seq = []
    for word in sentence:
        seq.append(word_dict[word])
    X.append(seq)
```

Finalmente al que el tamaño de la secuencia no tiene misma longitud se tiene que agregar ceros para tener la misma longitud que corresponde a la max=lenght que es con el parámetro padding='post' como se muestra en la siguiente:

```
X = pad_sequences(X, maxlen=max_len, padding='post')
```

Después definimos el modelo como se muestra:

```
# Definir el modelo
model = Sequential()
```

```
#Primera capa convertir palabras de cada documento
model.add(Embedding(len(word_dict) + 1, 50, input_length=max_len))
#segunda capa para aplanar los los vectores de salida de la capa anterior
model.add(Flatten())
#Produce salida del tamaño de las clsaes
model.add(Dense(num_classes, activation='softmax'))
```

La primera capa, Embedding, se encarga de convertir las palabras de cada documento en vectores de longitud fija que representan el significado semántico de las palabras. La capa Embedding tiene como parámetros el tamaño del vocabulario (el número de palabras únicas en el conjunto de datos + 1), la dimensión del espacio de embedding (50) y la longitud máxima de la secuencia de entrada (en este caso, max_len).

La segunda capa, Flatten, se utiliza para aplanar los vectores de salida de la capa anterior en una sola dimensión.

La tercera y última capa, Dense, es una capa completamente conectada que toma los vectores aplanados como entrada y produce una salida de tamaño num_classes (en este caso, 4), con una función de activación softmax que convierte las salidas en una distribución de probabilidad sobre las clases.

RESULTADO

```
Epoch 1/10: 11s 145ms/step - loss: 0.0074 - accuracy: 0.9992 - val_loss: 0.0522 - val_accuracy: 0.9967
Epoch 2/10: 11s 149ms/step - loss: 0.0034 - accuracy: 0.9996 - val_loss: 0.0588 - val_accuracy: 0.9951
Epoch 3/10: 12s 161ms/step - loss: 0.0085 - accuracy: 0.9992 - val_loss: 0.0575 - val_accuracy: 0.9951
Epoch 4/10: 12s 159ms/step - loss: 0.0034 - accuracy: 0.9996 - val_loss: 0.0547 - val_accuracy: 0.9967
Epoch 5/10: 12s 153ms/step - loss: 0.0121 - accuracy: 0.9988 - val_loss: 0.0625 - val_accuracy: 0.9951
Epoch 6/10: 11s 167ms/step - loss: 0.0071 - accuracy: 0.9992 - val_loss: 0.0626 - val_accuracy: 0.9951
Epoch 7/10: 11s 166ms/step - loss: 0.0034 - accuracy: 0.9996 - val_loss: 0.0551 - val_accuracy: 0.9967
Epoch 8/10: 12s 154ms/step - loss: 0.0118 - accuracy: 0.9988 - val_loss: 0.0622 - val_accuracy: 0.9951
Epoch 9/10: 11s 148ms/step - loss: 0.0088 - accuracy: 0.9992 - val_loss: 0.0534 - val_accuracy: 0.9967
Epoch 10/10: 1s 48ms/step - loss: 0.0534 - accuracy: 0.9967
Test Accuracy: 0.9967426657676697
```

9.Imagen "Resultados capa de incrustación"

En este caso se obtuvo un resultado demasiado óptimo como se muestran en la 9.Imagen "Resultados capa de incrustación" esto puede ser debido a que está mal entrenado y que existe la posibilidad de que exista un overfitting al ser entrenado por el mismo corpus con el mismo tamaño de entradas para la capa del modelo a comparación de la de word2vec que solo se hizo de 400 entradas. Aunque como tal este tipo de capa de incrustación puede trabajar para un tipo de clasificación como lo son las noticias

COMPARACIÓN DE MODELOS

En este caso, podemos observar que la regresión logística tiene la mayor precisión en ambos conjuntos, con una precisión en el conjunto de prueba del 92.13%, seguido por Naive Bayes con una precisión del 88.32%, y finalmente KNN con una precisión del 70.30%.

También podemos ver que la precisión de Naive Bayes y la regresión logística son similares en el conjunto de entrenamiento, pero la precisión de la regresión logística aumenta significativamente en el conjunto de prueba, lo que sugiere que la regresión logística es mejor para generalizar a nuevos datos.

Por otro lado, KNN tiene una precisión significativamente menor en ambos conjuntos, lo que indica que este modelo no es adecuado para clasificar este tipo de datos.

Además, al analizar las métricas de precisión, recall y f1-score para cada categoría, podemos observar que la regresión logística tiene la mejor precisión en la mayoría de las categorías, mientras que KNN tiene una precisión baja en las categorías de "mundo" y "política".

En resumen, la regresión logística parece ser el modelo más efectivo para clasificar texto en categorías en este conjunto de datos, seguido de Naive Bayes.

En cuanto a la comparación que se tiene con las capas de embeddings se puede ver que con la de word2vec es menor a los demás modelos que se aplicaron pero consideran la capa de incrustación realizada con tensorflow se puede notar un precisión demasiado optima pero que se puede considerar un overfitting.

CONCLUSIONES

Durante la realización de esta práctica, se encontraron algunas complicaciones en la extracción de datos, a pesar de que en cierta medida fue más fácil obtenerlos de un repositorio como Kaggle. Asimismo, surgieron complejidades en el preprocesamiento, especialmente al buscar las palabras innecesarias para la clasificación del texto. Aunque existen librerías como Spacy que eliminan algunas palabras, se observó que la lematización y el stemming en las librerías utilizadas (NLTK y Spacy) no son las más óptimas para el idioma español.

En general, se obtuvieron resultados óptimos con la capa de incrustación y la regresión logística, alcanzando una precisión superior al 90%. Sin embargo, se debe tener en cuenta que la capa de embedding hecha por Tensorflow no funcionaría bien para datos que no están relacionados con noticias, debido a que solo está entrenada para ciertos tipos de texto. Por esta razón, se recomienda el uso de una capa de incrustación pre-entrenada como Word2vec, como se puede observar en la comparación de ambos resultados.