FACULTAD DE INGENIERIA
Universidad de Buenos Aires

[75.06] ORGANIZACIÓN DE DATOS

1º CUATRIMESTRE 2020

CURSO ÚNICO

TP2 - Predicciones - ¿Real Or Not?

AUTORES - Grupo COVIgData - 19

Carbón Posse, Ana Sofía - #101 187
<scarbon@fi.uba.ar>

Giampieri, Leonardo - #102 358
<lgiampieri@fi.uba.ar>

Goijman, Lautaro Enzo - #101 185
<lgoijman@fi.uba.ar>

Rodriguez Dala, Tomás - #102 361
<trodriguez@fi.uba.ar>

DOCENTES

Argerich, Luis

Golmar, Natalia

Ramos Mejia, Martín

Martinelli, Damian

Índice

1. Introducción

El objetivo de este segundo trabajo es predecir, para cada tweet brindado, si el mismo esta basado en un hecho real o no.

Para esto vamos a utlizar **Machine Learning**, una disciplina que busca poder generar clasificaciones en base a un entrenamiento sobre información pasada, seguida de una validación de las predicciones generadas. En el trabajo se prueban distintos algoritmos, los cuales todos en distinta manera hacen uso de los datos. Es por esto que es muy importante saber qué datos usar, y buscar cómo codificarlos de tal forma que mejor se aprovechen.

Con dicho propósito se utilizan como base el set de datos https://www.kaggle.com/c/nlp-getting-started. Esta misma contiene tweets realizados por distintas personas alrededor de todo el mundo anunciando una posible emergencia. Pero, no siempre está claro si las palabras de las personas realmente anuncian un desastre.

1.1. ¿Qué es Twitter?

Es un servicio de comunicación bidireccional con el que puedes compartir información de diverso tipo de una forma rápida, sencilla y gratuita. En otras palabras, se trata de una de las redes de microblogging más populares que existen en la actualidad y su éxito reside en el envío de mensajes cortos llamados “tweets”. Fue creada por Jack Dorsey y su equipo en 2006 y la idea se inspira en el envío de fragmentos cortos de texto, donde puedes añadir un enlace, imágenes, vídeo, encuestas o incluso gifs.

1.2. Elementos principales de Twitter

- **Tweet** : Es cada uno de los mensajes que se publica. Recordemos que cada uno de ellos contiene hasta 280 caracteres sin contar el material multimedia que incluyas en tus contenidos.
- **Hashtag (#)** : Se representa con el numeral y permite añadir tras él los términos que queramos. Se utiliza para facilitar búsquedas.
- **Mención (@)** : Cuando escribimos un tweet y queremos nombrar a una persona (o varias) que es usuario de Twitter, añadiremos su nombre de usuario precedido de @. De ésta manera, el usuario recibirá notificación en sus menciones y podrá respondernos (si así lo desea).
- **Ubicación** : Permite añadir la ubicación de lo comentado en el tweet.

2. Set de Datos

En la Figura 2.1 se muestra la estructura inicial de nuestro set de datos:

	id	keyword	location	text	target
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
2	5	NaN	NaN	All residents asked to 'shelter in place' are ...	1
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1
...	...	...	...	...	...
7608	10869	NaN	NaN	Two giant cranes holding a bridge collapse int...	1
7609	10870	NaN	NaN	@aria_ahrary @TheTawniest The out of control w...	1
7610	10871	NaN	NaN	M1.94 [01:04 UTC]?5km S of Volcano Hawaii. htt...	1
7611	10872	NaN	NaN	Police investigating after an e-bike collided ...	1
7612	10873	NaN	NaN	The Latest: More Homes Razed by Northern Calif...	1

7613 rows × 5 columns

Figura 2.1

- Este mismo pesa **297.5 KB**.
- No sabemos a que periodo de tiempo pertenecen estos tweets.

Filas:

- Hay 7613 filas.
- Cada fila identifica un tweet distinto.

Columnas:

- Hay 5 columnas.
- No todas presentan información, algunos fueron completados con **NaN**.

El contenido de las 5 columnas son :

- **id**: Identificador único para cada tweet.
- **keyword**: Palabra clave del tweet que indica que trata de una emergencia.
- **location**: La ubicación de donde fue el tweet publicado.
- **text**: El texto del tweet.
- **target**: Identifica si el tweet habla de una emergencia o no.

3. Conocimientos previos

En el primer trabajo se realizo un análisis exploratorio del set de datos de esta competencia. Este análisis nos permitió encontrar muchos casos donde había distinciones claras entre los tweets que eran auténticos y los que no. Por otro lado, también había casos donde la diferencia no era significativa y podíamos concluir que no debíamos tenerlos en cuenta a la hora de predecir. Estos son los insights que vamos a tener en cuenta para poder realizar las predicciones:

- A la hora de analizar las longitudes de los textos encontramos que los tweets verdaderos estaban más concentrados en el rango de los 130-150 caracteres mientras que en el caso de los tweets falsos estaban distribuidos de manera más uniforme con un pequeño pico en el mismo rango. Solo con esta predicción no podríamos decir si un tweet anuncia un desastre o no. Sin embargo, si tenemos en cuenta los otros factores, podría ser de ayuda para terminar de determinar sin un tweet anuncia un desastre o no.
- La separación entre las palabras más utilizadas para los tweets falsos y verdaderos está bien definida. Encontramos palabras relacionadas a desastres y muertes de un lado mientras que del otro no hay un tema común que las une.
- Observando el uso de hashtags y menciones dentro de los tweets encontramos que en los desastres verídicos se referían a organizaciones gubernamentales, medios de comunicación y desastres específicos.
- En cuanto a las keywords, lo que se pudo ver es que a medida que encontramos un keyword en más tweets verdaderos, menos probable será encontrarlo en tweets falsos y viceversa, lo que nos dice que hay una segmentación. Aquellas keywords que aparecen más frecuentemente en tweets verdaderos son más descriptivas sobre estos, por el hecho de tener más de una palabra y/o por usar palabras técnicas. Además encontramos keywords referentes a un sucesos particulares como el vuelo MH370, lo que nos permitió tener un filtro más sobre el resto de los tweets para poder encontrar aquellos que sean catastróficos sin necesidad del target.
- A la hora de hablar de las locations de los tweets, si bien la mayoría de los tweets se encuentran dentro de los distintos estados de Estados Unidos, no hay un patrón definido para determinar si un desastre ocurrió o no. Creemos que este parámetro no es útil a la hora de determinar la veracidad de los mismos.

Finalmente teniendo en cuenta todos estos ítems podremos determinar, con cierto margen de error, si un tweet desconocido habla de un desastre que verdaderamente ocurrió o no.

3.1. Feature Engineering

El feature engineering consiste en hacer análisis, limpieza y estructuración de las distintas columnas ('features') del set de datos. Este proceso es una tarea mayormente manual, es uno de los más importantes y más costosos del proceso de predicción. El objetivo es eliminar las columnas o 'features' que no sirven para hacer la predicción o crear nuevos features que permitan organizar adecuadamente el modelo para que no reciba información que no le es útil y que podría provocar predicciones de poca calidad o confianza.

3.2. Transformación de features

Las siguientes transformaciones se aplicaron sobre el feature "text". Decidimos eliminar diferentes palabras o cosas que no son relevantes para el análisis de los tweets. La razón principal para realizarlas fue generar una mejora en la calidad del texto para luego poder ser analizado.

- Eliminación de URLS.
- Eliminación de HTML.
- Eliminación de EMOJIS.
- Eliminación de puntuación.
- Eliminación de 'stopwords'.
- Corrección de gramática.

Esta transformación en particular, se realizo para poder utilizar estos features en modelos que solo aceptan features del tipo numérico.

- Conversión de tipo para las locations.

3.3. Nuevos features individuales

Como pudimos observar en el trabajo practico anterior, estas métricas resultaron interesantes para poder predecir la veracidad de los tweets , ya que que sugerían cierta tendencia hacia un target o el otro. Entonces decidimos agregarlas a nuestro data set con el fin de poder analizarlas y mejorar la calidad de la predicción.

- Cantidad de palabras por tweet.
- Cantidad de caracteres por tweet.
- Cantidad de hashtags por tweet.
- Cantidad de menciones por tweet.

4. Creación de nuevos Set de Datos

Luego de aplicar el feature engineering sobre nuestro set de datos original, vamos a trabajar con los siguientes set de datos:

	id	keyword	text	text_len	target	target_bool	location	city	country	state
0	1	NaN	Our Deeds Reason earthquake May ALLAH Forgive us	69	1	True	NaN	NaN	NaN	NaN
1	4	NaN	Forest fire near La range ask Canada	38	1	True	NaN	NaN	NaN	NaN
2	5	NaN	All residents asked shelter place notified off...	133	1	True	NaN	NaN	NaN	NaN
3	6	NaN	13000 people receive wildfire evacuation order...	65	1	True	NaN	NaN	NaN	NaN
4	7	NaN	Just got sent photo Ruby Alaska smoke wildfire...	88	1	True	NaN	NaN	NaN	NaN
5	8	NaN	RockyFire Update California why 20 closed dire...	110	1	True	NaN	NaN	NaN	NaN
6	10	NaN	flood disaster Heavy rain causes flash floodin...	95	1	True	NaN	NaN	NaN	NaN
7	13	NaN	Im top hill I see fire woods	59	1	True	NaN	NaN	NaN	NaN
8	14	NaN	Theres emergency evacuation happening building...	79	1	True	NaN	NaN	NaN	NaN
9	15	NaN	Im afraid tornado coming area	52	1	True	NaN	NaN	NaN	NaN

Figura 4.1: Set de Datos con transformaciones sobre text.

Este set es como el set de datos original con la diferencia que el feature 'text' tiene aplicado todas las transformaciones mencionadas en la sección anterior.

	id	keyword	text	text_len	target	target_bool	location	city	country	state	cantidad_palabras	cantidad_caracteres	cantidad_stopwords	cantidad_menciones	cantidad_hashtags
0	1	NaN	Our Deeds Reason earthquake May ALLAH Forgive us	69	1	True	1	1	1	1	8	48	5	0	1
1	4	NaN	Forest fire near La range ask Canada	38	1	True	1	1	1	1	7	36	0	0	0
2	5	NaN	All residents asked shelter place notified off...	133	1	True	1	1	1	1	13	95	9	0	0
3	6	NaN	13000 people receive wildfire evacuation order...	65	1	True	1	1	1	1	7	58	1	0	1
4	7	NaN	Just got sent photo Ruby Alaska smoke wildfire...	88	1	True	1	1	1	1	10	59	6	0	2
5	8	NaN	RockyFire Update California why 20 closed dire...	110	1	True	1	1	1	1	13	88	3	0	3
6	10	NaN	flood disaster Heavy rain causes flash floodin...	95	1	True	1	1	1	1	12	86	2	0	2
7	13	NaN	Im top hill I see fire woods	59	1	True	1	1	1	1	7	28	8	0	0
8	14	NaN	Theres emergency evacuation happening building...	79	1	True	1	1	1	1	7	60	5	0	0
9	15	NaN	Im afraid tornado coming area	52	1	True	1	1	1	1	5	29	5	0	0

Figura 4.2: Set de Datos con el agregado de nuevos features.

Este set de datos contiene las transformaciones sobre el feature 'text' y además incluye todos los nuevos features generados mencionados en la sección anterior.

Decidimos tener dos set de datos porque hay algunos modelos de Machine Learning que utiliza únicamente el feature 'text'. Entonces a modo de ahorrar memoria y tiempo a la hora de ejecutar los algoritmos, en estos casos usamos el primer set de datos.

5. Modelos

Los modelos que vamos a utilizar son distintos algoritmos de Machine Learning. En un algoritmo vamos a llamar **parámetros** a aquellos valores que el algoritmo encuentra a partir de los datos y vamos a llamar **hiper-parámetros** a aquellos datos que el algoritmo necesita para poder funcionar.

5.1. KNN

KNN es un algoritmo de clasificación. Se denomina KNN (K-Nearest-Neighbors) ya que para predecir la clase de un determinado punto en $\mathbb{R}^n$ utiliza la clase mayoritaria de sus K-vecinos mas cercanos. Para esto es necesario proporcionarle un set de datos con elementos los cuales conocemos su clase. Este algoritmo no necesita tiempo de entrenamiento porque calcula los vecinos a la hora de predecir, esto significa que los tiempos de predicción son mayores a los de otros modelos. Otra desventaja de KNN es el problema de la dimensionalidad, es un algoritmo que a medida que aumentamos la cantidad de dimensiones se vuelve cada vez menos eficiente. Para poder usar KNN hay que definir el valor de k, es decir, cuantos vecinos vamos a considerar.

A la hora de implementarlo tenemos que vectorizar el texto de todos los tweets, para esto utilizamos dos métodos:

- Bag of words: Implica generar un vector de N dimensiones (para N cantidad de palabras diferentes) donde cada palabra tiene una dimensión asignada. Luego ponemos un 1 en cada dimensión si el tweet tiene la palabra correspondiente o un 0 en otro caso.

- TF-IDF: Similar a bag of words pero en vez de poner unos y ceros calculamos el tf-idf. Este tiene en cuenta la cantidad de veces que apareció en el tweet y la frecuencia global de la palabra.

Los vectores resultantes son vectores muy dispersos con más de 15000 dimensiones, es por eso que debemos reducir su tamaño. Para esto utilizamos feature hashing y terminamos con vectores más pequeños que si podemos usar en nuestro modelo.

La figura 5.1 visualiza los vectores generados a partir de TF-IDF (reduciendo las dimensiones de los vectores utilizando una combinación de SVD y TSNE).

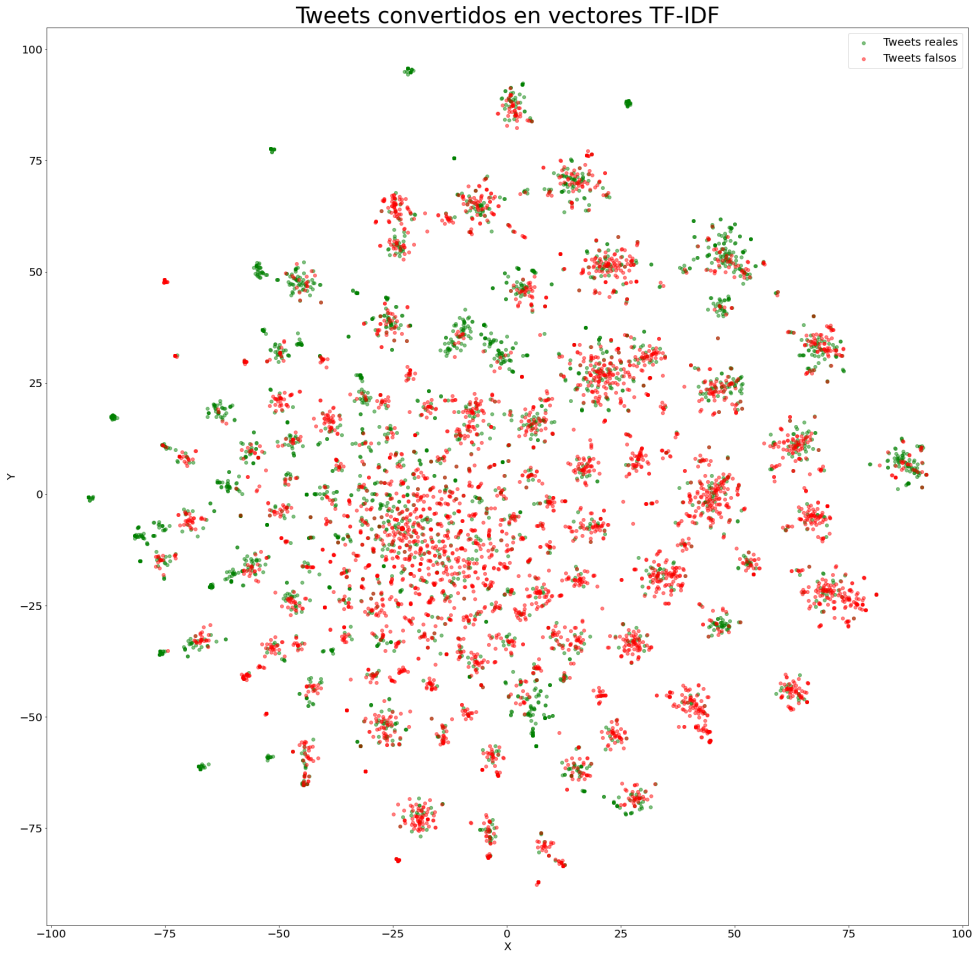

Figura 5.1

Una vez preprocesado el set de ”entrenamiento” podemos crear nuestro KNN y alimentarlo con nuestros vectores. Antes de hacer esto separamos una parte del set para poder validar la precisión más adelante. A la hora de crear el modelo tenemos que elegir el valor de k que minimice el error.

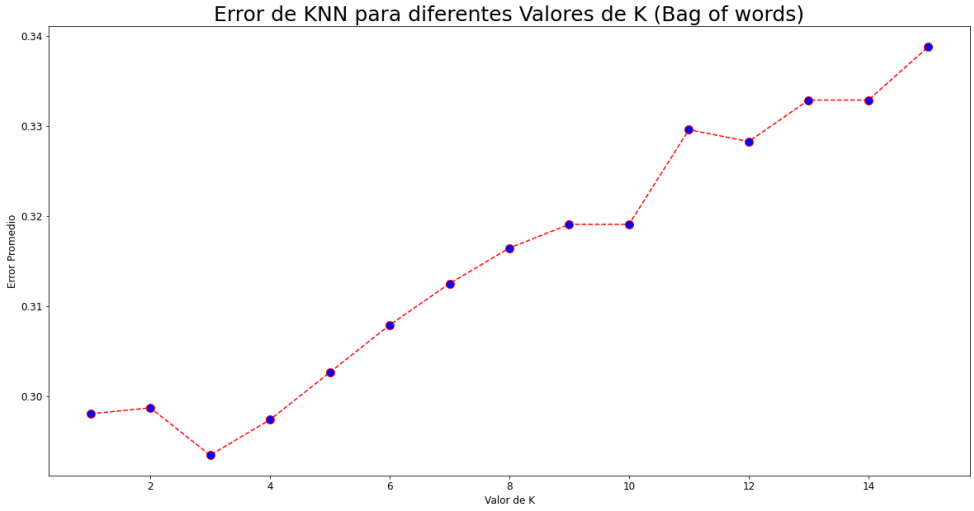

Figura 5.2

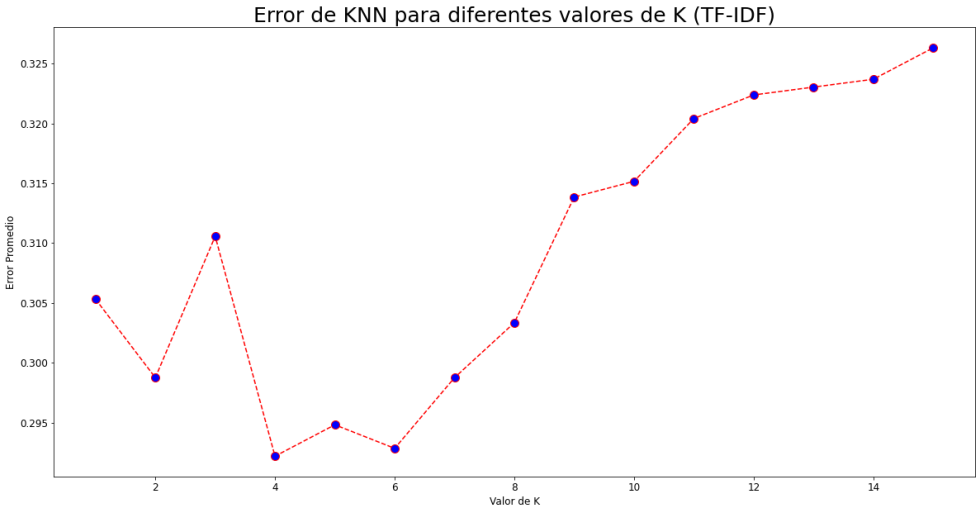

Figura 5.3

A fin de cuentas bag of words y TF-IDF tienen resultados similares para sus respectivos valores óptimos de k. Ambos tienen un validation accuracy de alrededor de 0,7.

En conclusión, KNN no tiene el mejor de los puntajes pero nos sirve de base para el resto de los modelos.

5.2. Arboles de decisión

Un árbol de decisión es un árbol binario en donde en cada nodo dividimos el set de datos en dos de acuerdo a un cierto criterio. El objetivo es llegar a nodos hoja en los cuales podamos clasificar correctamente nuestros datos. El XGboost (Extra Gradient Boosting) parte del árbol de decisiones que se implementa, pero potenciando los resultados de estos, debido al procesamiento secuencial de la data con una función de pérdida o coste, la cual, minimiza el error iteración tras iteración, haciéndolo de esta manera, un pronosticador fuerte.

Decidimos utilizar distintos arboles de decisión, para estudiar los distintos alcances de diversos modelos, además de sacar varias conclusiones con respecto al uso de los mismos para información de textos. Los arboles de decisión utilizados en nuestro trabajo son, XGBoost, LightGBM y CatBoost. Estos tres son los mas comunes y utilizados para los modelos de Machine Learning. Todos estos son muy eficientes a la hora de predecir resultados rápidos. Debemos aclarar que todo esto es posible si se utilizan los features correctos, ya que el árbol justamente divide las predicciones en nodos(ramas) hasta lograr llegar a una hoja y predecir, en nuestro caso la veracidad de un tweet.

En el siguiente link podemos ver la gran cantidad de posibles hiper-parámetros que reciben estas. Algunos son muy complejos.

5.2.1. XGBoost

1. **Clasificación**

   Inicialmente se probo este modelo utilizando únicamente el feature 'text'. Como sabemos, estos modelos solo trabajan con features numéricos, entonces tuvimos que convertir el text. Para esto se utilizo:

   - **CountVectorizer:** Convierte una colección de documentos de texto en una matriz de tokens. Esta implementación devuelve una representación dispersa. La dimensión de esta depende de la cantidad de distintas palabras que se encuentren en todos los documentos. En este caso cada tweet se toma como si fuera un documento.

   Manualmente se busco los hiper-parametros que mejor ajustaran, y leugo de varios intentos, lo mejor que se obtuvo fue:

   - Accuracy = 75.13 % - Score = 0.74287

   Si bien este resultado no esta mal, se esperaba un score mas alto. Cabe destacar que el text tenia implementado varias transformaciones.

   Luego, se realio la misma prueba, es decir, solo se trabajo con el feature 'text' tranformado pero en esta caso, le aplicamos un embedding:

   - **Universal Sentece Encoder:** Genera por cada tweet un vector de 512 dimensiones numéricas.

   Esta vez se realizo RandomSearch reiteradas veces, hasta obtener:

   - Accuracy = 79.66 %

(Esta vez no se probo en Kaggle porque teníamos una cantidad limitada de submmits, pero se supone un Score = 0,77 - 0,78.) El accuracy aumento levemente, esto nos demuestra que el encoder utilizado es mejor. A su vez, nos empezamos a cuestionar si era posible que las transformaciones realizadas sobre el texto, estén haciendo 'ruido'. Para ello, decidimos probar el mismo enconder, pero ahora solamente eliminado las URLs. Los resultados obtenidos:

- Accuracy = 83.41 % - Score = 0.81336

Realmente el resultado nos sorprendió, si bien este se aplico sobre el feature que tiene mas peso en todo el set de datos, es un score muy bueno para ser un árbol de decisión.

Para no dejar de lado los features numéricos, se probo el algoritmo con ellos.

- Accuracy = 66.51 %

Si comparamos con el resto de los resultados, este es muy bajo. Nuevamente nos demuestra que los features numéricos no tienen un gran peso a la hora de predecir la veracidad de los tweets.

Luego de esto, se decidió probar los features numéricos adicionando el text con el embedding. Se obtuvo:

- Accuracy = 79.71 %

Si bien se nota el aporte que brinda el texto del tweet a los features numéricos, el Score disminuyo. Esto demuestra que a pesar de aportar cierta ayuda, de alguna manera, los features numéricos generan ruido.

Para este modelo, se pudo optimizar la búsqueda de los hiper-parámetros utilizando RandomSearch y GridSearch. Cada uno se especifica en la sección (Tuning de hiperparametros).

2. **Regresión**

Este modelo divide los nodos(ramas) según el calculo de MSE, el error cuadrático medio. Cada nodo se divide en como siempre en la mayor cantidad de muestras por nivel. Calculando cuanto error tiene la perdición en cada nodo. Ejemplo de uno de los arboles generados luego del entrenamiento del modelo.

Figura 5.4: XGB de Regresión

Los árboles de decisión para regresión se construyen usando algoritmos de Greedy . Para regresión, la función de coste es la siguiente:

$$J(a, l_a) = \frac{m_{izquierdo}}{m} MSE_{izquierdo} + \frac{m_{derecho}}{m} MSE_{derecho}$$

- a es la abreviatura del feature
- l_a significa el límite del atributo
- m se refiere al número de muestra

Acá se puede observar que features fueron los que mas se usaron para dividir los nodos del árbol.

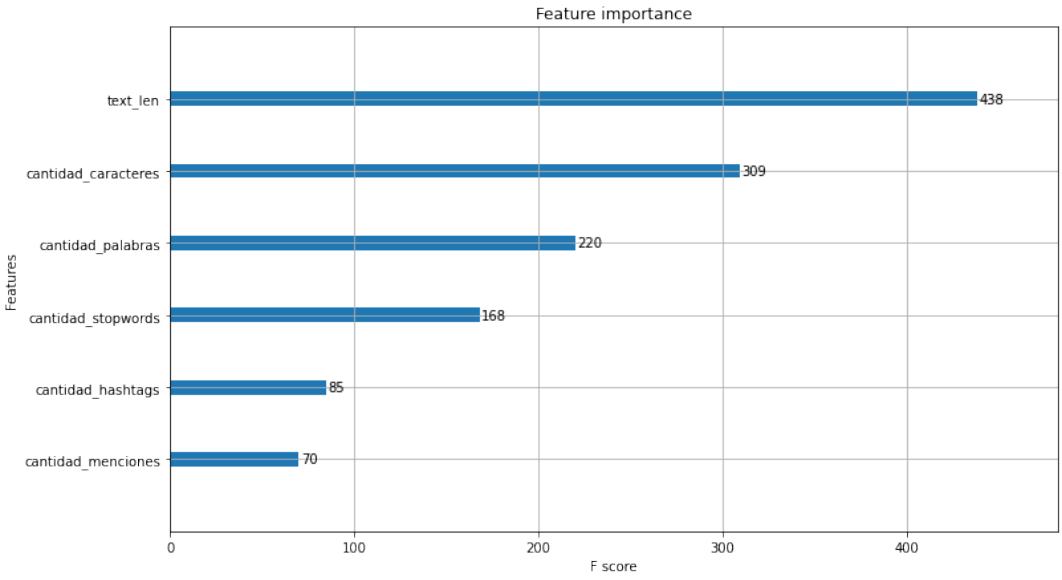

Figura 5.5: Features mas importantes

Este modelo, luego del proceso de tuning, con el los features numéricos llego al score de 0,66717 Utilizando el enconder(Universal Sentece Encoder) del texto el score que obtuvimos fue de 0,81121 y luego del tuning 0,82623

5.2.2. LightGBM

Este modelo es relativamente nuevo y de los mas exitosos a la hora de ser comparado con otros árboles de decocción. Esto se debe a la forma en la cual el algoritmo genera el árbol. A diferencia de la mayoría de los modelos este creces en forma vertical, es decir, intenta ser un árbol mas alto (mayor profundidad).

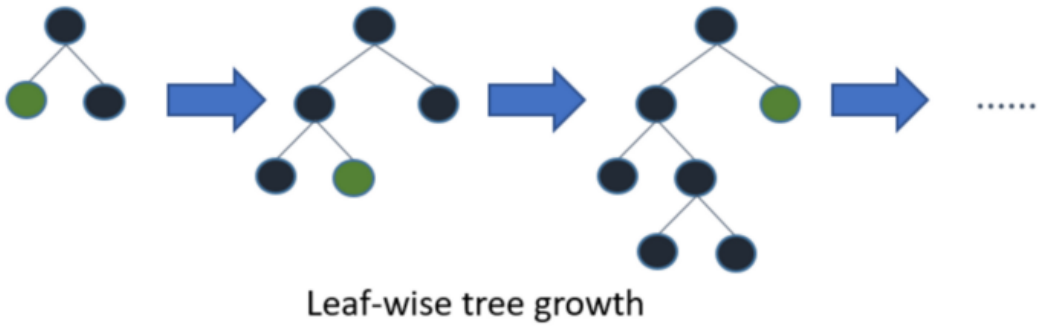

Figura 5.6: LightGBM

Figura 5.7: Otros Arboles

Decidimos probar este modelo, ya que es muy rápido a la hora de generar resultados. Como dice su nombre, es liviano y veloz. A la hora de procesar la información, es muy ligero y esta preparado para ser utilizado en la GPU de ser necesario.

El problema con este modelo, es su tendencia a overfittear en casos de pocas filas de información. Se recomienda usar este modelo a la hora de entrenar el mismo con mucha mas información que la que se nos dio para este trabajo. Por ende, no seria este el mejor modelo para llevar adelante el mejor score. De todas formas luego al entrenar este modelo, con los features numéricos, que obtuvimos del texto, llegamos a obtener un score de 0,67698 y luego con embedding(Universal Sentece Encoder) obtuvimos 0,79619

5.2.3. CATBoost

Este modelo, al igual que los anteriores, es muy bueno clasificando la información. Lo que lo hace especial es su gran cantidad de herramientas. Estas son de tuning, plotting e incluso su código abierto. Esto permite meterse en varios detalles muy interesantes a la hora de analizar en profundidad el modelo. Por ejemplo utilizamos la libreria Shap para analizar los features en este modelo.

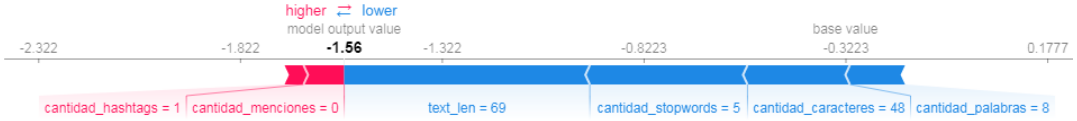

Figura 5.8: Primeras predicciones del modelo

A continuación compráramos el efecto de dos features. Se pude ver claramente que el impocato de text_len es superior al de la cantidad_meciones que aparecen en el texto de los tweets.

Figura 5.9: Como afecta text_len

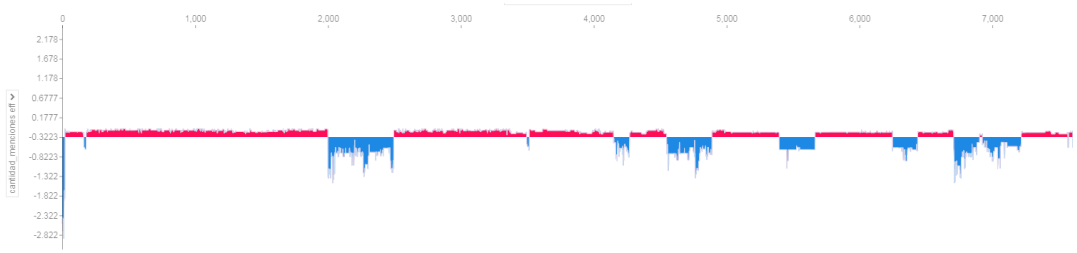

Figura 5.10: Como afecta cantidad_meciones

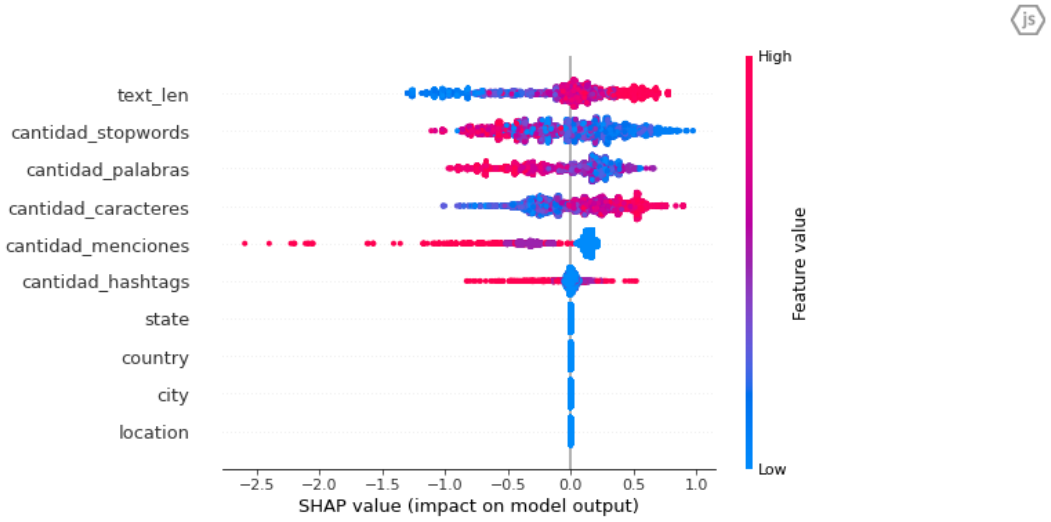

Figura 5.11: Features mas importantes

A pesar de aprender y analizar mucho el comportamiento del modelo, este no obtuvo muy buenos resultados. Esto debido a que tiende a necesitar mayor cantidad de filas y otros tipos de features. Este modelo, luego del proceso de tuning llego al score de: 0,77382

5.2.4. Naive-Bayes

Este algoritmo esta basado en el teorema de Bayes, tiene varias propiedades interesantes. Es un algoritmo extremadamente sencillo, rápido y que escala muy bien El único elemento necesario para entender el algoritmo es el teorema de Bayes. Es un clasificador muy popular para textos pero puede usarse también en cualquier otro caso, su principal virtud es la capacidad de escalar a datos realmente masivos sin ningún tipo de problema, no todos los algoritmos tienen esta ventaja.

$$P(\text{class}|D) = \frac{P(\text{class})P(D|\text{class})}{P(D)}$$

Figura 5.12: Teorema de Bayes

El algoritmo fue utilizado en todas sus variantes:

- **Gausiano**
- **Bernouilli**
- **Multinomial**
- **Complemento**

Inicialmente se probo los cuatro algoritmos únicamente con features numéricos creados previamente durante el proceso de 'feature engineering'. Los resultados que se obtuvieron fueron bajos:

- Gaussiano: Accuracy = 61.35 %
- Bernouilli: Accuracy = 60.05 %
- Multinomial: Accuracy = 62.60 %
- Complemento: Accuracy = 61.73 %

(Esto es una aproximación ya que no se pudo probar en Kaggle absolutamente todos los algoritmos porque teníamos una cantidad limitada de submits.)

Luego se decidió probar los algoritmos con los features numéricos adicionando el texto vectorizado, ya que estos modelos solo aceptan features numéricos. Para esto se utilizo:

- **Universal Sentece Encoder:** Genera por cada tweet un vector de 512 dimensiones numéricas.

Los resultados que se obtuvieron mejoraron notoriamente en cuanto a la prueba anterior, pero estos aun seguían siendo bajos también:

- Gaussiano: Accuracy = 78.41 %

- Bernouilli: Accuracy = 78.84 %

Los otros dos modelos no se pudieron probar ya que no aceptan números negativos, y 'USE' devuelve tanto números positivos como negativos en las distintas posiciones de los arreglos. Esta notaria mejora nos hace notar que el feature 'text' tiene bastante peso a la hora de predecir.Por esta razón, se decidió hacer una ultima prueba utilizado únicamente el texto con el mismo embedding. Nuevamente los resultados que se obtuvieron mejoraron:

- Gaussiano: Accuracy = 81.01 %
- Bernouilli: Accuracy = 80.39 %

Estos resultados confirman la hipótesis de la prueba anterior, el feature 'text' tiene un gran peso para predecir la veracidad de los tweets y los features numéricos hacen ruido.

Si bien este algoritmo no dio los mejores resultados, se obtuvo un buen resultado cercano a lo deseado. Además se pudo notar su rapidez y sencillez.

5.3. Redes Neuronales

Utilizamos distintos tipos de redes neuronales, entre ellas CNN's, RNN's y modelos pre-entrenados en un corpus grande, entre ellos BERT y XLNet. Al ser estos modelos más complejos, consideramos que pueden sacar ventaja a los features numéricos que se puedan obtener del set (aunque podrían beneficiarse mutuamente haciendo algún tipo de ensamble). El texto de los tweets es algo que hasta el momento no atacamos directamente, y pensamos que ahi se encuentra un gran porcentaje de la información que nos ayudará a predecir.

5.3.1. Redes Neuronales Convolucionles (CONV-1D)

Estas redes precisan un embedding, no necesariamente para el texto entero, sino para cada palabra en un texto. Lo que hacen es aplicarle n filtros a estos embeddings de una cantidad fija de palabras k. La idea es que se entrenen a medida que se entrena toda la red, y que aprenda los k-gramas presentes en los textos. A medida que se va profundizando se pueden agregar filtros de una cantidad mayor de palabras, y estos subsecuentemente aprenden bi-gramas, tri-gramas, etc. Luego, se aplica una capa de MaxPooling, en donde se obtiene el valor mas alto del resultado de la convolución para cada filtro. Estos valores capturan lo que para los filtros es lo mas importante del texto. Lo que logramos capturar es una suerte de distancia entre los textos. Sabemos que si el embedding es acorde, aquellos tweets que compartan estos features complejos serán candidatos a ser agrupados. Luego de las capas de convolución y max pooling, los resultados pasan por una o más capas Fully-Connected, en donde se termina de clasificar el texto (en nuestro caso utilizando una funcion de activacion sigmoid en la ultima capa para poder clasificar outputs binarios)

La primera red utiliza un embedding sin preentrenamiento. El preprocesamiento del set en este caso fue eliminar urls, tags, emojis, etc, presentes en los tweets, además de eliminar stopwords en ingles (pues es el idioma más presente en este). Los tweets fueron tokenizados utilizando el tokenizador de la libreria de Keras, pasando cada palabra a un entero. Cada tweet se procesó utilizando estos enteros, obteniendo una secuencia de longitud variable de enteros para cada tweet, a la cual se le agrego un padding para obtener una longitud fija. Luego, se creó una matriz de embedding, en la cual cada fila tiene un vector representando la palabra apuntada por el índice, anteriormente obtenido del tokenizador. Estos vectores fueron todos inicializados utilizando muestras de una distribución normal media 0 varianza 1. Encontramos que es un tanto difícil entrenar a este modelo sin que logre overfittear el set, pero luego de usar el Tuner de Keras obtuvimos una precisión en el set de test de 0.75. El próximo modelo de CONV-1D utiliza Word Embeddings pre-entrenados de GloVe (Global Vectors for Word Representantions). Es un modelo de aprendizaje no supervisado que se entrena en un corpus utilizando una matriz de co-ocurrencias de palabras, que tabula que tan frecuentemente aparecen pares de palabras juntas en el conjunto de textos en cuestión. En particular, utilizamos el corpus pre-entrenado que ofrece GloVe de Twitter. Este fue entrenado en miles de millones de Tweets, y tiene un rango de 1.2 millones de palabras en minúscula, cada una con vectores en 50, 100, 150 o 200 dimensiones. En la práctica obtuvimos mejores resultados utilizando vectores de 200 dimensiones. Lo que ofrece GloVe es una medida de similitud semántica entre palabras, a través de la distancia coseno. Esto nos permitiría agregando una red obtener importancia semántica de los tweets, de modo que tweets que se refieran a temas

similares terminaran juntos en el espacio de embedding. La ventaja sobre el modelo anterior es que aqui, el embedding ya está entrenado en cierto corpus, y por lo tanto entiende parte del lenguaje. Es bastante probable que en este set, que es pequeño, no podamos llegar a mucha mayor precisión utilizando una capa de embedding inicializada de manera random, por lo que decidimos utilizar este modelo. Con este modelo llegamos a una precisión del 0.8 en el set de test, sacando una buena ventaja del último modelo.

5.3.2. Redes Neuronales Recurrentes (RNN y LSTM)

Lo que queremos intentar ahora es tener una suerte de persistencia de lo que la red puede aprender sobre un tweet. Esto es, queremos que la red recuerde las palabras que vio antes de la que procesa actualmente, ya que naturalmente una persona aprende de esta manera, no resetea su entendimiento de lo que esta leyendo cada vez que encuentra una palabra nueva. Las redes neuronales recurrentes tratan de atacar este problema, incorporando la funcionalidad de persistencia. De esta manera, se tiene un entendimiento más claro de como la red infiere la próxima palabra en relación a las anteriores. Las RNN's reciben un input y el output es realimentado junto con un nuevo input. De esta manera, la información previa persiste en la red. Esto es bastante útil cuando la red trata de aprender como predecir la próxima palabra en una frase corta. El problema ocurre cuando la frase se vuelve larga, la red tiene que "mirar hacia atrás"para encontrar información que la ayude a predecir, pero si el espacio entre lo que se esta tratando de predecir y la información necesaria es muy grande, la red pierde la capacidad de recuperar esta información. Este concepto de mirar hacia atrás se llama Backpropagation Through time. Si pensamos una sola iteración de esta red, es una función f del input y de los estados de la salida anterior (estos estados ofrecen esa persistencia):

$$h(t+1) = fw(h(t), x(t+1))$$

W es lo que queremos optimizar, los pesos para cada iteración. Si calculamos el gradiente del campo con respecto al costo (w es función de cada estado), encontramos que la matriz de pesos se multiplica n veces por cada secuencia:

$$x(t) = W^n x(0)$$

Si W>1, cuando n se hace grande x(t) tiende a infinito, y cuando W<1, x(t) tiende a cero. Por lo tanto, el gradiente explota o desaparece. Para solucionar esto entran las redes LSTM. Cada celda LSTM decide si quedarse con la información que obtuvo de estados anteriores, o sobreescribirla y borrarla, de modo tal que se mitiga el efecto del gradiente. Para esta arquitectura, utilizamos los embeddings de GloVe como input y probamos diferentes unidades, regularizaciones y dropouts para las celulas de LSTM. Ademas, utilizamos el Wrapper de Keras de Bidirectional, para que la capa de LSTM pueda leer la informacion de izquierda a derecha y de derecha a izquierda. Esto es interesante porque se guarda información sobre el contexto del tweet en ambas direcciones. Luego de hacer un poco de fine-tuning, encontramos que la mejor accuracy la obtuvimos con un modelo LSTM unidireccional, con un score de 0.80784. A continuación mostramos una visualización: como los outputs de las celdas LSTM se distribuyen dependiendo de la predicción que hizo el modelo sobre el set de TEST. Esto nos muestra si el modelo realmente puede separar los tweets o si sus predicciones son azarosas. Los outputs

de las lstm son un vector en 20 dimensiones, ya que es la cantidad de unidades que usa este modelo. Se reducen de 20 a 2 dimensiones usando T-SNE.

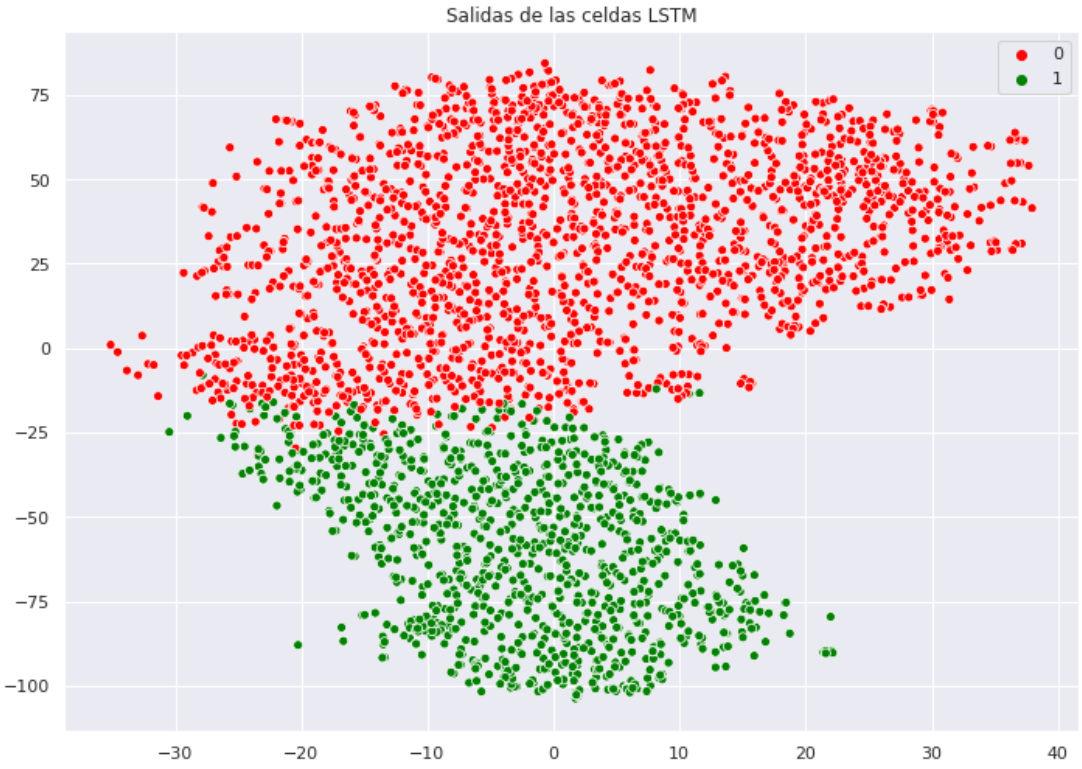

Figura 5.13

Podemos notar que el modelo clusteriza los dos grupos. A partir del resultado obtenido en kaggle y de esta visualizacion, podemos notar que el modelo tiene un criterio de segmentación de los tweets. Notamos de todas maneras que la frontera no está muy separada, lo cual puede indicar que este modelo no decide sobre ciertos tweets.

5.3.3. Attention Mechanisms y transformadores (BERT y XLNet)

Los modelos de attention mechanisms para lenguaje tratan de replicar el concepto de atención. En vez de utilizar toda la oración para tratar de predecir cada palabra, solo tiene en cuenta a la parte del input que es relevante para cada elemento del output. Para poder llevar esto a cabo, se utilizan todas las capas internas para predecir cada token (de esta manera se puede representar casos donde hay palabras que están lejos una de otra pero están relacionadas) pero cada conexión capa-palabra tiene un peso por lo tanto, las capas más relevantes tienen mayor capacidad de modificar ese elemento de la salida. Esto significa que se genera un contexto para cada palabra y no tiene que prestarle atención a toda la oración.

En nuestro caso nosotros utilizamos transformadores, estos son modelos que se distinguen de las RNN por usar puramente attention mechanisms para encontrar el contexto de cada palabra de la salida. Increíblemente, usan la misma técnica para encontrar relaciones entre las diferentes palabras del input y, por separado, las diferentes palabras del output. De esta forma logra encontrar varios contextos diferentes que después puede combinar para mejorar la salida. Una ventaja de todo esto es que las diferentes búsquedas de contextos se pueden llevar a cabo de forma paralela (a diferencia de las RNN que entrenan de manera secuencial) y así aumentar la velocidad de entrenamiento.

BERT BERT es un modelo basado en la arquitectura de transformador, un mecanismo de atención construido para aprender relaciones entre palabras (o sub palabras) en un corpus. El transformador original utiliza dos mecanismos, un Encoder y un Decoder. Generalmente los transformadores se utilizan en tareas de traducción de idiomas, el Encoder utiliza información contextual de la entrada y el Decoder se encarga de mapear esta información guardada en tensores al idioma objetivo. BERT utiliza solo el Encoder, para poder aprender de tal lenguaje como se relacionan las palabras y como se estructuran las oraciones. Al igual que las LSTM'S bidireccionales, BERT puede obtener contexto de izquierda a derecha y de derecha a izquierda a una oración, pero la diferencia es que puede hacer esto de manera secuencial, es decir que puede leer la oración entera en una pasada. El modelo es un stack de Encoders (12 para BERT-base y 24 para BERT-large), y además cada encoder tiene su propia red oculta (768 salidas para base y 1024 para large). Esto nos deja un total de 110M de parámetros para base y 300M de parámetros para large, lo cual es inmenso a comparación de otros modelos (incluso no pudimos correrlo localmente, se utilizo pura y exclusivamente colab con GPU/TPU). La intención no es entrenar un modelo BERT en nuestro set de datos, sino levantar un modelo pre-entrenado y aplicarlo a una tarea especifica, en nuestro caso clasificación de tweets. Para esto utilizamos la simpática librería Transformers, en la cual se encuentran varios modelos de transformadores además de BERT, como XLNet, GPT2, entre otros. Esta manera de entrenar un modelo se conoce como Transfer Learning, ya que estamos utilizando un modelo entrenado en una tarea especifica, Language Modeling, para aplicarlo en una tarea de clasificación. Creemos que es muy interesante aplicar BERT ya que, a partir de haber modelado un lenguaje, puede entender a lo que los tweets refieren, contexto de las palabras, contexto general, información semántica, etc. Habiendo levantado un modelo preentrenado, utilizamos un tokenizador especifico para BERT que, además de convertir las palabras y sub palabras en enteros, agrega dos tags, [CLS] y [SEP]. El primero es un tag de clasificación, que guarda información sobre la sentencia. Este tag esta en todas las capas, y a medida que se va profundizando, los features que obtiene el modelo sobre el input son mas y mas específicos, y esa información queda guardada en la posición de este tag (internamente se hacen Avg. Pooling del resto de las capas). El otro simplemente es un separador que utiliza BERT. Para clasificación, podemos aplicar simplemente a la salida de la posición del tag [CLS] en la ultima capa de BERT una capa densa de salida de una dimensión con función de activación sigmoidea (recibiría un tensor de 768 dimensiones) que clasifique el texto de entrada. Al ser el modelo denso en cantidad de parámetros y el set de entrenamiento chico, este overfittea violentamente, de modo que tenemos que ajustar el Learning Rate para el algoritmo de Gradient Descent, de tal manera de no obliterar los pesos del modelo preentrenado. A continuación vemos como el modelo logra

overfittear el set de entrenamiento con tan solo unas "pocas"épocas (pocas a comparación del resto de los modelos). Para estos ejemplos, utilizamos BERT con una longitud máxima de secuencia de 128 (cantidad máxima de palabras admitidas en un tweet), un batch size de 16, el split de validación del 20 % respecto del set de train (no agrandamos el batch size por falta de memoria en la gpu de colab). Para optimizar, utilizamos Adam con un lr de $1e{-}05$, tal que overfittee lo menos posible pero al mismo tiempo que pueda aprender. Vemos el output del fit del modelo en las siguientes visualizaciones.

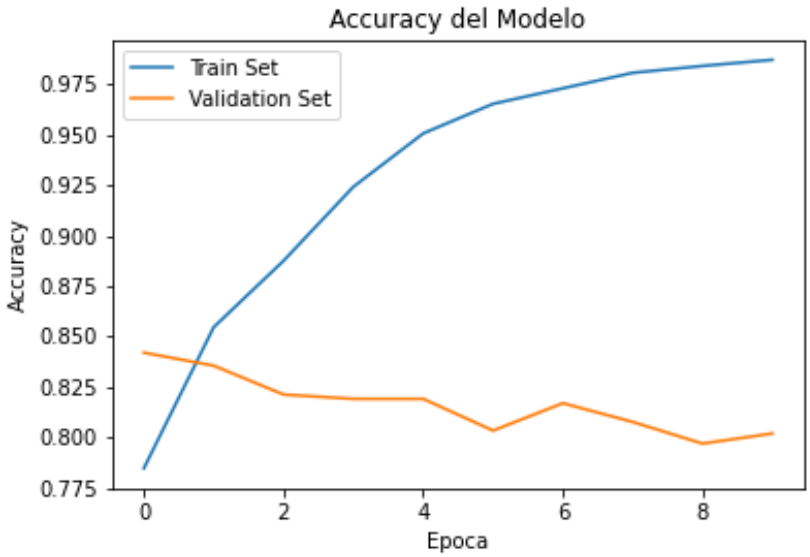

Figura 5.14

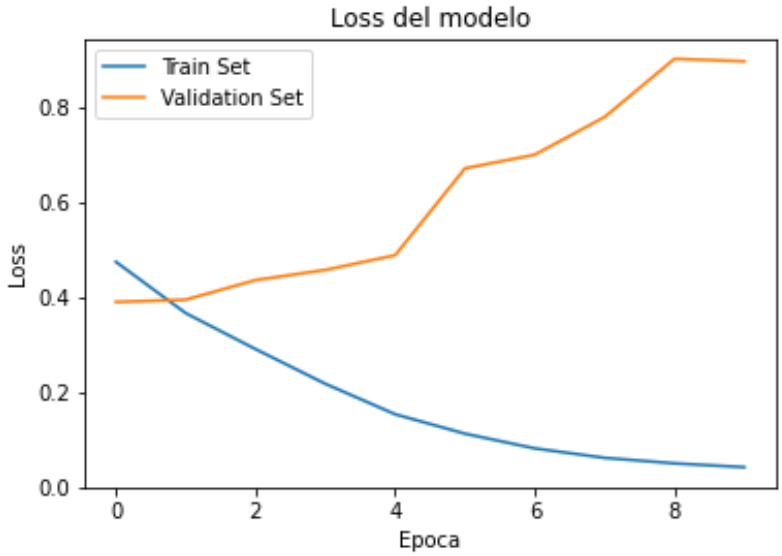

Figura 5.15

Lo que notamos es que ya a partir de la segunda época, el validation loss sube muy rápidamente, y que el validation accuracy lentamente desciende, por lo cual todos los modelos en los cuales hicimos fine-tuning de BERT utilizamos entre 1 y 2 épocas para evitar esto. También, a modo de comparativa con otros modelos, nos encontramos que con un learning rate órdenes de magnitud menor a los vistos en CONV-1D y LSTM'S, el modelo aun así logra overfittear mucho. Esto se debe a que, al tener una cantidad grande de parámetros, el modelo empieza a hacer cuello de botella con la cantidad chica de datos del set de entrenamiento. Para nuestra tarea de clasificación, utilizamos varias alternativas a la salida de BERT:

- BERT, 2 epocas, batch size 32, lr=1e-05 : Kaggle = 83.879 %
- BERT + CONV-1D: Kaggle = 83.665 %
- BERT + concat últimas 4 hidden layers: Kaggle = 82.133 %
- BERT-Large: Kaggle = 82.133 %

Hasta el momento, la primera combinación nos dió el puntaje más alto que obtuvimos en Kaggle, 83.879 %. En definitiva, si bien es un modelo muy pesado, pudimos obtener resultados excelentes en la tarea de clasificación, que ningún otro modelo que utilizamos hasta el momento pudo equiparar. En la práctica, es un modelo difícil de fine-tunear, ya que requiere utilizar una computadora potente. Usando nuestras computadoras, entrenar una sola época nos devolvió resultados no favorables: esperar muchas horas o mensajes OOM. Por eso decidimos usar Colab, para poder aprovechar las Cloud GPU's/TPU's. A continuación, visualizamos algo similar a lo visto anteriormente con las redes LSTM's, ¿Cómo se distribuye el output de la salida [CLS] de la última capa oculta de BERT en relación al target predecido? A esta salida decidimos llamarle "Tweet Embeddings"

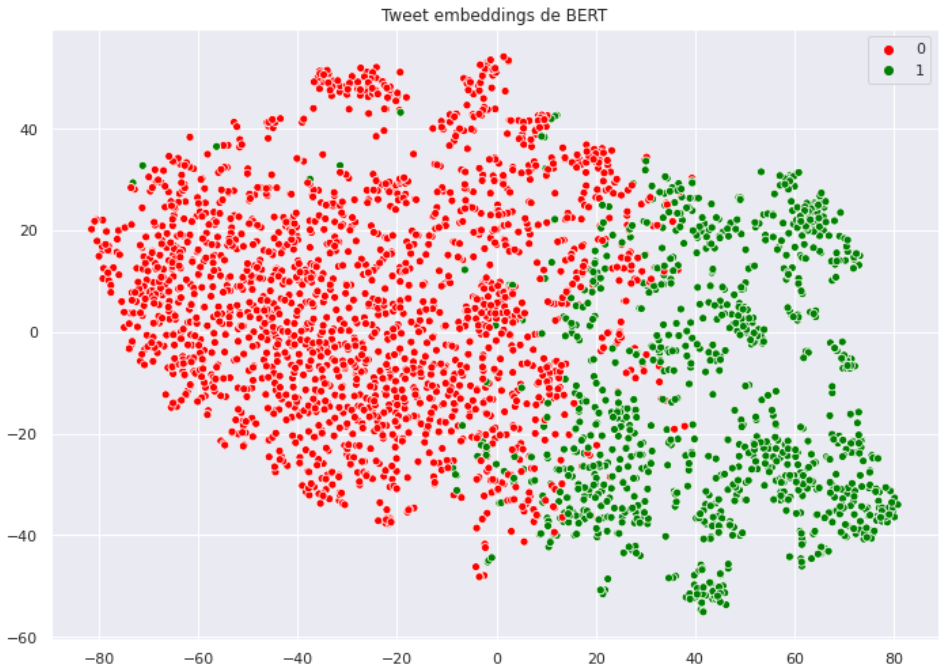

Figura 5.16

Para realizar esta visualización, se redujo los vectores de la salida [CLS] de la última capa de 768 dimensiones a 2. Utilizamos dos algoritmos, SVD, donde se redujeron estos vectores de 768 a 50 dimensiones, para no perder demasiada información, y luego TSNE, para reducir finalmente a 2 dimensiones. La razón por la cual no utilizamos directamente TSNE es que el algoritmo es muy costoso. A diferencia de lo visto en LSTM, notamos un poco mas de dispersión en las salidas. Incluso hay pequeños clusters dentro de los ambos grupos, tweets falsos repartidos entre tweets verdaderos y viceversa. Por último, podemos encontrar una visualización interactiva el mecanismo de atención en el script "visualizacion atencion bert.ipynb". Esta visualización se logro utilizando la librería BertViz.

XLNet XLNet es un modelo de lenguaje muy parecido a BERT, ambos son transformadores así que usan mecanismos de atención como base pero difieren en la forma que tienen de entrenar sobre un set de datos. A la hora de entrenar, BERT, reemplaza algunas de las palabras en la secuencia con el token especial [MASK] y luego busca reconstruir la secuencia original en base a todo el contexto (el resto de las palabras que no fueron "tapadas") previo y posterior. XLNet recorre la secuencia en ambos sentidos (atrás para adelante y adelante para atrás) y en cada lugar trata de predecir cual será la siguiente palabra. Esto significa que BERT utiliza toda la secuencia a la vez para predecir la palabra mientras que XLNet solo tiene todo lo anterior a la palabra actual. El problema con BERT es que supone que las palabras tapadas por el token [MASK] son independientes cuando tranquilamente podrían estar relacionadas. Además para mitigar el hecho de que XLNet solo puede recorrer en una dirección lo que se hace es realizar el mismo entrenamiento (predecir el siguiente en la secuencia) para diferentes permutaciones de las palabras.

Decidimos utilizar XLNet porque encontramos que era a superior a BERT en muchas tareas de NLP. El proceso que utilizamos para generar las predicciones es muy similar al de BERT. Utilizamos el set de datos con los textos corregidos y utilizamos un tokenizador especial para XLNet. Luego de tokenizar todos los tweets los paddeamos para que tengan siempre la misma longitud y se lo entregamos al modelo preentrenado para poder amoldarlo a nuestro set de datos. En total el modelo tiene más de 117 millones de parámetros entrenables así que es una herramienta muy poderosa y es muy posible que overfitee cuando lo entrenemos con nuestro set de 7600 datos.

Lamentablemente hasta el momento de escribir el informe no logramos que XLNet supere el score de BERT. El modelo overfitea muy rápido y lo máximo que pudimos lograr es un validation accuracy de 0,82.

6. Tuning de hiper-parametros

A medida que se fueron corriendo los distintos modelos se empezó a notar lo que ya se sabía: los hiper-parámetros son lo más importante de cada algoritmo, y la diferencia entre un buen modelo y uno promedio o incluso malo.

- **GridSearch:** Este proceso es muy costoso en tiempo. Si bien esto no encuentra la combinación óptima, al menos nos dio en algunos casos resultados favorables en comparacion a los probados manuealmente.
- **RandomSearch:** Con este método obtuvimos resultados cuestionables, este algoritmo realiza una búsqueda aleatoria de híper-parámetros, entonces nunca se podia saber con exactitud que conjunto iba a ser el mas adecuado.
- **HyperBand:** Es uno de los modelos de optimización de parametros que ofrece Kerastuner. Obtuvimos la mejor versión de las redes LSTM en muy poco tiempo. También lo utilizamos para CONV-1D.

Además las herramientas utilizadas para poder visualizar las métricas de cada algoritmo y tener una aproximación del Score fueron:

- RMSE
- Accuracy

7. Ensambles

7.1. Blending

El proceso de blending es uno de los que mejores resultados da en la creación de ensambles a partir de clasificadores diferentes. La idea es entrenar varios clasificadores diferentes y armar un set de datos con sus predicciones para luego entrenar otro clasificador que realice las predicciones finales en base a la combinación de los clasificadores. Probamos usando un blending de 3 modelos, CONV-1D, LSTM y BERT, y no obtuvimos resultados muy buenos, 78.823 %, esto se debe a que el modelo CONV-1D no utilizaba embeddings, su score individualmente era 75 %. Luego, probamos utilizando BERT y LSTM, obteniendo un score de 82.837 %, superior al anterior pero aun así no superior a BERT solo (aunque en promedio, todos los modelos que subimos de BERT a Kaggle rondan por este valor). Lo que obtuvimos para este ensamble es que el algoritmo como mínimo tiene mejor performance que el peor de los modelos y a lo sumo tiene la misma performance que el mejor. Esto puede deberse a que las redes pueden estar equivocándose en los mismos tweets al taggear.

Además se probo el blending con los arboles de decisión que mejor Score tuvieron: XGBoost de Clasificaciones y Naive-Bayes : Gaussiano. Nuevamente no obtuvimos buenos resultados, ya que cada árbol individual tenia un Score de 0.81336 y 0.801211 respectivamente y al realizar el blending se obtuvo un Score de 0.81152. Pareceria que el resultado es como una especie de promedio.

8. Conclusión

Este trabajo fue muy bueno y muy interesante para familiarizarnos con diferentes algoritmos de clasificación, probar múltiples librerías, entrenar distintos algoritmos y rescatar diferencias de performance y resultados entre ellos. Podemos decir que entrenamos distintos modelos y en base a sus resultados, combinar algunos de ellos mediante un ensamble para poder obtener aun mejores resultados. De esta manera, podemos decir que Machine Learning requiere de mucho mas que correr distintos algoritmos de predicción, hay que tener practica y saber de algunos 'trucos' para poder mejorar la predicción.

Una observación interesante es que con el agregado de features que a nuestro parecer eran de importancia para mejorar la predicción, el score empeoro notablemente. La mayoría de la información se encuentra en el texto de los tweets, y debido a eso, los algoritmos que usaron word embeddings o alguna manera de modelar el texto tuvieron mayor score que aquellos que no utilizaron ninguna representación de este. Además, notamos que ante mínimos cambios, ya sea de los valores de los hiper-parametros o el agregado de un feature, el score empeoraba o mejoraba de forma notable, demostrando el efecto avalancha.

Por otro lado, consideramos que aún quedarían distintas opciones que nos hubiese gustado realizar que no hemos podido, ya sea por falta de tiempo para estudiarlas en profundidad o porque lo hayamos intentado erroneamente:

- Probar mayor cantidad de algoritmos.
- Obtener mas features que hagan un aporte positivo al score.

A modo de reflexión personal consideramos que es bastante sorprendente que la implementación de una serie de algoritmos y el uso de una cantidad de features limitada pueda generar resultados con el nivel de predicción que se obtuvieron.

9. Código

En esta sección dejamos un acceso directo a nuestro Google Colab donde van a poder ver el código de todos los modelos presentes en el informe. https://drive.google.com/drive/folders/1wQUJx7RBNzZc5XzqQOA6RcAYaeXCAt5O?usp=sharing