FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2

[75.06] Organización de datos
13 de agosto de 2020
Primer cuatrimestre de 2020

Repositorio: https://github.com/IgnacioCarol/Tp2Datos

Alumno:	Bizari, Daniel
Número de padrón:	100445
Email:	dbizari@fi.uba.ar
Alumno:	Ignacio Carol Lugones
Número de padrón:	100073
Email:	icarol@fi.uba.ar
Alumna:	Cruz, Milagros
Número de padrón:	101228
Email:	mvcruz@fi.uba.ar
Alumno:	Torresetti, Lisandro
Número de padrón:	99846
Email:	ltorresetti@fi.uba.ar

Índice

1. Introducción

El objetivo principal de este trabajo es predecir, para cada tweet publicado, si está basado en hechos reales o no. Para ello, se probarán distintos algoritmos de Machine Learning y se determinará cuál arrojó un mejor resultado.

Se utilizará el set de datos de la competencia: https://www.kaggle.com/c/nlp-getting-started, el mismo con el que se elaboró el TP 1. Para aprovechar lo previamente explorado, se llevará a cabo una investigación sobre el trabajo anterior. Esto servirá no sólo para saber la composición del dataset y cuánto se relacionan sus columnas entre sí sino también para limpiar los datos antes de empezar con el proceso de predicción. A partir de ello, se procederá a crear los features que necesitan los modelos y luego, seleccionar los más importantes que serán los que finalmente se utilizarán. Además, se optimizarán los hiperparámetros de los algoritmos a probar para maximizar la precisión. Finalmente, se procederá a probar los modelos propuestos y luego, se extraerán conclusiones de lo observado.

2. Investigación previa

El primer paso fue investigar lo estudiado en el Trabajo Práctico 1, en el cual se hizo un análisis exploratorio del set de datos de la competencia: https://www.kaggle.com/c/nlp-getting-started. Recordando un poco la estructura del archivo train.csv, el mismo contaba con las siguientes columnas:

- id - identificador único para cada tweet
- text - el texto del tweet
- location - ubicación asociada al tweet
- keyword - un keyword para el tweet
- target - en train.csv, indica si se trata de un desastre real (1) o no (0)

Si bien en el TP 1 se analizaron todas las variables, ahora el enfoque estará en el target ya que es lo que se intentará predecir.

2.1. Relación keyword-target

Una inquietud que surgió fue si existe una conexión entre las keywords y el nivel de veracidad del tweet. Para investigar este tema, se agrupó por keyword y se analizaron los resultados tanto para tweets reales como para falsos. Lamentablemente, no se pudo afirmar que exista una relación entre estas dos variables, por lo que se puede suponer que, a la hora de entrenar los modelos, el feature 'keyword' no influirá mucho en las predicciones.

2.2. Relación location-target

Otra pregunta planteada fue si existe alguna relación entre la ubicación desde donde se mandó el tweet y la veracidad del mismo. En un primer acercamiento, se percibió que la mayoría de los tweets reales de este set de datos provienen de India, Pakistán, Arabia Saudita y Turquía. En cambio, los tweets falsos se concentran en China, Brasil, Sudráfrica, Kenia, Tanzania y Alemania. Esto resulta interesante ya que el feature 'location' podría resultar importante a la hora de entrenar a los algoritmos de Machine Learning.

2.3. Relación text-target

Para este caso, se extrajeron distintas propiedades del texto y se analizaron las posibles relaciones que estas pudieran tener con el target:

2.3.1. Relación longitud del tweet-target

Dado que se tiene un set de datos con tweets de distintas longitudes, una pregunta que surgió fue si la longitud de los tweets influye en la veracidad. Lo que se pudo concluir fue que los tweets reales tienden a tener una logitud mayor que los tweets falsos, por lo que la longitud del texto también podría resultar ser un feature interesante.

2.3.2. Relación hashtags-target

Dentro del texto del tweet, pueden aparecer (o no) distintos hashtags. En el TP 1, se analizó qué hashtags predominan en los tweets categorizados como reales según el target y cuáles predominan en los tweets falsos. Si bien hubo hashtags presentes tanto en los tweets reales como en los falsos, solamente en los tweets reales aparecieron como recurrentes #hiroshima y #earthquake y algunos hashtags más relacionados con desastres naturales/acontecimientos catastróficos. De manera análoga, algunos hashtags sólo aparecieron recurrentemente en los tweets falsos como #hot o #job. Estos tópicos recurrentes en cada tipo de tweet (real y no real) podrían ser útiles para crear nuevos features.

2.3.3. Relación desastres naturales - target

En base a lo mencionado previamente con respecto a las keywords, se armó un caso de estudio interesante en el trabajo anterior: desastres naturales. Se buscaron en el texto aquellas palabras que aludieran a desastres naturales y se agruparon los tweets a partir de eso; luego, se analizó la veracidad de los distintos conjuntos presentados[1].

Se pudo observar que la cantidad de tweets reales cambiaba según el grupo, siendo *debris* el que mayor porcentaje de veracidad presentó y *blizzard*, el que menos. Estos resultados se tendrán muy en cuenta a futuro.

2.3.4. Relación cantidad de hashtags/links/usuarios etiqueados-target

Un aspecto analizado en el TP1 fue la cantidad de hashtags, links y usuarios etiquetados que aparecían en la columna 'text'. Si bien no se ahondó mucho en la conexión que estas variables pudieran tener con la veracidad del tweet, se puede construir rápidamente una matriz de correlación para conocer la relación que tienen las distintas columnas entre sí:

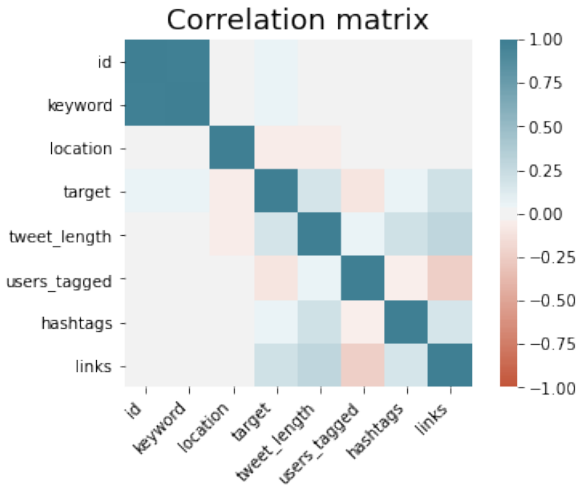

Figura 1: Matriz de correlación entre algunas de las variables del TP1

[1]Para más información, ver la sección **Caso de estudio particular: Desastres naturales** del TP1. Link https://github.com/d-bizari/Organizacion-de-datos

Se puede observar que el target se encuentra bastante relacionado con la longitud del tweet (como se comentó previamente), con la cantidad de links utilizados y, en menor medida, con la cantidad de hashtags. Esto se tendrá en consideración a la hora de llevar a cabo el *Feature Engineering*.

2.3.5. Relación sintáxis del texto-target

En este caso, se planteó la siguiente interrogante: ¿existe algún tipo de palabra que predomine dependiendo de la veracidad del tweet?. Para refrescar la memoria, se presenta el siguiente gráfico:

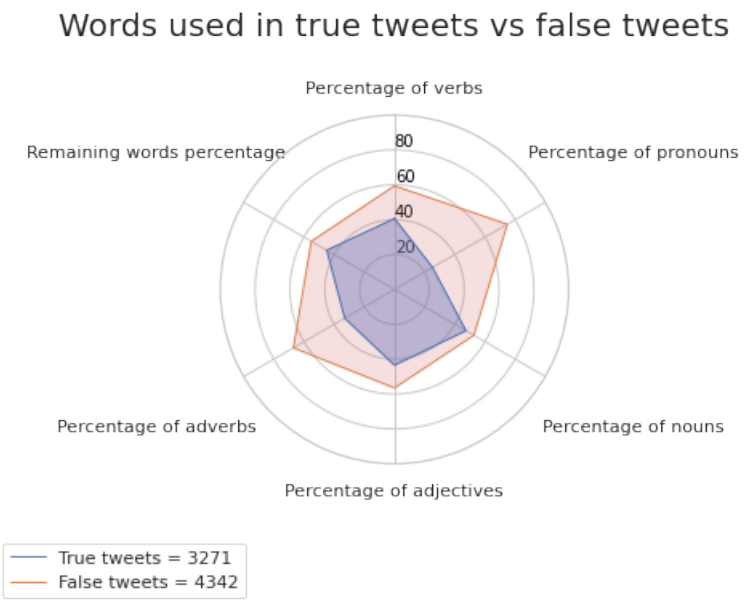

Figura 2: Tweets reales y falsos: distribución del tipo de palabras que los conforman

Lo que se concluyó en el trabajo anterior fue que para los tweets verídicos predominan los sustantivos mientras que para los tweets falsos, predominan los adverbios y pronombres. Lamentablemente, con los datos trabajados no se pudo establecer una relación entre el resto de las palabras que aparecen en el gráfico y la veracidad del tweet. Este resultado obtenido del trabajo anterior se utilizará más adelante.

3. Preprocesamiento del data set

Con el objetivo de optimizar las predicciones, se exploraron distintas formas de trabajar el dataset antes de llevar a cabo los procesos de *Feature Engineering* y, posteriormente, entrenamiento de los modelos.

Como primera instancia, se llevó a cabo una limpieza de datos. El preprocesamiento que se realizó es el mismo del TP1[2] y se hizo tanto para los datos del set train como para los del set de test; ya que si no, se estaría entrenando con una cosa y prediciendo para otra.

Luego, en base a lo comentado en la sección 2.3.3 **Relación desastres naturales - target**, se decidió dividir el set de entrenamiento en dos: los tweets que mencionan algún desastre natural y los que no. Para ello, se creó una etiqueta con 1 si aparece algún desastre natural en la columna *text* y 0 en caso contrario y la misma se utilizó como filtro para el fraccionamiento.

[2]Para más información, ver la sección **Limpieza de información** del TP1. Link https://github.com/d-bizari/Organizacion-de-datos

Finalmente, se probaron los 3 casos posibles: el set de datos sin dividir, el subset de datos cuyos tweets mencionan algún desastre natural y el subset de datos que no nombran ninguno.

Además, a modo de prueba, se utilizó train.csv con la limpieza de datos y sin ella para analizar qué arrojaba mejores resultados. La división final de los sets probados puede visualizarse a continuación:

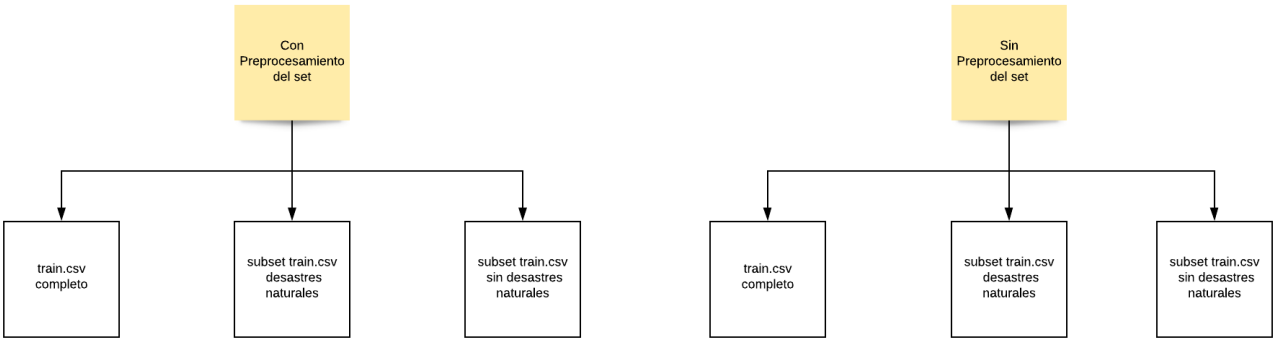

Figura 3: División de los distintos sets probados

Utilizando el set de datos limpio, se consiguió una mejor predicción creando features a partir del set de train completo. En contraparte, con el set de datos sin el preprocesamiento del TP1 se destacaron los sets de datos fraccionados. Al comparar los resultados obtenidos del set completo+preprocesamiento y subsets+sin preprocesamiento se pudo concluir que las predicciones óptimas se alcanzaron utilizando el set completo con la limpieza de datos, por lo que, a excepción de algunos casos particulares, se trabajó con este set para llevar adelante el proceso de creación de features que se explica a continuación.

4. Feature engineering

Los features, los cuales son los parámetros de entrada de los modelos a entrenar, pueden definirse como: *una porción de información o propiedad medible que es útil para construir aplicaciones de machine learning* . Cualquier atributo puede ser un feature siempre y cuando resulte provechoso a la hora de generar un buen algoritmo de predicción. Esta definición resulta interesante ya que implica que considerar a todas las variables como features resulta contraproducente, por lo que luego de crear distintos atributos a partir del set dado, será necesario seleccionar los que funcionen como "features"(es decir, aquellos que sean provechosos para los modelos de machine leargning).

Para empezar, se trabajó de forma particular con el texto de los tweets. Para ello, se tuvieron en cuenta las siguientes cuestiones:

- **Stopwords:** aquellas palabras vacías de contenido (como artículos, pronombres, proposiciones, etc) fueron eliminadas ya que no aportan un valor significativo al análisis.
- **Caracteres repetidos:** muchas veces, al escribir una palabra se repite un mismo caracter varias veces. En estos casos, los caracteres repetidos fueron eliminamos (por ejemplo: "happyyyyyy"pasó a ser "happy"luego de la limpieza)
- **Contracciones:** en inglés, las palabras se pueden acortar eliminando algunas de las letras y colocando un apóstrofe adelante. En este trabajo, se restauraron las palabras sin contracciones (por ejemplo “I’m” pasó a ser “I am”).
- **Puntuación:** se elimina ya que solamente se trabajará con las palabras y sus significados, no con oraciones como construcción.

Luego de esto, se dividieron los posibles features en las siguientes categorías para hacer un análisis mas certero:

- Composición del tweet
- Características sintácticas del tweet
- Vectorización del tweet
- Operaciones generales sobre las variables

4.1. Composición del tweet

Este tipo de features analiza las características "físicas"del texto tal y como la cantidad de palabras que tiene, su longitud, la cantidad de keywords que aparecen, etc. A continuación se muestran aquellos features que pertenecen a esta categoría:

`['textProm', 'textMean', 'textMeanStd', 'tweet_length', 'length_proportion', 'tweet_size', 'keywordAppearance', 'norm2', 'error_percentage', 'textStd', 'amount_of_words_proportion', 'textMeanMean']`

Si bien en el trabajo general se usó el dataset limpio, ciertos features fueron creados como una razón entre el valor del tweet del data set preprocesado y sin limpiar, como por ejemplo *Amount of words proportion, Length proportion*, etc.

4.2. Características sintácticas del tweet

Estos features se fundamentan en analizar sintácticamente el tweet; es decir, se focalizan en el tipo de palabras que contiene, la cantidad que hay de cada una(cantidad de verbos, sustantivos, adverbios, etc), usuarios etiquetados, hashtags y links, entre otras cosas. Los features que se obtuvieron para esta categoría se mencionan a continuación:

`['verbs', 'stopTrue', 'pronouns', 'adverbs', 'hashtags', 'nouns', 'hasTopUser', 'Porn_words', 'alphaTrue', 'links', 'Natural_disaster', 'users_tagged' 'hash_users', 'links_hash', 'links_users', 'Total_elements']`

Para la creación de algunos de los features se utilizó la información de ciertas páginas web. Por ejemplo, se usó una página [3] que contiene los usuarios mas populares de twitter para generar el feature *hasTopUsers*, el cual indica si un tweet menciona a una de las 100 cuentas mas famosas de Twitter. Otra de las páginas empleadas fue la de EM-DAT[4], la cual contiene un dataset con muchos desastres naturales (como por ejemplo *earthquake* o *flood*) que permitió definir el feature *Natural disaster* que indica si un tweet hace referencia a un desastre natural o no.

Siguiendo la línea de features que indican si el tweet hace una referencia a cierto tópico o no se creó *Porn words*, ya que, tal y como se comentó en la sección 2.3.2 **Relación hashtag-target**, muchos tweets falsos del dataset contienen palabras relacionadas al porno.

4.3. Vectorización del tweet

Utilizando los features mencionados en las dos secciones anteriores, la máxima accuracy obtenida fue de aproximadamente 0.72, lo cual es bastante bajo. Para mejorarla, se decidió agregar nuevos features centrados en *Natural Language Processing (NLP)*. Entre la variedad de algoritmos disponibles se optó por utilizar, en primera instancia, *Word2Vec*.

[3] https://es.wikipedia.org/wiki/Anexo:Cuentas_de_Twitter_con_más_seguidores
[4] Página de EM-DAT https://www.emdat.be/classification

4.3.1. Word2Vec

Word2vec se desarrolla utilizando redes neuronales de dos capas. Toma una gran cantidad de datos de texto o corpus de texto como entrada y genera un conjunto de vectores a partir del texto dado.

En una primer momento se trató de obtener un modelo de word2Vec basado en el texto del train dataset. Luego, se utilizó un set de datos obtenido de Google que ya se encuentra entrenado, el mismo se llama *GoogleNews-vectors-negative300*. Al entrenar los algoritmos de Machine Learning con las codificaciones de estos dos modelos, se pudo observar que al usar los features del modelo entrenado con el set de Google hay una mejora significativa en respecto al accuracy. La desventaja de esta decisión es que el set provisto por Google posee un tamaño 1,5 GB, lo que implica un alto costo computacional a la hora de entrenar el modelo de word2vec.

Luego de obtener el modelo entrenado con el data set de Google, se realizó una suma componente a componente de los vectores de las palabras que posee el tweet y el resultado final de esta sumatoria se lo insertó en el data frame, en el cual cada componente del vector equivale a una columna nueva.

Realizando el agregado de estas columnas, con los mismos algoritmos de machine learning, el accuracy pasó de 0.72 a 0.8 aproximadamente, mostrando una clara mejora.

4.3.2. BERT

BERT (Bidirectional Encoder Representations from Transformers) es un encoder desarrollado por investigadores de Google AI. Como su nombre lo indica, está basado en *Transformers*, los cuales transforman una secuencia de elementos (como en este caso, secuencias de palabras en un tweet) en otra secuencia, también conocido como arquitectura *Seq2Seq*. Por lo general, los *Transformers* constan de 2 partes: un codificador y un decodificador. El codificador se encarga de recibir el input de secuencias y mapearla a una dimensión más alta (vector de N dimensiones); luego, ese vector abstracto sirve de alimentación para el decodificador que lo transforma en otra secuencia. Esta secuencia de salida puede ser otro lenguaje, conjunto de símbolos, etc. Se puede pensar esta relación codificador - decodificador como dos traductores humanos: uno por ejemplo habla francés y un idioma imaginario, y el otro habla el mismo idioma imaginario y alemán. Inicialmente ninguno de los dos conoce bien el idioma imaginario, por lo que hay que entrenarlo con muchos ejemplos para que puedan aprenderlo.

Ahora entra una pieza clave de *BERT* que es el *Attention-mechanism*, este observa una secuencia de input y decide en cada paso qué partes de las secuencias son importantes. Por ejemplo, en este momento el lector está concentrando en las palabras que contiene este informe pero a la vez retiene en la cabeza la información importante de forma tal que tenga contexto. Un mecanismo de atención funciona de manera similar para una secuencia, volviendo un poco al ejemplo del codificador y decodificador humano, se puede imaginar que en vez de solo escribir la traducción el codificador también escribe las palabras claves que son importantes semánticamente para la oración y se lo informa al decodificador.

BERT justamente utiliza transformaciones con mecanismos de atención que aprenden relaciones contextuales entre palabras en un texto. Como el objetivo es generar un modelo del lenguaje, sólo el mecanismo del codificador será necesario. Se dice que es bidireccional porque, a diferencia de los modelos direccionales que leen un text input secuencialmente (izquierda a derecha), *BERT* lee la secuencia entera de palabras. Esta característica permite al modelo aprender el contexto de una palabra.

La gran desventaja es que toma mucho tiempo y necesita un montón de datos para entrenarse, pero en el Hub de *Tensorflow* se cuentan con varios modelos ya entrenados que se pueden cargar y reutilizar. En este trabajo, se utilizó el siguiente modelo https://tfhub.dev/tensorflow/bert_en_uncased_L-24_H-1024_A-16/1 que fue pre-entrenado para el inglés con textos de Wikipedia y BooksCorpus.

Ahora, para poder utilizar el modelo precargado es necesario realizar previamente un proceso al texto, que sería *Tokenizar* cada tweet y, particularmente, se utilizan 2 token especiales que son:

- *[CLS]* para marcar el inicio de la sentencia.
- *[SEP]* se utiliza como separador entre sentencias.

Luego de *Tokenizar* el tweet se deben generar 3 vectores asociados a ese tweet:

- **Token Embeddings:** son una representación numérica de las palabras en la sentencia de entrada, en nuestro caso particular los tweets. También hay algo llamado *sub-word tokenization* que *BERT* usa para partir palabras largas o complejas en palabras más simples y después convertirlas en tokens. Por ejemplo, para la palabra "playing" se parte en "play" y "##ing" antes de generar los tokens. Esto hace que pueda entender mejor el contexto de la palabra en vez de tratar a cada palabra como una nueva.
- **Segment Embeddings:** son usados para ayudar a *BERT* a distinguir entre diferentes sentencias en un input. Los elementos de este vector serán los mismos para las palabras de la misma sentencia y el valor cambia para una sentencia diferente. En este caso, como se tokenizó un tweet por input, el vector siempre tendrá sus componentes en 0 respetando el largo del vector de Token embedding.
- **Mask tokens:** este vector sirve para indicarle a *BERT* las componentes importantes del vector de token embeddings y cuáles simplemente son padding. Ya que *BERT* funciona en ancho fijo, como máximo 512 dimensiones, en las especificaciones del paper indica que crece cuadráticamente según las dimensiones usadas, y dado que los tweets se basan en textos cortos, se decidió dejar un ancho fijo de 160 dimensiones.

Estos 3 vectores se pasarán como input al *bert-layer*, que es el modelo precargado del que hablamos anteriormente que consta de una red neuronal pre-entrenada con una arquitectura de Transformers, y este devolverá 2 outputs:

- **pooled-output** que tiene la forma *[batchSize, 1024]* con representaciones para todas la secuencias de entradas.
- **sequence-output** que tiene la forma *[batchSsize, maxLength, 1024]* con representaciones para para cada token de entrada. En la mayoría de las fuentes indican que con usar solo el output correspondiente al primer token funciona muy bien para tareas de clasificación. En este trabajo, se usó solo esto y dio muy buenos resultados.

Este encoder se utilizó exclusivamente para los modelos de redes neuronales. Se probó utilizar *BERT* tanto para el texto limpio (sin stopwords, caracteres repetidos, etc como se detalla al principio de esta sección) como para el texto en crudo. Lo que se pudo observar es que se obtuvo ligeramente un mejor resultado en la predicción para los features generados a partir del texto crudo. Esto podría llegar a deberse a que en la etapa de *Tokenization* se hace internamente una limpieza/procesamiento de texto, como por ejemplo partir una palabra compleja en subwords como se mencionó anteriormente. Evidentemente, la limpieza exhaustiva hecha por este equipo estaba eliminando ciertas partes que a *BERT* le resultaron útiles.

4.4. Operaciones generales sobre los features

Además de todo lo comentado, se realizaron ciertas operaciones aritméticas sobre los features, como por ejemplo una normalización estándar, promedio, media, etc. El objetivo de este proceso fue aplicar técnicas estadísticas para acotar los valores de los distintos tipos de tweets entre si (es decir que los tweets verídicos se acerquen entre si al igual que los falsos, pero no que se acerquen ambos tipos de tweets).
En su totalidad, los features creados por este medio son los siguientes:

```
    ['alphaTrue_minMaxNorm', 'amount_of_words_proportion_minMaxNorm',
'hashtags_standardNorm', 'links_natNat', 'links_minMaxNorm',
'error_percentage_minMaxNorm', 'Total_elements_minMaxNorm',
'links_users_minMaxNorm', 'Total_elements_standardNorm', 'length_proportion_standardNorm',
'users_tagged_natNat', 'tweet_size_standardNorm', 'Porn_words_minMaxNorm',
'tweet_length_minMaxNorm', 'pronouns_standardNorm', 'verbs_minMaxNorm',
'alphaTrue_standardNorm', 'nouns_minMaxNorm', 'nouns_standardNorm',
'hash_users_minMaxNorm', 'amount_of_words_proportion_standardNorm',
'text', 'Valid_location', 'Porn_words_standardNorm', 'links_users_natNat',
'hashtags_minMaxNorm', 'links_hash_natNat', 'links_hash_minMaxNorm',
'links_users_standardNorm', 'users_tagged_minMaxNorm', 'textMeanMean_minMaxNorm',
'adverbs_minMaxNorm', 'hasTopUser_standardNorm', 'length_proportion_minMaxNorm',
'stopTrue_standardNorm', 'tweet_length_standardNorm', 'hash_users_standardNorm',
'error_percentage_standardNorm', 'verbs_standardNorm', 'tweet_size_minMaxNorm',
'hash_users_natNat', 'hashtags_natNat', 'links_hash_standardNorm',
'hasTopUser_minMaxNorm', 'Porn_words_natNat', 'links_standardNorm',
'users_tagged_standardNorm', 'adverbs_standardNorm', 'textMeanMean_standardNorm',
'pronouns_minMaxNorm', 'stopTrue_minMaxNorm']
```

Ahora bien, antes de entrenar a los algoritmos se deben llevar a cabo dos procesos que tienen un gran impacto en la performance obtenida.

5. Selección de features

Los features con los que se entrenan a los modelos influyen mucho en los resultados obtenidos. Por ello, es muy importante elegir cuáles utilizar mediante el proceso de *Feature Selection*. Llevar a cabo esto correctamente trae consigo las siguientes ventajas:

- **Menos tiempo de entrenamiento:** al reducir la cantidad de variables, el costo computacional disminuye por lo que el entrenamiento se vuelve más rápido.
- **Reducción del overfitting:** tomando solamente los features más relevantes, se asegura que las variables ruidosas sean descartadas, lo que provoca que se reduzca el overfitting.
- **Modelo más sencillo de interpretar:** el modelo se vuelve menos complejo y, en consecuencia, es más sencillo de interpretar.
- **Aumenta la precisión de las predicciones:** si el set de features seleccionado es apropiado, aquellas variables que entorpezcan la predicción (como las que contengan mucho ruido, por ejemplo) habrán sido eliminadas, logrando una mejor precisión en las predicciones.

Si bien existen diversos métodos para seleccionar qué features usar, en este trabajo se utilizó una técnica de filtrado. En primer lugar, se hizo una limpieza de features: se eliminaron los features constantes, cuasi constantes y aquellos que proporcionaran información redundante.

Luego, se tomó la *Feature Importance* para determinar qué features utilizar. Para ello, se usó la biblioteca **feature_selector** [5], la cual tiene una función que permite calcular la importancia de los features entrenando el modelo *LGBMClassifier* utilizando una métrica determinada. Como en este trabajo se encaró un problema de clasificación binario, se utilizó la métrica **AUC** (Area Under the Curve). Finalmente, se graficaron los features más importantes:

[5] link del repositorio: https://github.com/WillKoehrsen/feature-selector/blob/master/feature_selector/feature_selector.py

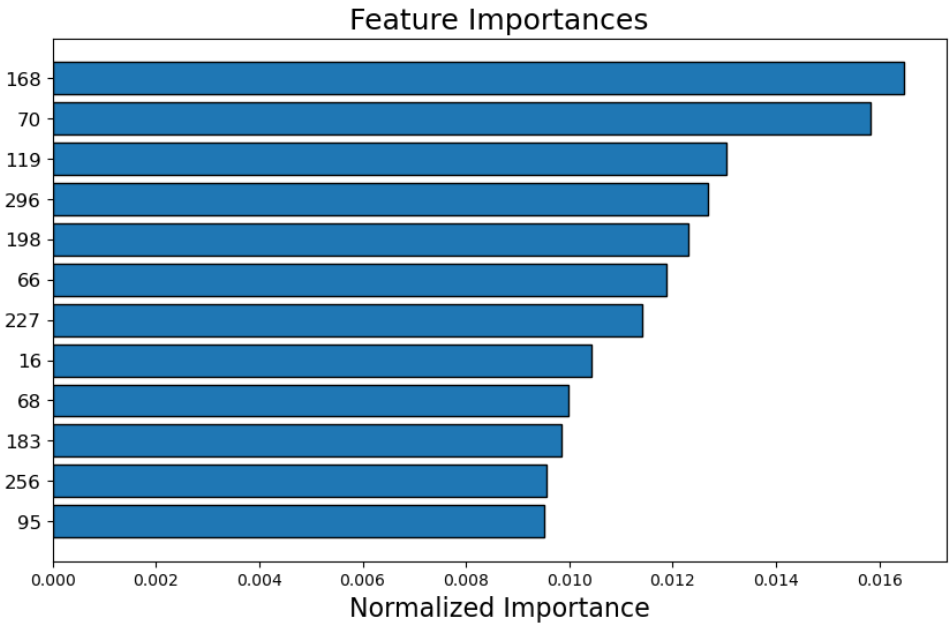

Figura 4: Features más importantes

Sin embargo, todavía queda una incógnita: ¿con cuántos features se debe trabajar? Para resolver esto, se graficó la importancia acumulada conforme va aumentando el número de features. En este trabajo práctico, se considera óptimo que la importancia acumulada por los features sea de un 95 %, lo cual establece un límite en el número de features tal y como se ve a continuación:

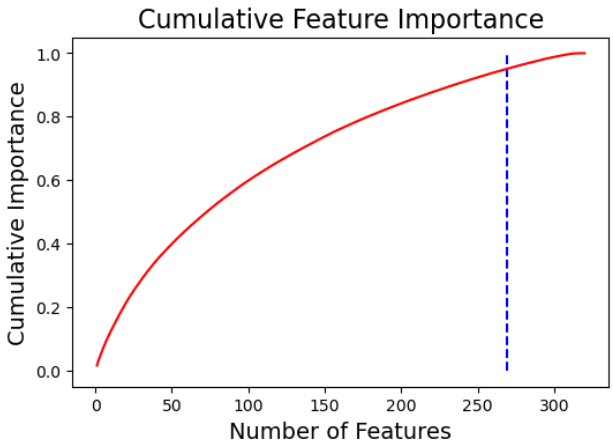

Figura 5: Importancia acumulada de los features

Se puede observar que con los primeros 270 features mejor rankeados se puede lograr una predicción precisa ya que en ellos se concentra el 95 % de la información requerida para entrenar a los modelos.

6. Optimización de hiperparámetros

Los hiperparámetros tienen un gran impacto en la exactitud de la predicción del modelo. Dado que el entrenamiento del algoritmo no determina cuáles son los mejores hiperparámetros a utilizar,

es necesario llevar a cabo el proceso de *Hyperparameter Tuning* para poder encontrarlos antes de entrenar a los modelos.

En este trabajo, se utilizará el algoritmo de *GridSearch* para poder realizar esta optimización, el cual prueba todas las combinaciones posibles y se queda con la que se obtuvo el mejor resultado. Si bien el costo computacional es alto, el data set con el que se trabaja no es muy grande. Además, se corre el algoritmo una vez, se copian los resultados y se los colocan directamente en las funciones *train* de los modelos. De esta forma, el tiempo de entrenamiento del algoritmo no depende del tiempo de GridSearch ya que no es necesario llevar a cabo esta búsqueda cada vez que se quiera entrenar un modelo.

7. Algoritmos probados

7.1. Decision Tree Classifier

Los árboles de decisión son algoritmos de machine learning supervisados usados para problemas de regresión y, como en este caso, clasificación. El objetivo es crear un modelo que prediga el valor buscado (el target) aprendiendo reglas de decisión basadas en los features. De esta forma, se forma una especie de árbol en donde hay una rama para cada valor de comparación y, en cada nodo, se divide el set de datos de acuerdo a un cierto criterio. Una vez entrenado el modelo, el objetivo es ir recorriendo el árbol hasta llegar a un nodo hoja, el cual indica a qué clase pertenece el dato tomado y de esta forma, poder clasificarlo.

En este trabajo, se utilizó la implementación DecisionTreeClassifier de la librería sklearn y se obtuvo una precisión de 0.6850393700787402. Este valor es extremadamente bajo en comparación con los otros que fueron obtenidos, por lo que se puede afirmar que, en este caso, un árbol de decisión no es óptimo para hacer predicciones.

7.2. Gaussian Naive Bayes Classifier

Los métodos de Naive Bayes son un conjunto de algoritmos de aprendizaje supervisado que se basan en aplicar el teorema de Bayes bajo la hipótesis de que todo par de features son condicionalmente independientes entre si. Dado el vector features "Xƶ la variable "Y", el teorema de Bayes afirma lo siguiente:

$$P(y|x_1, ..., x_n) = \frac{P(y)P(x_1, ..., x_n|y)}{P(x_1, ..., x_n|y)}$$

Luego, utilizando la hipótesis mencionada, se obtiene lo siguiente:

$$P(y|x_1, ..., x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, ..., x_n|y)}$$

Dado que $P(x_1, ..., x_n)$ es constante, se puede utilizar la siguiente regla de clasificación:

$$y = arg_y max P(y)\prod_{i=1}^{n} P(x_i|y)$$

estimando $P(y)$ y $P(x_i|y)$ con Maximum A Posteriori (MAP).

En este trabajo, se utilizó el método GaussianNB perteneciente a la librería sklearn. El mismo asume que la probabilidad de los features tiene una distribución Gaussiana:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}}\left(-\frac{(x_y - \mu_y)^2}{2\sigma_y^2}\right)$$

Finalmente, al entrenar este modelo y utilzarlo para predecir, se obutvo una precisión de 0.6968503937007874. Esto no varía mucho del caso anterior, por lo que se siguió probando con otros modelos.

7.3. Passive Aggressive Classifier

Los algoritmos Pasivo-Agresivos llevan ese nombre debido a la forma en la que operan:
Pasivo: si la predicción es correcta, el modelo se mantiene y no se aplica ningún cambio.
Agresivo: si la predicción es incorrecta, el modelo se cambia para corregirlo.[6]

Para este trabajo, se utilizó el método PassiveAggressiveClassifier, el cual se encuentra en la librería sklearn.linear_model. Para este caso, solamente se trabajó con la columna utilizó la columna 'text' de la siguiente forma:

1. Se divide el en train y test en una proporción 80 - 20.
2. Tanto para el set de train como el de test, se limpia el texto eliminando las stopwords y aquellas palabras demasiado frecuentes. Luego, se transforma el texto en una representación numérica para ser pasado como input al modelo utilizando Tfidfvectorizer.
3. Se entrena el modelo con el set de train con los features generados por Tfidfvectorizer.
4. Se predice utilizando el modelo con el set de test (el cual también contiene features de Tfidfvectorizer).

Además de probar un algoritmo nuevo, también se buscó explorar nuevas alternativas para la codificación del texto. Al calcular la exactitud obtenida con el método accurracy_score, se obtuvo un 0.7662508207485227, por lo que se hubo una mejora importante con respecto del caso anterior. No obstante, no es un valor tan alto, por lo que se siguió explorando otros modelos.

7.4. KNN Classifier

K-Nearest-Neighbor es un algoritmo de tipo supervisado basado en instancia. Esto último quiere decir que no aprende explícitamente un modelo sino que memoriza las instancias de entrenamiento que son usadas como "base de conocimiento" para la fase de predicción.

Sirve para clasificar valores buscando los puntos de datos "más similares" aprendidos en la etapa de entrenamiento y haciendo conjeturas de nuevos puntos basándose en esa clasificación. El mecanismo es el siguiente:

1. Calcular la distancia entre el dato a clasificar y el resto de los datos del dataset de entrenamiento.
2. Seleccionar los "k" elementos más cercanos (con menor distancia, según la función que se use).
3. Realizar una "votación de mayoría" entre los k puntos: aquella clase/etiqueta predominante será su clasificación final.

En este trabajo, se utilizó como métrica la distancia euclídea y k=10 como cantidad de vecinos a tomar. La exactitud obtenida fue de 0.7755905511811023, el cual no varía tanto con la obtenida anteriormente.

7.5. Regresión logística

La Regresión Logística es un algoritmo supervisado y se utiliza para clasificación. A partir de un conjunto de datos de entrada (features), la salida será discreta (y no continua) por eso se utiliza Regresión Logística (y no Regresión Lineal), Se utilizó la implementación del paquete sklearn en Python y la selección de features previamente obtenido para ponerlo en práctica. En este caso, el método accurracy_score arrojó un resultado de 0.7957977675640184, el cual es mayor por 0.2 (apróximandamente) que el obtenido con KNN.

[6]Por motivos de complejidad, se omite en este trabajo toda la matemática detrás de los algoritmos pasivo-agresivos. Para más información, visitar: https://www.bonaccorso.eu/2017/10/06/ml-algorithms-addendum-passive-aggressive-algorithms/

7.6. Support Vector Machine

SVM es un algortimo cuyo objetivo es encontrar un hiperplano en un espacio N-dimensional, siendo N el número de features, que establezca límites que permitan clasificar los datos: dependiendo de la posición que tengan los datos con respecto al hiperplano, pertenecerán a una clase o a otra.

Si bien hay muchos hiperplanos que permiten separar los datos en dos clases, el objetivo es encontrar aquel hiperplano cuya distancia entre los puntos de las dos clases sea máxima; es decir, se busca maximizar el margen. De esta forma, los futuros puntos se clasificarán con más confianza.

Se utilizó la implementación de Python SVM, nuevamente de la librería sklearn, y se obtuvo una exactitud en las predicciones de 0.8057742782152231. Presenta una mejora de 0.01 aproximadamente con respecto de regresión logística pero se siguió investigando para obtener un valor más alto. Un inconveniente al utilizar este algoritmo es que el mismo demora un tiempo considerable en ser ejecutado.

7.7. Gaussian Process Classifier

Los Procesos Gaussianos (GP) son métodos genéricos de aprendizaje supervisado utilizados para resolver problemas de regresión y clasificación. En este trabajo, se utilizó la implementación GaussianProcessClassifier de la librería sklearn, el cual se basa en la aproximación de Laplace.

Si bien la exactitud obtenida fue 0.8083989501312336, mayor que la obtenida en SVM, resultó ser uno de los algoritmos más lentos utilizados en el caso de que se utilice el dataset completo. Sin embargo, cuando se trabaja con el mismo fraccionado en desastres naturales y no naturales la exactitud obtenida no disminuye, pero el tiempo de ejecución de este algoritmo se redujo significativamente.

7.8. SGD Classifier

Es un algoritmo de clasificación lineal optimizado con aprendizaje del tipo gradiente descendiente estocástico (SGD), el cual es un método de optimización numérica para estimar los mejores coeficientes y se realiza punto a punto. Al ser una optimización y no un clasificador en sí, se puede controlar el modelo a ajustar; en este trabajo, se utiliza SVM (modelo a ajustar por defecto).

La exactitud de la predicción oscilaba, para el mismo set de entrenamiento, entre 0.3701657458563536 y 0.8011049723756906. Sorprende el amplio rango en el que varía la predicción, ya que trabajando con el mismo set de train se esperan valores muy similares entre sí, por lo que no se consideró aplicar esta optimización a ningún algoritmo para predecir en la competencia.

8. Ensambles

En función de obtener una mejor performance en la predicción, se decidió probar distintos métodos de ensamble, los cuales combinan distintos algoritmos de machine learning. Existen distintos tipos de ensamble, a saber:

- **Bagging:** consiste en aplicar el mismo clasificador n veces usando Bootsrapping, un mecanismo de entrenamiento en el cual se entrena a cada modelo tomando un subset random de entrenamiento con reeplazo (es decir, se repiten algunos registros para que el subset no contenga todos los registros del set de entrenamiento original). Luego, se promedian sus resultados.
- **Boosting:** consiste en combinar los resultados de varios clasificadores débiles para obtener un clasificador robusto. A medida que se va entrenando cada algoritmo, el siguiente algoritmo a entrenar le dará mayor peso a los resultados en los cuales el anterior tuvo una mala performance (es decir, va corrigiendo los resultados erróneos en cada iteración). Finalmente, el resultado final se obtiene a base de un promedio ponderado.

8.1. Bagging Classifier

Se utilizó la librería scikit-learn, la cual ofrece varios métodos de bagging unificados unificados en BaggingClassifier. Este modelo se encuentra comentado en el código ya que, como el proceso de entrenamiento es muy lento, no se pudo predecir la veracidad de los tweets. No obstante, se probó utilizar otro algoritmo que incluya bagging y que sea más rápido como el que se presenta a continuación.

8.2. Random Forest Classifier

Dado que el algoritmo Decission Tree no dio muy buenos resultados, se decidió probar un ensamble de árboles como lo es Random Forest, el cual consiste en aplicar Bagging sobre árboles de decisión. Se utilizó RandomForestClassifier, perteneciente a la libería sklearn, y se obtuvo una exactitud de 0.800524934383202. Si bien es una mejora considerable al obtenido con un solo árbol de decisión, no es el valor máximo obtenido hasta ahora (el cual corresponde a Gaussian Process Classifier). Por lo tanto, se siguió explorando con otros tipos de ensamble.

8.3. ExtraTrees Clasifier

Es similar a Random Forest pero difiere en la construcción de los árboles de decisión, ya que en cada paso se utiliza todo el set de entrenamiento y los nodos se forman de manera aleatoria. Esto difiere del algoritmo anterior en el cual se aplica bagging sobre los árboles y los nodos se forman buscando el mejor split. Debido a que los splits se hacen de manera aleatoria, su costo computacional es más bajo que el de Random Forest, por lo que se probó este modelo para analizar sus resultados. La exactitud obtenida fue de 0.8057742782152231, mejor que la obtenida anteriormente. No obstante, la óptima de las mencionadas corresponde a Gaussian Process Classifier y se siguió probando con otros ensambles para mejorar esa predicción.

8.4. AdaBoost Classifier

Es un caso de boosting en donde se usa como algoritmo de clasificación a un simple separador del set de entrenamiento en 2. El funcionamiento es el siguiente:

- Se crea un clasificador sencillo (árbol de decisión de profundidad 1) y se clasifica la primera iteración.
- Se les dará mayor peso para la siguiente clasificación a aquellas muestras que han sido mal clasificadas.
- Se vuelve a crear un árbol para la siguiente iteración. Y así sucesivamente.
- Finalmente, se promedian todas las clasificaciones ponderadas y se obtiene la clasificación final. (Los pesos se corresponden de forma inversamente-proporcional al error de cada clasificador).

En este trabajo, se utilizó AdaBoostClassifier, de la libería sklearn, y se obtuvo una exactitud de 0.7821522309711286. Este es el menor valor obtenido de todos los ensambles probados, por lo que se decidió intenter con otros modelos de boosting para analizar si mejora la predicción.

8.5. Gradient Boosting Classifier

Gradient boosting es otro caso de boosting que implica tres elementos:

- **Función de perdida a optimizar:** la principal característica que debe poseer es que sea diferenciable y, dependiendo del tipo de problema, cambiará la función. Por ejemplo, para problemas de clasificación se puede utilizar una pérdida logarítmica o una entropía cruzada.

- **Algoritmo de aprendizaje débil para hacer las predicciones:** en Gradient boosting se utilizan árboles de decisión. Es común restringirlos de manera específica para asegurar que el algoritmo permanezca débil. Se suelen restringir el número máximo de capas, nodos, divisiones u hojas.
- **Modelo aditivo:** sirven para añadir los algoritmos de aprendizaje débiles que minimizan la función de perdida. Los árboles de decisión son agregados de a uno a la vez y los árboles existentes en el modelo no cambian. Para determinar los parámetros que tendrán cada uno de los árboles de decisión que son agregados al modelo se utiliza un procedimiento de gradiente descendiente que minimizará la función de perdida. De esta forma, se van agregando árboles con distintos parámetros de forma tal que la combinación de ellos minimiza la pérdida del modelo y mejora la predicción.

Utilizando la implementación GradientBoostingClassifier de sklearn, se obtuvo una exactitud de 0.8031496062992126. Si bien se ve una mejora respecto del otro tipo de boosting comentado (AdaBoost), no supera el máximo valor obtenido por lo que se probó con otra variante de boosting

8.6. LightGBM Classifier

LightGBM Classifier es un algoritmo de machine learning que pertenece a LightGBM, un framework de gradient boosting que utiliza algoritmos de aprendizaje basados en árboles. Lo que lo diferencia de otros algoritmos es que el árbol crece verticalmente mientras que en otros, crece horizontalmente. Fue de interés probar este algoritmo ya que creció en popularidad en el último tiempo por los siguientes motivos:

- Alta velocidad de entrenamiento y buena eficiencia.
- Bajo uso de memoria.
- Buena exactitud de predicciones.
- Capacidad de manejar gran cantidad de data.

Utilizando este algoritmo, se obtuvo una exactitud de 0.8175853018372703, la más alta hasta el momento.

8.7. XGBoost Classifier

Este algoritmo es otra implementación open-source eficiente y popular del algoritmo *Gradient Boosted Trees*. Es una combinación perfecta de técnicas de optimización de software y hardware para producir resultados superiores utilizando menos recursos informáticos en el menor tiempo posible. La exactitud de la predicción con este algoritmo fue de 0.8070866141732284, la cual es menor que la obtenida con LightGBM Classifier.

8.8. Voting Classifier

La idea detrás del VotingClassifier tomar los resultados de distintos clasificadores y ver, para cada dato, cuál es la clase que obtuvo la mayoría de los votos. Un clasificador de este tipo puede ser útil para un conjunto de modelos de igual buen rendimiento con el objetivo de equilibrar sus puntos débiles. Este algoritmo se utilizó únicamente para el caso del dataset fraccionado, ya que la búsqueda de la mejor combinación de algoritmos a utilizar es un proceso costoso teniendo en cuenta el tiempo. Para el caso de los desastres naturales los algoritmos utilizados por el Voting classifier fueron *Random Forest, XGBoost y Gaussian Classifier* obteniendo una exactitud de 0.856353591160221. Para los otros tipos de desastres los algoritmos utilizados fueron *Random Forest, XGBoost y Gradient Boosting Classifier* obteniendo una exactitud de 0.8264604810996563.

Para poder llevar a cabo este ensamble, se implementó un método que determina cuál es la combinación de algoritmos con la que se obtiene una mejor predicción. Luego, se procede a realizar la clasificación con el resultado obtenido de esta función.

8.9. Neural Networks

Es un algoritmo de aprendizaje supervisado que aprende, entrenándose con un dataset, una función de la forma $f(X) : R^m \rightarrow R^o$, en donde m es el número de dimensiones de entrada (es decir, la cantidad de features) y o, el de salida (lo que se busca predecir). Se diferencia de la regresión logística debido a que entre la capa de entrada y la capa de salida puede haber una o más capas, llamadas *hidden layers*

La capa de entrada consiste en un conjunto de neuronas $x_i|x_1, ..., x_m$ que representa los features que se le pasan al modelo. Cada neurona de la *hidden layer* transforma los valores de la capa anterior con una sumatoria linear ponderada $w_1x_1 + w_2x_2 + ... + w_mx_m$ seguida de una función de activación no lineal $g(X) : R \rightarrow R$ (como lo puede ser, por ejemplo, la función hiperbólica, ReLu, sigmoide, etc).

La capa de salida recibe los valores provenientes de la última *hidden layer* y los transforma en los valores de salida que estos serán las clases con las que queremos clasificar los datos, para esta última capa se suele usar una regresión *Softmax*, que es una generalización de la regresión logística, dado que nos da una medida de probabilidad de pertencer a cada clase.

8.9.1. Multi-layer Perceptron Classifier

Multi-layer Perceptron (MLP) es un caso particular de redes neuronales. Para este trabajo, se utilizó MLPClassifier, perteneciente a la librería sklearn, el cual implementa un algoritmo multi-layer perceptron que entrena utilizando Backpropagation. La exactitud obtenida fue de 0.8057742782152231, lo cual daría a entender que este algoritmo es uno que se ejecuta muy bien, predice de manera eficiente y acorde a lo esperable. Sin embargo cuando se realizó una submission utilizando el mismo, se obtuvo, lo que es hasta el día de hoy, el resultado más bajo en accuraccy [7].

8.9.2. Deep learning

La idea de deep learning es tener en la *Hidden layer* múltiples capas densas de neuronas. Si bien el teorema de aproximación universal garantiza que con una sola capa oculta con un número suficiente de neuronas se puede representar cualquier función, es decir que podría separar cualquier set de datos, al tener más de una capa oculta se ofrece una mayor abstracción de los datos a la siguiente capa y a la vez, complejiza el modelo. Por lo tanto, para datos con mayor grado de libertad un modelo con múltiples capas puede dar mejores resultados.

Para el trabajo se utilizó la librería de código abierto *Tensorflow*, particularmente la herramienta *Keras*, que es una API simple y flexible especialmente pensada para modelos de redes neuronales. Para poder usarlo, primero se debe definir el modelo y para esto se deben configurar los siguientes hiperparámetros:

- **Optimizador:** Es la función interna que usará para llegar a la solución óptima. El ejemplo más común es el descenso del gradiente, pero existen otras como *RMSprop*, *Adam*, *Adadelta*, *Adamax*, etc.
- **Error:** Es la función interna que usará para calcular el error de predicción. El ejemplo más común es el error cuadrático medio, nuevamente la API de keras incluye otros, pero en este caso particular como se quiere calcular si un label es 1 o 0, se consideró más adecuado utilizar la función *BinaryCrossentropy*.
- **Épocas:** Este hiperparámetro determina la cantidad de veces que el entrenamiento hace una pasada completa por todo el set de datos de entrenamiento.
- **Batch size:** controla la cantidad de muestras del set de entrenamiento que va a procesar antes de actualizar los parámetros internos del modelo.

[7] El accuracy que se obtuvo fue de 0.43763

- **Learning rate:** Hiperparámetro que controla la velocidad de aprendizaje del modelo: mientras más chico sea, más tardará el algoritmo en hallar los parámetros óptimos pero hay mayor garantía de llegar a los parámetros óptimos.
- **Cantidad de capas:** Cantidad de capas ocultas del modelo, mientras más capas tenga, más complejo será el modelo y más tiempo de cómputo requerirá ya que debe optimizar más parámetros y, a la vez, más riesgo de overfitting habrá.
- **Cantidad de neuronas por capa:** Cada capa puede tener una cantidad de neuronas distintas, mientras más tenga más complejo será el modelo y mayor cantidad de cómputo requerirá por los mismos motivos detallados anteriormente.
- **Función de activación de las neuronas:** cada capa puede también tener una función de activación distinta, pero las neuronas de una capa siempre tendrán la misma función de activación.
- **Dropout:** Dado que la red neuronal tiende a overfittear o memorizar los datos rápido, se suele aplicar un dropout a las capas de neuronas, que implica "apagar" un porcentaje de neuronas por un tiempo durante el entrenamiento.

No hay ninguna receta o regla general para determinar estos hiperparámetros ya que dependen mucho de la naturaleza del modelo de datos, por lo que se seleccionaron algunos valores típicos y luego usando el estimador de *KerasClassifier* se hizo un GridSearch como se de detalló en la sección 6.Si bien gridSearch daba la configuración "óptima", se observó que:

- Los resultados en los subsets de train y validation eran muy buenos pero, al hacer la predicción para el set de test, daba MUY mal.
- El hiperparámetro **epochs** era la más grande (sobreentrenamiento del modelo).
- El hiperparámetro **dropout** era el más bajo (nunca se apaga la neurona)

En vista de los ítems mencionados, se podría suponer que los valores obtenidos llevaban a overfitear el modelo. Debido a esto, se optó por usar como guía los hiperparámetros que devolvía GridSearch pero a su vez realizar pruebas manuales buscando la mejor configuración.

Luego se tomaron 2 caminos

1. Se armó un modelo para trabajar con el *train.csv* con todo el feature engineering detallado en la sección 4, es decir el tweet convertido a word2vec + los features creados. Si bien daba muy buenos resultados con el conjunto de validación, por encima de 0.83, a la hora de hacer las predicciones con el conjunto de test y hacer el submit la precisión bajaba muchísimo, al orden 0,60.
2. Se armó otro modelo para trabajar con el modelo NLP de BERT, detallado en la sección 4.3.2, y sólo se utilizó el campo *Text* del tweet, es decir la representación del vector del tweet y con este modelo se obtuvo el mejor resultado que es de 0.83665. La desventaja es que fue de los algoritmos que más tardo en entrenarse: tardó, aproximadamente, 12hs.

9. Conclusiones

Lo primero a mencionar es el impacto que el trabajo previo al entrenamiento de los modelos tiene en la precisión de las predicciones. Se pudo observar que limpiar el set de datos y generar/-seleccionar los features adecuados fue lo que más permitió mejorar los resultados obtenidos.

En cuanto a la limpieza, es necesario separar lo que es el preprocesamiento y la limpieza de texto. Cuando se habla de *preprocesamiento*, hace referencia a la limpieza de datos que se llevó a cabo en el TP1 y que también se utilizó en este trabajo. La limpieza de texto, en cambio, implica la

eliminación de stopwords, caracteres repetidos, etc tal y como se encuentra detallado en la sección 4 **Feature Engineering**. Esta última fue útil para word2vec pero no así para *BERT*, ya que este encoder realiza su propia limpieza interna para tokenizar el texto y la depuración previa propuesta elimina ciertas palabras que *BERT* utiliza para distinguir positivos de negativos. Una propuesta interesante sería limpiar los datos antes de usar *BERT* pero de una forma más leve: por ejemplo, eliminar solamente links y usuarios de Twitter.

En cuanto al tema de los features, se pudo ver que los features elaborados a partir del encoding del texto aportaron mucho más valor que los features relacionados al contenido del texto, sus características, etc. Debido a esto, la elección de un buen encoder se vuelve de vital importancia a la hora de buscar mejorar la predicción.

Lo segundo a mencionar es la optimización de hiperparámetros que permitió, en general, mejorar los modelos. Sin embargo, en el caso de deep learning se obtuvo un conjunto hiperparámetros que parecía provocar un modelo overfitteado. Si bien GridSearchCV devuelve un modelo, en la implementación hecha en este trabajo simplemente se tomaron los hiperparámetros y se usaron para entrenar el modelo. Habría que analizar si utilizando directamente el modelo que devuelve se puede ver una mejora a la hora de predecir.

Otro aspecto a considerar es la importancia de los ensambles. Si bien combinar distintos algoritmos requiere más costo computacional, y en ciertas ocasiones es tan alto que no resulta conveniente, también permite mejorar la exactitud de la predicción utilizando modelos que por sí solos no arrojan buenos resultados. Tal fue el caso del árbol de decisión, el cual ofrecía una exactitud de aproximadamente 0.68 y utilizando Random Forest, saltó a 0.80.

Finalmente, el modelo que brindó una mejor precisión fue el de **Deep Learning** utilizando *BERT* como NLP, con el cual se obtuvo el mejor submit de 0.83665. Sin embargo, se debe hacer una observación no menor que es que el tiempo de entrenamiento de este modelo fue el más alto de todos, tardando en una computadora personal promedio aproximadamente 13 horas. En contraste, el segundo modelo que mejor resultado dio en el submit es *LightGBM* con un total de 0.81758 y el tiempo de entrenamiento fue cuestión de minutos. Por lo que surge el balance precisión vs tiempo, ya que la diferencia entre los resultados es menor del 2 % y, dependiendo de la aplicación que se le quiera dar, quizá sea preferible una herramienta rápida pero con un score un poco menor o viceversa si el tiempo no importa tanto y se prioriza una predicción más exacta.

10. Bibliografía

- Organización de Datos - Apunte del curso - L. Argerich, N N. Golmar, D. Martinelli, M. Ramos Mejía y J. A. Laura
- https://towardsdatascience.com/better-heatmaps-and-correlation-matrix-plots-in-python-41445d0f2bec
- https://heartbeat.fritz.ai/hands-on-with-feature-selection-techniques-an-introduction-1d8dc6d86c16
- https://heartbeat.fritz.ai/hands-on-with-feature-selection-techniques-filter-methods-f248e0436ce5
- https://www.datacamp.com/community/tutorials/feature-selection-python?utm_source=adwords_ppc&utm_campaignid=10267161064&utm_adgroupid=102842301792&utm_device=c&utm_keyword=&utm_matchtype=b&utm_network=g&utm_adpostion=&utm_creative=332602034364&utm_targetid=dsa-429603003980&utm_loc_interest_ms=&utm_loc_physical_ms=1000073&gclid=EAIaIQobChMI4biwkIDx6gIVkoaRCh0Ecwh4EAAYASAAEgJzV_D_BwE
- https://towardsdatascience.com/a-feature-selection-tool-for-machine-learning-in-python-b64dd23710f0
- https://www.oreilly.com/library/view/evaluating-machine-learning/9781492048756/ch04.html

- https://www.geeksforgeeks.org/passive-aggressive-classifiers/
- https://kavita-ganesan.com/tfidftransformer-tfidfvectorizer-usage-differences/#.XyiRhXUzY5k
- https://www.aprendemachinelearning.com/regresion-logistica-con-python-paso-a-paso/
- https://scikit-learn.org/stable/modules/tree.html
- https://scikit-learn.org/stable/modules/naive_bayes.html
- The optimality of Naive Bayes - H. Zhang (2004).
- https://www.aprendemachinelearning.com/clasificar-con-k-nearest-neighbor-ejemplo-en-python/
- https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessCl html
- https://medium.com/datos-y-ciencia/machine-learning-supervisado-fundamentos-de-la-regresion-lineal-bbcb07fe7fd
- https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47
- https://towardsdatascience.com/an-intuitive-explanation-of-random-forest-and-extra-trees-classifiers-8507ac21d54b
- https://scikit-learn.org/stable/modules/neural_networks_supervised.html
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier. html
- https://relopezbriega.github.io/blog/2017/06/10/boosting-en-machine-learning-con-python/
- https://medium.com/@jcrispis56/una-introducci%C3%B3n-completa-a-redes-neuronales-con-python-y-tensorflow-2-0-b7f20bcfebc5
- https://www.tensorflow.org/tutorials/keras/
- https://keras.io/examples/nlp/text_classification_from_scratch/
- https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270
- https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/bert-encoder
- https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04
- Paper *Attention Is All You Need* https://arxiv.org/pdf/1706.03762.pdf
- Paper *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* https://arxiv.org/pdf/1810.04805.pdf
- https://towardsdatascience.com/understanding-bert-bidirectional-encoder-representations-from-transformers-45ee6cd51eef
- https://www.kaggle.com/prashant111/lightgbm-classifier-in-python

- https://tfhub.dev/tensorflow/bert_en_uncased_L-24_H-1024_A-16/1
- Python Natural Language Processing: Advanced machine learning and deep learning techniques for natural language processing - J. Thanaki
- https://developers.google.com/machine-learning/crash-course?hl=es