UBA - Facultad de Ingeniería

Departamento de Computación

Organización de Datos (75.06)

Trabajo Práctico 2

1er cuatrimestre - 2020

GRUPO: **El Cuarteto Imperial**		
Alumno	**Padrón**	**Mail**
LARREA BUENDÍA, Hugo Marcelo	102140	hlarrea@fi.uba.arr
MARTINEZ SASTRE, Gonzalo Gabriel	102321	gmartinezs@fi.uba.ar
RIEDEL, Nicolás Agustín	102130	nriedel@fi.uba.ar
ZBOGAR, Ezequiel	102216	ezbogar@fi.uba.ar

Índice

1. Resumen

El objetivo del presente trabajo práctico es elaborar un modelo de *machine learning* capaz de predecir si un *Tweet* trata sobre un desastre/tragedia real. Como particularidad, gran parte del dataset que se utiliza tanto para el train como para el test están de alguna forma relacionadas a la temática sobre la que se tiene que trabajar. Las predicciones realizadas para el set de test fueron subidas a la página *Kaggle*, más específicamente a la competencia "Real or Not? NLP with Disaster Tweet"

Durante la realización del trabajo se probaron distintos algoritmos estudiados en la materia (*XGBoost*, *Random Forest*, redes neuronales, entre otros) con distintas *features* que se extrajeron del set de datos. Se fueron descartando aquellos que mostraban ser poco efectivos y se concentraron los esfuerzos en perfeccionar aquellos con mejores resultados. Asimismo, a lo largo del trabajo se pensaron qué nuevos *features* se podrían extraer del set, estos fueron utilizados para llevar a cabo la clasificación y, al igual que con los algoritmos, se conservaron aquellos que exhibían resultados positivos, dejando de lado al resto.

El modelo que ha mostrado mejores resultados al momento de la presentación de este trabajo es un ensamble de varios modelos basados en redes neuronales con distintos *features*. El **F1 score** obtenido con el mismo en la competencia de *Kaggle* es de 0.84339.

2. Introducción

Twitter es uno de los principales canales de comunicación de los últimos años. Su principal fortaleza reside en el carácter real-time del mismo lo cual, sumado a que un importante porcentaje de la población mundial posee acceso a smartphones u otros dispositivos con acceso a la web, permite que personas en cualquier parte del mundo se puedan comunicar al instante. Esto resulta ser de gran utilidad para una diversidad de áreas, donde analizando la información provista por dicha red social se puede obtener un indicador en tiempo real de los sucesos mundiales.

En el caso de este trabajo se busca generar un modelo predictor, que recibiendo un *Tweet* pueda clasificarlo en dos categorías: si realmente se encuentra asociado a un desastre real o si no lo hace. Esto, en un hipotético caso, podría ser de gran ayuda para servicios de emergencia, donde el analizar *Tweets* en tiempo real permitiría tener conocimiento de una catástrofe mucho más rápido que mediante medios convencionales.

Para concretar dicho objetivo se utilizaron y combinaron diversos algoritmos de *machine learning* vistos en clase, sumados a los conocimientos adquiridos previamente o durante el desarrollo de este trabajo.

El lenguaje de programación utilizado fue *Python*. Adicionalmente, se utilizaron las siguientes herramientas: *Git* como programa de control de versiones, *Jupyter* como entorno de desarrollo y las siguientes bibliotecas:

▪ Pandas ▪ Numpy ▪ Matplotlib ▪ XGBoost ▪ sklearn ▪ LightGBM ▪ CatBoost ▪ nltk ▪ wordcloud ▪ keras ▪ pickle ▪ spacy ▪ re ▪ gensim ▪ TensorFlow ▪ entre otras...

3. Procesamiento de datos

A continuación se detallan los *features* que se extrajeron del set de datos, hayan sido usados para el modelo final o no. Principalmente se trabajo con el texto del *Tweet* debido a que en muchos modelos al generar *features* de las keywords y la localización, sus resultados no se vieron mejorados. Por razones obvias no se uso el ID del *Tweet* como *feature*.

3.1. Análisis del texto

En un principio, con el fin lograr un mayor aprovechamiento de los *features* que se pueden extraer del texto, se procedió a filtrar el mismo en base a dos conjunto determinados de criterios. Los mismos se describen a continuación.

En primer lugar, se optó por un filtrado básico mediante el cual se removieron las *stopwords* (artículos, preposiciones, pronombres, etc.), todos los símbolos y caracteres numéricos, y se convirtieron todas las letras a minúsculas.

Por otro lado, en algunas implementaciones, se intentó profundizar dicho filtrado añadiéndole al mismo la expansión de abreviaturas de la lengua inglesa, tales como *thx* ('thanks', gracias), *lol* ('laughing out loud', riendo a carcajadas), *b4* ('before', antes), entre otras. Además se llevó a cabo la lematización de las palabras, es decir, se redujeron todas las palabras a su lema, aquel término que se toma como representante de todas las derivaciones posibles de una palabra.

3.2. NLP

NLP, del inglés **Natural Language Processing**, es una rama de la computación que estudia la relación entre las computadoras y el lenguaje humano. Dentro de NLP se hará mayor énfasis en una herramienta particular, los *Word Embeddings*.

La idea detrás de los Word Embeddings es representar palabras o frases del lenguaje

natural como vectores de números reales (proceso conocido como vectorización). Este es el pilar del presente trabajo, el concepto más importante dada la naturaleza del problema en cuestión.

A continuación se detallan los tipos de *Embeddigs* utilizados en el presente trabajo.

3.3. *Word2Vec*

Es un método de *encoding* que vectoriza palabras buscando asociaciones entre ellas mediante un modelo de red neuronal. Para este trabajo se uso un modelo pre-entrenado, esto quiere decir, con los *Embeddings* ya realizados. Las palabras de los *Tweets* fueron buscadas a modo de "diccionario" donde la clave fue la palabra a usar y, el valor, el resultado del *Embedding*. Este es un tipo de enconding no contextualizado.

3.3.1. *GloVe*

El método de uso es muy similar al de *Word2Vec*, también se decidió utilizar un modelo pre-entrenado.

3.3.2. *ELMo*

ELMo es un Word *Embedding* fuertemente contextualizado, fue considerado como el estado del arte en este tipo de modelos, para luego ser reemplazado por BERT. A diferencia de modelos mas simples como *Word2Vec* que se limitan a la palabra, *ELMo* crea los *Embeddings* teniendo en cuenta el contexto en el que fue usado la misma. Como consecuencia, una palabra puede tener distintos *embeddings* de acuerdo al contexto en que aparece.

3.3.3. *BERT*

Al igual que *ELMo* es fuertemente contextualizado, pero con una pequeña pero importante diferencia. *BERT* es "bidireccional", esto quiere decir que el contexto es analizado

de izquierda a derecha y de derecha a izquierda. Lo que obviamente genera un resultado más preciso en términos de contexto.

3.3.4. *TF-IDF* y *BOW*

Como primer acercamiento, se pensó en realizar un *TF-IDF* definiendo un vocabulario que contenga tan sólo palabras relacionadas con desastres. Con el mismo vocabulario también se probó usando *BOW* en lugar de *TF-IDF*. Los resultados obtenidos con ambos acercamientos fueron bastante pobres. En esta primer instancia, se trabajó únicamente con unigramas.

El siguiente acercamiento fue hacer lo mismo pero sin definir el vocabulario. Con esto se obtuvieron resultados más prometedores por lo cual se prosiguió a hacer lo mismo pero ahora no solo con uni-gramas sino también con bi-gramas y tri-gramas.

Para llevar esto a cabo se utilizó *tfidfvectorizer* de sklearn, también se observo una mejora al seleccionar un porcentaje de los *features* mediante *SelectPercentile* de la misma biblioteca. Se probaron distintos porcentajes y el mejor desempeño que se encontró fue con el 30 % de los *features*. También se probó filtrar los caracteres numéricos del texto, pero se observo que los modelos funcionaban mejor al dejar estos últimos.

4. Modelos de *machine learning*

En esta sección se repasarán brevemente los modelos que fueron probados por el equipo a lo largo de la realización del trabajo, estos mismos serán profundizados en la próxima sección.

Como paso previo, se explicarán rápidamente las métricas empleadas.

4.1. Métricas utilizadas

Como punto de partida para evaluar el desempeño de cada modelo se decidió utilizar cuatro métricas distintas de gran difusión y uso. Las mismas son exactitud, precisión, exhaustividad y puntaje F1.

Sin embargo, previamente, para facilitar el entendimiento de las mismas se mencionarán ciertos conceptos previos. Siendo un caso positivo cuando un *Tweet* se trata sobre un desastre y negativo en caso contrario, se definen:

- **Verdadero positivo (VP):** el *Tweet* es sobre un desastre y el modelo lo predice correctamente.
- **Verdadero negativo (VN):** el *Tweet* no es sobre un desastre y el modelo lo predice correctamente.
- **Falso positivo (FP):** el *Tweet* no es sobre un desastre, empero el modelo lo predice como tal.
- **Falso negativo (FN):** el *Tweet* es sobre un desastre, pero la predicción es incorrecta.

Una vez aclarado esto, se procede a explicar dichas métricas.

4.1.1. Exactitud (*Accuracy*)

Mide el porcentaje de casos que el modelo predijo correctamente. Si bien es una métrica muy utilizada, puede llegar a ser engañosa cuando se trata de clases desbalanceadas, por lo cual (si bien no ocurre eso en el caso puntual del trabajo) se decidió acompañar esta métrica con las próximas tres.

La exactitud se puede calcular como:

$$accuracy = \frac{VP + VN}{VP + VN + FP + FN} \tag{1}$$

4.1.2. Precisión (*Precision*)

Esta métrica se utiliza para conocer la calidad de la predicción, puntualmente en este caso, informa el porcentaje de los *Tweets* clasificados por el modelo como relacionados a un desastre que realmente lo están. Su fórmula es la siguiente:

$$precision = \frac{VP}{VP + FP} \tag{2}$$

4.1.3. Exhaustividad (*Recall*)

En este caso el *recall* indica el porcentaje de *Tweets* sobre desastres que el modelo predice correctamente. Se calcula de la siguiente manera:

$$recall = \frac{VP}{VP + FN} \tag{3}$$

4.1.4. Puntaje F1 (*F1 Score*)

Finalmente, debido a que en este caso se consideró que la importancia tanto de la precisión como del *recall* es la misma, se eligió el puntaje F1 para incluir ambos valores en uno único, el cual se obtiene como se muestra a continuación.

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{4}$$

4.2. Modelos basados en Árboles de Decisión

En este caso, no se escogió la utilización de árboles de decisión singulares sino que se hizo uso de cinco algoritmos distintos de ensamble de este tipo de árboles. Los mismos se describen en esta sección.

Es necesario mencionar que si bien en un comienzo se utilizaron las variantes clasificadoras de cada algoritmo, con el progreso del trabajo se optó por los regresores puesto que los valores obtenidos por estos podían ser de mayor provecho para modelos posteriores.

Los valores obtenidos por dichos regresores se puede interpretar como la probabilidad de que un tweet trate sobre un desastre, es decir, que el target sea **uno**. Esta probabilidad es posteriormente redondeada, con lo que se predecirá **uno** si esta es mayor a 0.5 y **cero**, en cualquier otro caso.

4.2.1. *Gradient Boosting*

Se trata de un algoritmo de ensamble basado en la técnica de *boosting*, el mismo consiste en distintos estimadores secuenciales débiles en los cuales cada nuevo árbol se encarga de mejorar el resultado del anterior para así generar un estimador robusto. Dicha mejora consiste en la minimización de una función pérdida (*loss*) entre iteraciones.

En los modelos se utilizaron *GradientBoostingRegressor* y *GradientBoostingClassifier* de la biblioteca Scikit Learn.

4.2.2. *XGBoost*

XGBoost es una implementación especial de *Gradient Boosting* con la diferencia de que presenta mejoras en la velocidad de ejecución y en la performance de las predicciones.

En el caso particular de este trabajo, éste representa el algoritmo de esta índole con el mejor desempeño a la hora de realizar las predicciones.

4.2.3. *LightGBM*

LightGBM al igual que *XGBoost* también es una implementación de *Gradient Boosting*. Su principal diferencia con el anterior reside en que sus árboles tienden a crecer verticalmente. Esto genera una mejor *perfomance* cuando se trabaja con una gran cantidad de datos pero asimismo tiende a hacer *overfitting* con sets de datos pequeños.

4.2.4. *CatBoost*

CatBoost es un algoritmo que utiliza *Gradient Boosting* sobre árboles de decisiones. Como principal diferencia a los otros algoritmos, este permite atributos no numéricos.

4.2.5. *Random Forest*

Random Forest es un ensamble de árboles de decisión que usa *bagging*, esto quiere decir que no todos los arboles ven la totalidad de los datos. Esto genera que cada árbol se entrene con un set de datos distinto, lo que al momento de combinar los resultados, hace que los errores se compensen.

4.3. KNN

Es uno de los algoritmos más simples de *machine learning*, el cual consiste en clasificar un determinado punto consulta en base a la clase a la que pertenecen los K vecinos más cercanos. En este modelo el único hiper-parámetro es K.

Unas de sus principales desventajas es que se vuelve inviable cuando se trabaja con grandes cantidades de datos, debido a la complejidad del algoritmo.

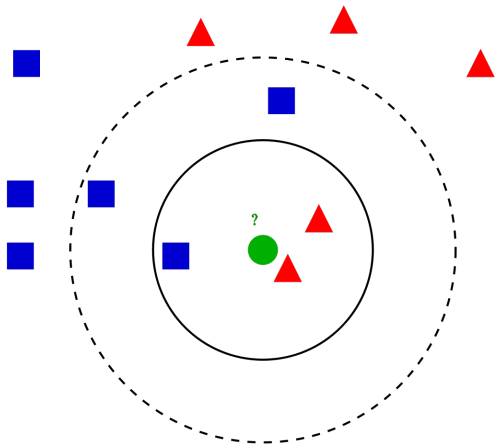

Figura 1: En este caso, si K=3, el resultado de la consulta es triángulo

Si bien es un algoritmo que puede dar buenos resultados, ese no fue el caso de este trabajo.

4.4. Naïve Bayes

Este algoritmo está basado en el teorema de Bayes, es sencillo, rápido y escala muy bien en *datasets* de gran tamaño pero casi nunca es "el mejor" algoritmo y por esa razón se decidió no implementarlo. Este algoritmo funciona bajo la hipótesis de independencia entre las variables predictoras, y es debido a eso que recibe el apelativo de naïve, es decir, ingenuo.

4.5. Redes Neuronales

Los modelos basados en redes neuronales son con diferencia los que mejores resultados proporcionaron. La mayoría son modelos basados en el algoritmo de perceptrón multicapa. Esta se aplico a todos los diferentes *Embeddings* anteriormente mencionados, a *TF-IDF* (con unigramas; unigramas y bigramas; unigramas, bigramas y trigramas). En todos los

casos mencionados se probaron diferentes hiper-parámetros, los cuales se fueron ajustando para optimizar las predicciones hechas con cada modelo.

Además, se utilizó una técnica desarrollada en 2018 por Google llamada *BERT* (Bidirectional Encoder Representations from Transformers) la cual es actualmente considerada por muchos como el "estado del arte" en NLP. La misma consiste en aplicar *fine-tunning*, esto es, adaptar a nuestro problema una red neuronal que fue previamente entrenada, en el caso de *BERT*, con técnicas de aprendizaje semi-supervisado.

Una de las ventajas de las representaciones (*Embeddings*) pre-entrenadas de *BERT* es que son contextualizadas, esto es, genera representaciones de cada palabra basándose en las otras que conforman la oración. A diferencia de modelos *context-free* como *Word2Vec* o *GloVe* que generan un *Embedding* por palabra y por lo tanto no se tiene en cuenta el contexto en el que se encuentran, lo cual en muchos casos es crucial para generar la respuesta correcta a nuestro problema.

Este último modelo es el que particularmente produjo los mejores resultados. No obstante y para nuestra sorpresa, el resultado final mejora luego de aplicar ensamble con el resto de los modelos (La mayoría de ellos redes neuronales) a pesar de que los resultados individuales son considerablemente peores.

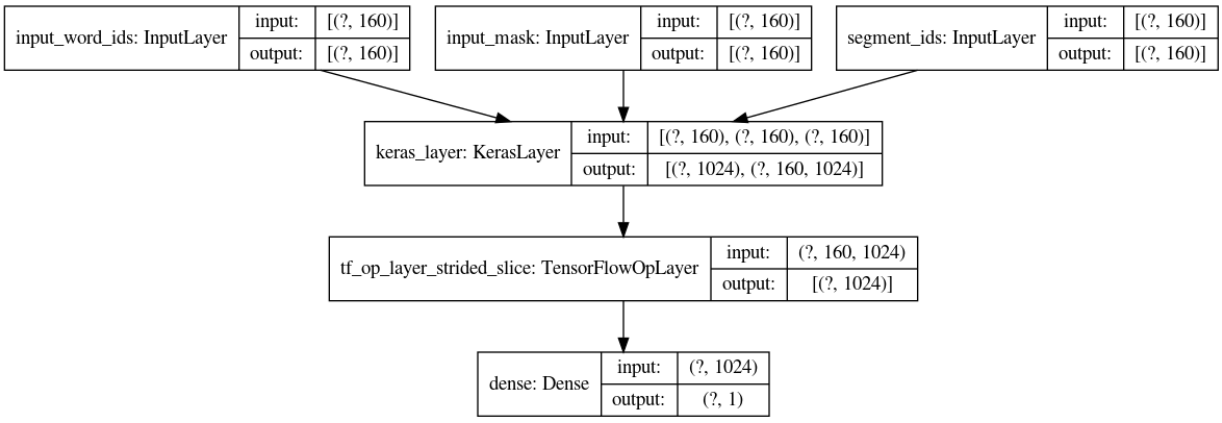

Figura 2: Diagrama de la red neuronal de BERT

4.6. Tuneo de Hiper-Parámetros

Conforme se empiezan a ejecutar varias veces cada uno de los algoritmos se hace evidente la importancia que tienen los hiper-parámetros en los resultados. Encontrar un buen conjunto de hiper-parámetros es crucial para que el modelo alcance su rendimiento óptimo, dado que la diferencia es sustancial entre un buen conjunto de hiper-parámetros y uno malo. Si bien no es tarea fácil hallar los mejores hiper-parámetros para cada algoritmo, se llevaron a cabo un par de ideas vistas en clase que dieron buenos resultados:

4.6.1. Grid Search

Grid Search busca cual es el conjunto de Hiper-Parámetros óptimos del modelo, entrenando el mismo para las distintas combinaciones los posibles de un cierto rango de valores predefinidos. La principal desventaja de Grid Search es que prueba **todas** las combinaciones de hiper-parámetros, lo cual puede generar un tiempo de ejecución bastante alto.

4.6.2. Random Search

Random Search busca los Hiper-Parámetros que optimicen el modelo de manera aleatoria entre un rango de valores para cada Hiper-parámetro. La principal ventaja que presenta este método es que, como se definen la cantidad iteraciones (combinaciones de Hiper-Parámetros que se van a probar) de antemano, el tiempo que tarda es n veces el tiempo que tarda el modelo en entrenarse (siendo n la cantidad de iteraciones del Random Search).

Debido a esto, es común pasarle amplios rangos de posibles valores para cada Hiper-Parámetro, pues el tiempo que tarda es constante (si el tiempo de entrenamiento del modelo no depende de alguno/s de los Hiper-Parámetros) o, en el peor de los casos está acotado (si el tiempo de entrenamiento del modelo depende de alguno/s de los Hiper-Parámetros como los Epochs de las redes que fueron usado). Esto conlleva a encontrar valores para los Hiper-Parámetros que, difícilmente, se probarían en un Grid Search.

La principal desventaja es que al no probar absolutamente todas las combinaciones posibles de Hiper-Parámetros como en Grid Search, puede que se llegue a un sub-óptimo entre el conjunto de valores posibles.

4.7. Ensambles

Los mejores algoritmos de ML suelen surgir de la combinación de varios algoritmos. Es muy raro que un solo algoritmo de ML logre mejores resultados que un ensamble. Esto queda en plena evidencia conforme se combinaron los resultados de distintas predicciones, observando que el rendimiento del modelo aumentaba considerablemente.

Particularmente, el mejor resultado obtenido por el equipo a la fecha de presentación del trabajo es resultado de un ensamble de las predicciones de los métodos que dieron mejores resultados.

4.7.1. Boosting

Este tipo de ensamble fue usado en los modelos basados en *XGBoost*, *LightGBM*, *CatBoost* y GradientBoosting que, como ya se mencionó anteriormente, no produjeron resultados tan buenos como otros algoritmos probados.

4.7.2. Majority Voting

Este método simplemente se basa en ver cual es la clase mayoritaria entre los resultados de cada uno de los clasificadores. Si bien se obtuvo una mejora en el resultado cuando se puso en práctica, no es el tipo de ensamble que llevó a cabo los mejores resultados. Esto último puede ser posible debido a que el resultado del *Majority Voting* mejora notablemente si se usan resultados que tengan poca correlación entre sí, lo cual a primera vista no parecería ser este caso ya que los clasificadores tiene como input los resultados de aplicar los *Embeddings*.

Este modelo suele mejorar notablemente el resultado de cada modelo individual cuando los resultados de estos tienen poca correlación entre sí. Debido a esto y a los pobres resultados obtenidos al ponerlo en practica se puede decir que es probable que los resultados de los modelos usados tengan una fuerte correlación.

4.7.3. Averaging

Este es sin lugar a dudas el tipo de ensamble que mejores resultados produjo. El mismo se basa en promediar los resultados de varios clasificadores y, en base a eso, obtener el resultado final de la consulta. El hecho de promediar clasificadores produce una separación más suave entre las clases que probablemente un solo clasificador lleve a cabo irregularmente, y esto conduce a un aumento en el rendimiento del algoritmo.

5. Cronología de los modelos

5.1. Primer modelo

En una primer instancia se planteó un modelo bastante ambicioso para el cual se buscaron, en base al análisis exploratorio del primer trabajo, todas aquellas características del set de datos que marcaran alguna tendencia sobre el carácter de veracidad de un *Tweet*. Una vez seleccionadas, se procedió a elaborar los *features* correspondientes. Los mismos contemplaban:

- Palabras con mayor y menor porcentaje de veracidad.
- Pares de 2 palabras con mayor y menor porcentaje de veracidad.
- Aquellos *Tweets* que contienen @, ¿?, ¡! tienden a ser falsos.
- Keywords con mayor y menor porcentaje de veracidad.
- Locaciones más y menos veraces.
- Longitud del *Tweet*, tanto en palabras como en caracteres.

Una vez obtenido el dataframe deseado se utilizaron algoritmos basados en árboles para entrenar el modelo y así, realizar las predicciones. Se hizo uso de *RandomForest*, *GradientBoosting*, *LightGBM*, *CatBoost* y *XGBoost*, siendo este último el que mejores resultados obtenía con el set de entrenamiento. Posteriormente, se obtuvo un desempeño similar cuando se realizó el *submit* en *Kaggle* (0.78823).

Algunos de los árboles generados por *XGBoost* se muestran a continuación.

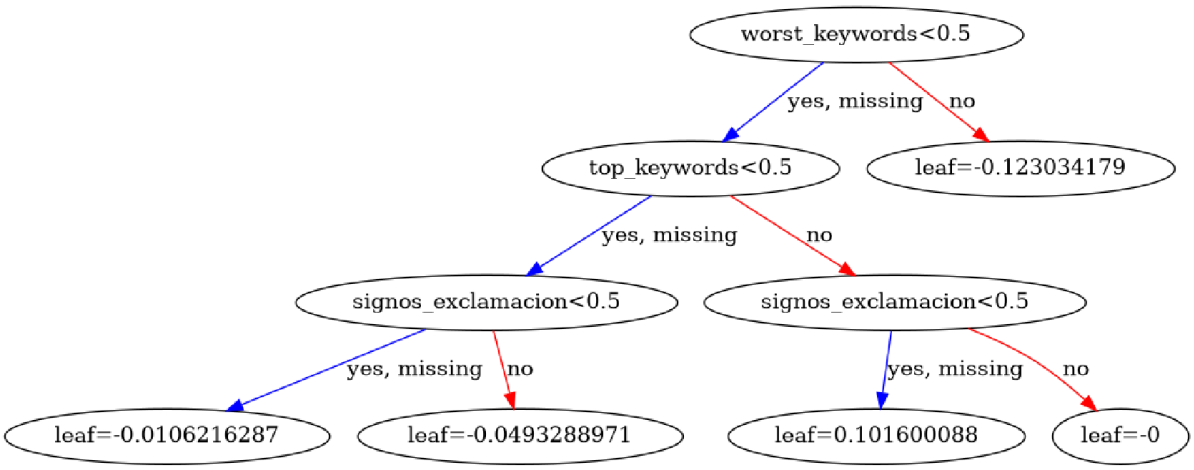

Figura 3: Árbol generado por *XGBoost*

Aquí *worst_keywords*, *top_keywords* y *signos_exclamacion* son features binarios que indican si el tweet tiene un keyword con un grado de veracidad muy bajo, un grado muy alto y si presenta signos de exclamación, respectivamente.

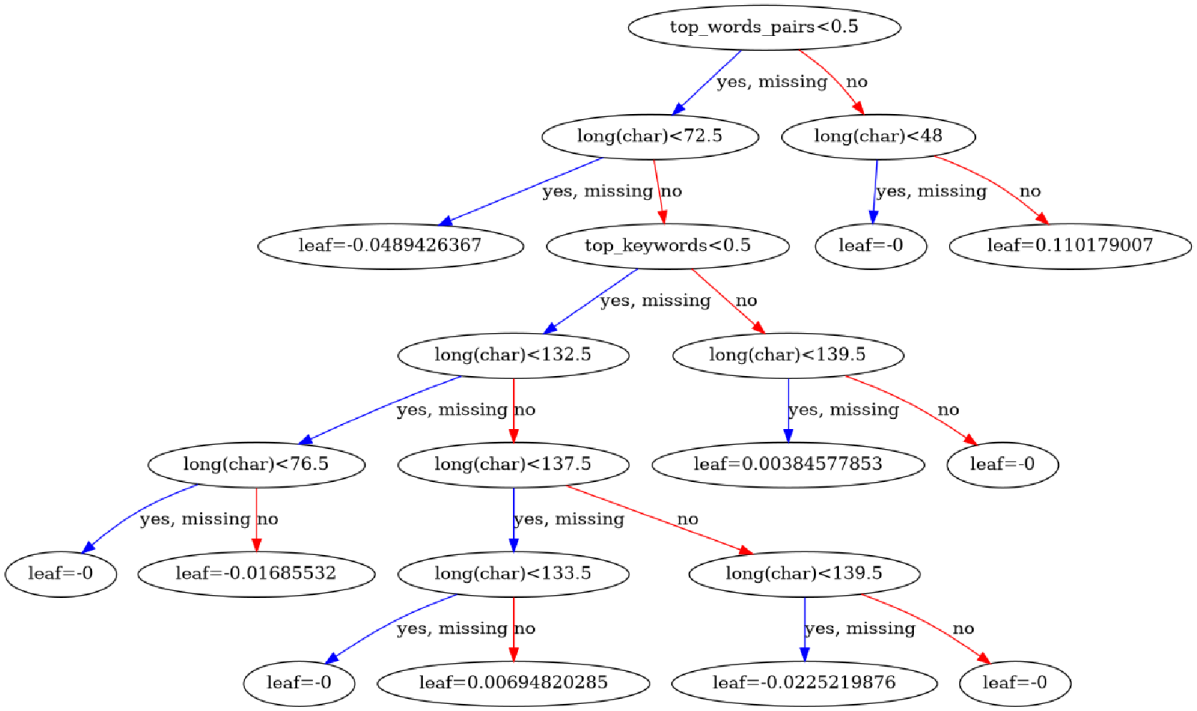

Figura 4: Árbol generado por *XGBoost*

Este árbol considera además el atributo *long(char)* el cual indica la longitud en caracteres del tweet.

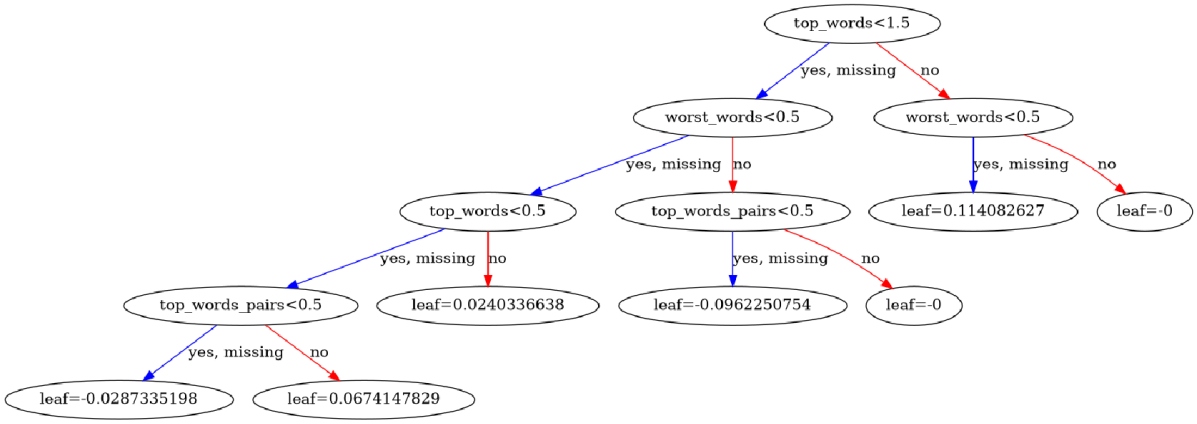

Figura 5: Árbol generado por *XGBoost*

Por último, este árbol pone en consideración *top_words_pairs*, un feature que indica la cantidad de pares de palabras con mayor grado de veracidad que contiene el *Tweet*.

Durante los primeros días, el equipo se centró en buscar mejorar principalmente el modelo de *XGBoost* modificando sus hiper-parámetros. No se tardó mucho tiempo en llegar a uno de los problemas comunes que hay al momento de encontrarse con trabajos de esta índole, el tan temido *overfitting*. Este ocurre cuando el modelo es demasiado complejo para el problema que se busca resolver y, debido a esto, memoriza el set de entrenamiento. Esto tiene la terrible consecuencia de que el modelo termina generalizando mal.

Esta problemática a la que se llegó evidenció la necesidad de replantear el modelo, extraer nuevas *features* de los datos y, en general, traer nuevas ideas a la mesa con el objetivo de mejorar el puntaje.

5.2. Introduciendo los *embeddings*

Con el objeto de mejorar los resultados obtenidos del modelo anterior se buscó la forma de explotar de mejor manera la información brindada por el texto de los *Tweets*, llegando

así a lo que se conoce como NLP. Tras varios días de indagación y algunas pruebas, el equipo decidió utilizar dos algoritmos de *Embedding* de palabras, *Word2Vec* y *GloVe*, y dos algoritmos de *Embedding* contextualizado, *ELMo* y *BERT*.

De esta manera, una vez limpiado el texto (como se aclaró en la primer parte del apartado 3.1), el tratamiento brindado con los dos primeros algoritmos consistía en obtener un *Embedding* por cada *Tweet* el cual representaba el promedio de los *Embeddings* de cada una de palabras que componía al *Tweet*. Cabe destacar que en un primer momento no se generaba el promedio sino que simplemente se realizaba la suma de los valores obtenidos por cada palabra, pero al no obtener buenos resultados esta opción fue desestimada.

Por otro lado, *ELMo* y *BERT*, al tener la funcionalidad de generar un *Embedding* a partir de oraciones completas, simplemente recibían el *Tweet* entero.

Una vez obtenido un *Embedding* por cada *Tweet*, se procedió a enviar los resultados de cada algoritmo a los cinco estimadores basados en árboles mencionados en el apartado 4.2. Posteriormente, se tomaron las predicciones de los cinco algoritmos y se les aplicó un *majority voting*.

Si bien se podía observar un claro *overfitting* con estos resultados, pues las métricas evaluadas en el set de entrenamiento devolvían valores mayores al 0.9 mientras que en *Kaggle* el puntaje de las predicciones del set de testing no conseguían superar la línea de 0.8, se optó por guardarlos pues existía la posibilidad de que todavía se podría sacar provecho de estos modelos. Más adelante se hablará de que efectivamente así fue.

Por último, se debe mencionar un detalle no menor que en la teoría parecía muy prometedor, sin embargo, en la práctica se pudo observar que no fue así. Se trata de la lematización de las palabras y la expansión de las abreviaturas de la lengua inglesa. Al volver a realizar los *Embeddings* de los *Tweets* sometidos a dicho proceso de limpieza y aplicarles los mismos algoritmos, se pudo observar que el desempeño de las predicciones no mejoraba, sino todo lo contrario, disminuía tanto en el set de entrenamiento como en el set de testing evaluado en *Kaggle*. Por esta razón, dicho proceso de filtrado de texto fue

descartado.

5.3. Dándole una oportunidad a KNN

Para pasar por todos los modelos se decidió experimentar con el más simple, KNN, cuyo funcionamiento fue explicado previamente.

Primero, lo que se probó fue usar como dimensiones del espacio a las predicciones de algoritmos previamente utilizados, esto como es de esperarse sufre un *overfitting* muy grande. Luego se intentó hacer lo mismo pero usando menos predicciones, para reducir las dimensiones y con esto el *overfitting*, no obstante, el resultado empeoró.

En consecuencia, se decidió dejar este algoritmo de lado, pudiendo concluir que no es la herramienta adecuada para este tipo de problema.

5.4. La alegría que necesitaba el grupo: Redes Neuronales

Viendo que el desempeño de los modelos utilizados hasta el momento, si bien tenían una performance aceptable, no eran los esperados, se decidió intentar utilizar redes neuronales.

El primer modelo de este estilo fue una red bastante sencilla de tres capas (*Embedding*, *LSTM* y *Dense*) la cual recibía los *embeddings* de los tweets producidos por *GloVe*, se entrenaba y producía las predicciones. Sorprendentemente los resultados de la misma fueron muy positivos y permitieron al equipo mejorar el nivel de los futuros modelos.

Una vez que quedó claro que las redes neuronales eran un campo provechoso, la gran parte de los esfuerzos del grupo se orientó en dicho sentido. Un claro ejemplo de esto, fue un modelo en el cual se tomaron los *embeddings* que habían sido procesados por los algoritmos basados en árboles con los cuales se entrenaron cuatro redes neuronales distintas (una por cada embedding) y luego se hizo un *majority voting* de las predicciones de cada una. Como resultado se obtuvo un puntaje de 0.81489 en Kaggle.

A continuación se muestran los diseños más óptimos encontrados de las redes de dicho

modelo junto con la progresión del entrenamiento de cada una. Es necesario mencionar que los valores recibidos en la capa *Input* corresponden a la dimensionalidad de cada tipo de embedding y que los *epochs* se pueden entender como la cantidad de iteraciones de la red.

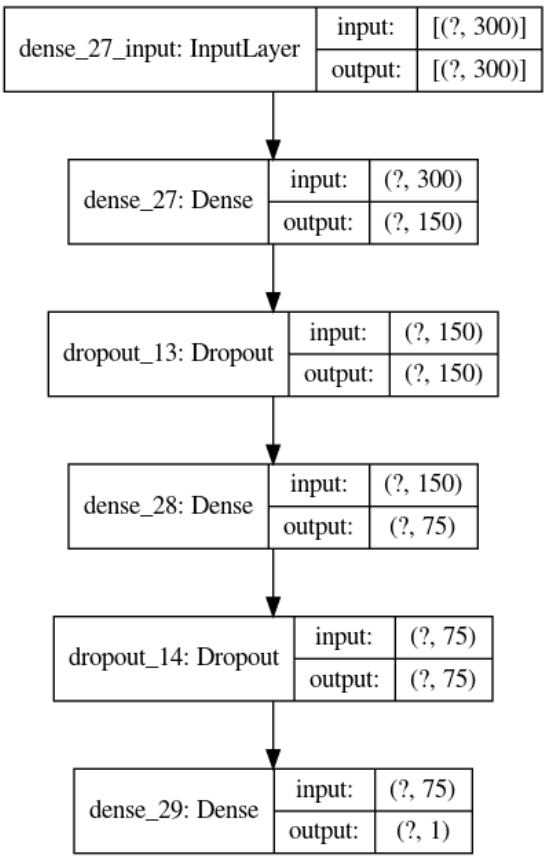

Figura 6: Modelo de red para *Word2Vec*

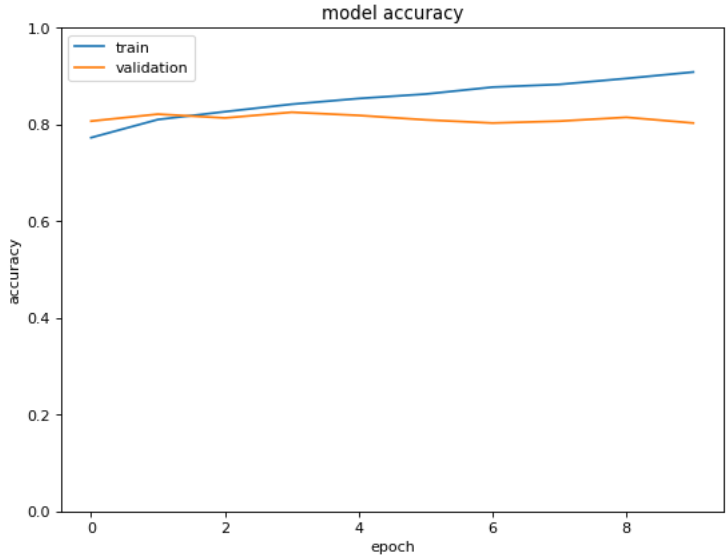

Figura 7: Evolución del entrenamiento de red para *Word2Vec*

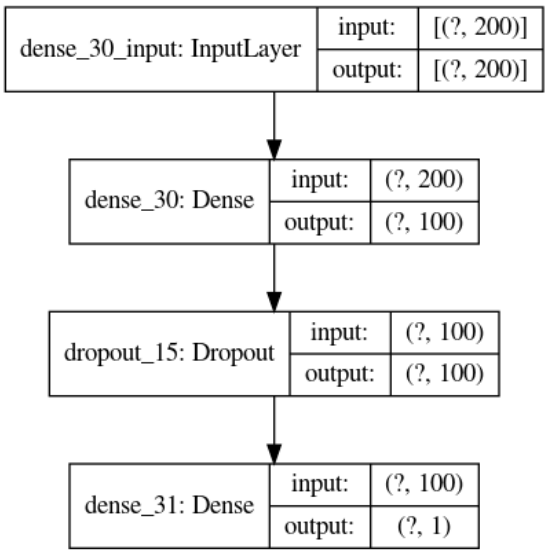

Figura 8: Modelo de red para *GloVe*

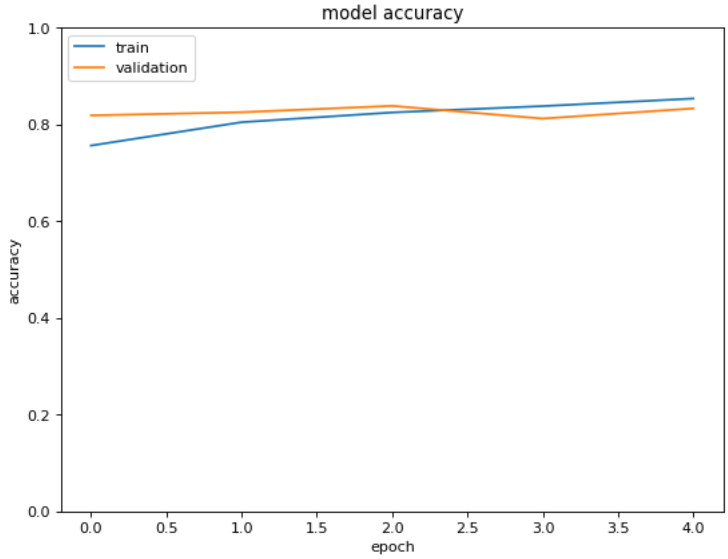

Figura 9: Evolución del entrenamiento de red para *GloVe*

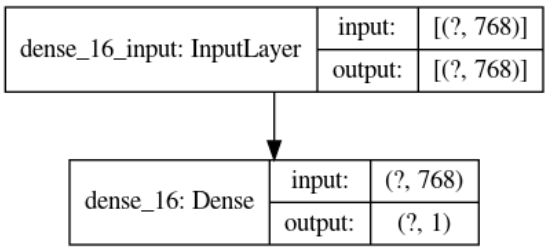

Figura 10: Modelo de red para *BERT*

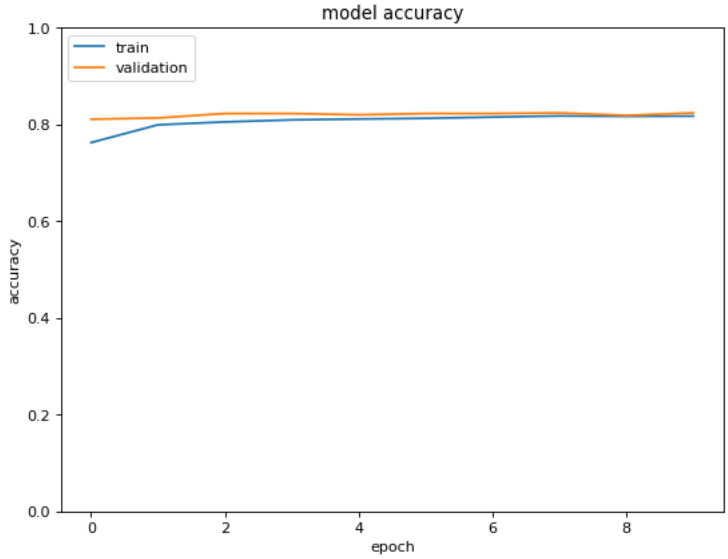

Figura 11: Evolución del entrenamiento de red para *BERT*

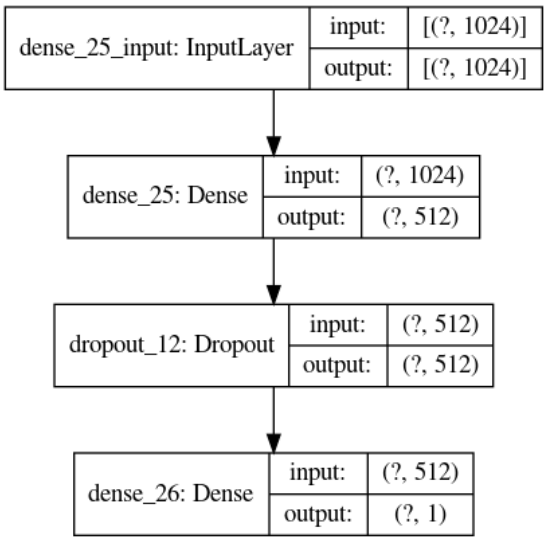

Figura 12: Modelo de red para *ELMo*

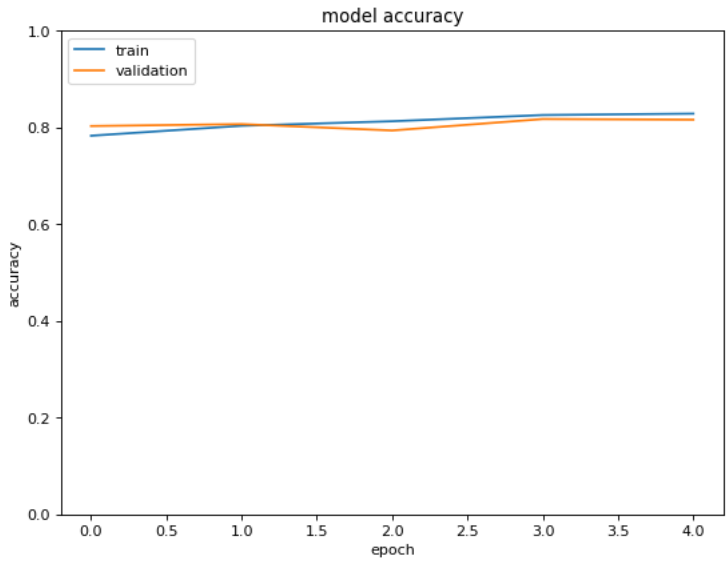

Figura 13: Evolución del entrenamiento de red para *ELMo*

Si bien se trataba de un buen desempeño, todavía había features sin explorar y el método de ensamble se podía mejorar, lo cual ocurrió posteriormente cuando se hizo uso del Averaging, cuestión que será explicada en otra sección.

5.5. Indagando con *TF-IDF*

El séptimo modelo que se uso consistió en entrenar a una red neuronal para que aprenda a predecir el target con *features* de *TF-IDF* sacadas del texto. Entre las primeras variaciones del modelo se probó: usar *TF-IDF* solo teniendo en cuenta términos relacionados a tragedias y desastres (para lo cual se utilizó un documento encontrado en la web), se probó hacer lo mismo pero con *BOW* (la diferencia es que los valores son 1 o 0).

Finalmente, de las pruebas iniciales lo que mejores resultados trajo fue usar *TF-IDF* sin un vocabulario definido. Durante toda esta etapa se usaron solamente unigramas (más tarde se prosiguió a probar con bigramas y trigramas). Al agregar las predicciones del modelo al Averaging de las redes con los *Embeddings* se mejoró el resultado a 0.81887.

Luego, se prosiguió a hacer nuevas redes probando su rendimiento sólo con bigramas y

sólo con trigramas (no se pensó usar n-gramas de mayor n debido a la poca longitud que suelen tener los *Tweets*). Lastimosamente los resultados obtenidos con estas variaciones no se mostraban demasiado prometedoras por el *Accuracy* obtenido tanto con los sets de entrenamiento como con los sets de validación. Debido a esto, se prosiguió a probar tres variantes:

- Un modelo que solo tenga en cuenta unigramas
- Un modelo que solo tenga en cuenta unigramas y bigramas
- Un modelo que tenga en cuenta unigramas, bigrams y trigramas

Además, se probó seleccionando *features* con SelectPercentile, se encontró que usando el 30 % de los *features* en cada una de las variantes se obtenían los mejores resultados entre los porcentajes probados. Al agregar estar variantes al Averaging se observó que el score del equipo subió a 0.82745 (cabe aclarar que cuando se agregaron estos nuevos modelos al Averaging, las redes con los *Embeddings* tuvieron ciertas optimizaciones y si bien agregarlos mejoró el score, el impacto real de estos no debe ser comparado directamente con el score obtenido con el primer acercamiento al uso de estos *features*).

5.6. Los *features* faltantes

Como se mencionó previamente existían *features* cuyo pontencial todavía no había sido explorado, por ende, llegado cierto momento se probó agregar a algunos de los modelos ya existentes elementos tales como la veracidad del keyword asociado al tweet o el análisis sobre si el tweet pertenece una locación válida o no.

Por un lado, los modelos en los que se uso *TF-IDF* se probó agregar *features* extraídos de las keywords con un acercamiento similar al que se tuvo con el texto del *Tweet* (aunque sólo teniendo en cuenta unigramas pues al ser solo una palabra no tenia sentido probar otras variantes). En todos los casos, agregar estos *features* no sólo no mejoraron el resultado previo de los modelos sino que los empeoraron notablemente.

Sin embargo, partiendo de un modelo que comprendía los resultados de las redes neuronales aplicadas a los *embeddings* y al *TF-IDF*, así como el majority voting entre los algoritmos basados en árboles de cada embedding; al agregarle el porcentaje de veracidad de cada keyword y realizar un *Averaging*, se pudo observar una leve mejora en las predicciones. De esta manera se llegó, en Kaggle, a un puntaje de 0.82868.

Por otro lado en modelos de arboles se probo incluir el feature de la ubicación. Esto se afronto de 2 maneras :

1. Primero pensándolo como un atributo binario, clasificando la *location* en valida o no. (Una *location* valida es la que tiene sentido geográfico)

2. Luego como un atributo numérico, donde cada *location* valida fue representada por sus coordenadas (usando la biblioteca *geopy*) .

   La inclusión de estos *features* no mejoro el modelo, sino genero todo lo contrario. Se decidió dejar de lado estos atributos para centrarse en el análisis del texto.

5.7. Explorando el tuning de hiper-parámetros

Previamente se mencionó los dos métodos usados para la búsqueda de hiper-parámetros (Grid Search y Random Search). En las redes que usan *Embeddings* se usó principalmente Grid Search y con ello se consiguió mejorar bastante el desempeño de dichas redes. En las redes basadas en *TF-IDF* se utilizó primero un Grid Search y posteriormente se intento usar un Random Search pero este último tuvo resultados poco satisfactorios.

El tuneo de hiper-parámetros se centra en alcanzar la mejor performance de un modelo de *machine learning*. Debido a esto es un proceso que se suele llevar a cabo en las ultimas etapas de competiciones como la del presente trabajo. A esto se debe que solo se haya buscado tunear los modelos que estaban incluidos en el ensamble, los cuales son los que mejores resultados le dio al equipo.

5.8. Visualizando con reducción de dimensiones

Llegado a cierto punto el grupo ya contaba con los resultados de varios estimadores así como algunos *features* de gran importancia, por ende, se decidió observar cómo era el comportamiento de los datos obtenidos hasta el momento si se los reducía a dos dimensiones.

Para ello se aplicó PCA, UMAP, t-SNE, Laplacian Eigenmaps y MDS a un set de datos compuesto por las cuatro predicciones por majority voting de los algoritmos basados en árboles de cada uno de los *Embeddings*, los resultados asociados a las redes neuronales de dichos *Embeddings*, los valores resultantes de aplicar redes al *TF-IDF* y una última columna que posee el porcentaje de veracidad de la keyword asociada al *Tweet* (es decir, si una keyword aparece en 5 *Tweets* de los cuales 3 resultan ser sobre un desastre, entonces dicha keyword tiene un porcentaje de veracidad de 0.6). Los resultados obtenidos se muestran a continuación.

Cabe destacar que en los próximos esquemas cada punto equivale a un *Tweet*, el cual posee target igual a 1 si se trata sobre un desastre y 0, en caso contrario. También es menester aclarar que en los gráficos del set de test el color no representa que el target sea 0, se usa solo un color debido a que se desconoce el real target de los datos.

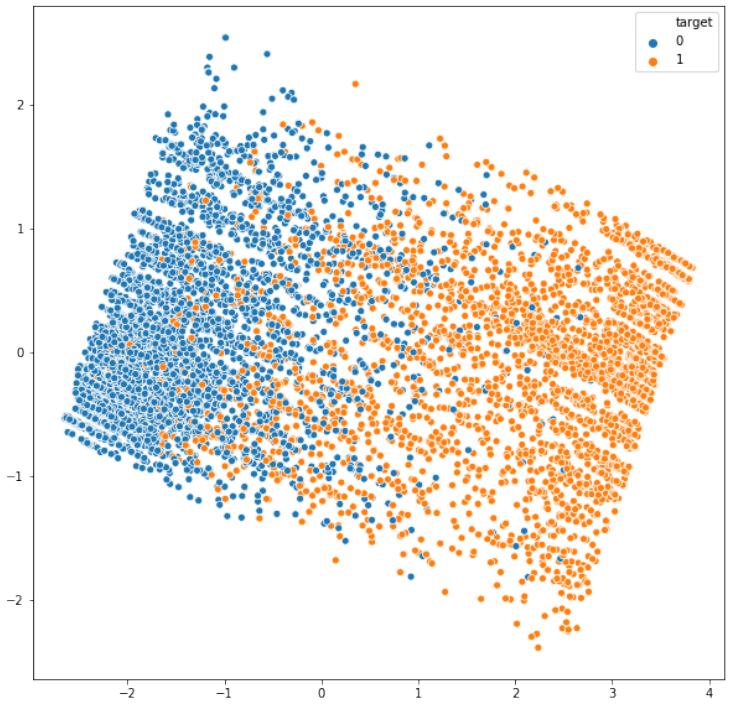

Figura 14: PCA aplicado al set de datos del training set

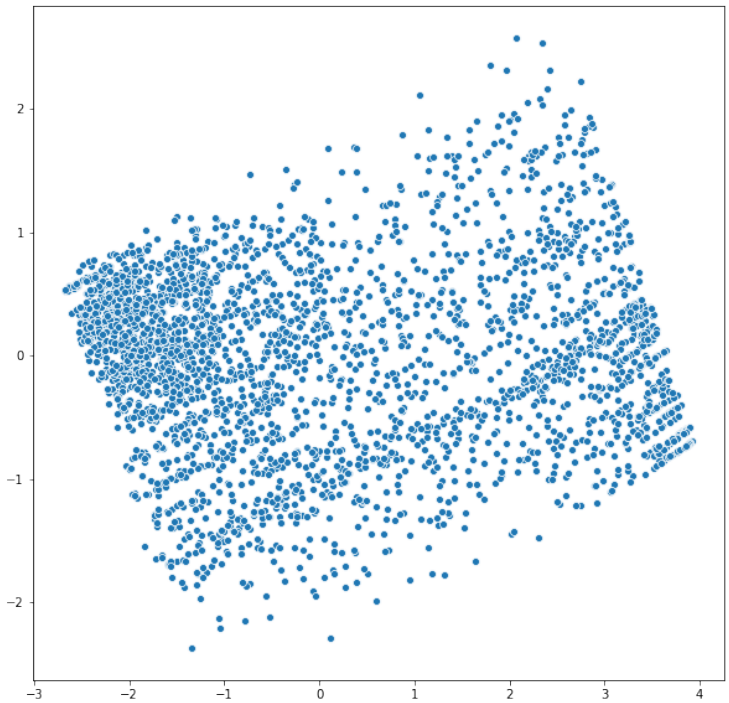

Figura 15: PCA aplicado al set de datos del testing set

Como se puede observar si bien en el training set se encuentran bien diferenciados los tweets que hablan de un desastre de aquellos que no, no se pueden apreciar clústeres lo suficientemente marcados como para que haya sido viable la aplicación de algún algoritmo de clustering que pueda ser de utilidad para las predicciones. Además, tampoco existía la certeza de que si se aplicaba una red neuronal que funcione bien en el train, el comportamiento fuese el esperado en el test, por lo que también se desestimó esta idea.

Lo mismo se puede observar que ocurre en las demás reducciones de dimensiones.

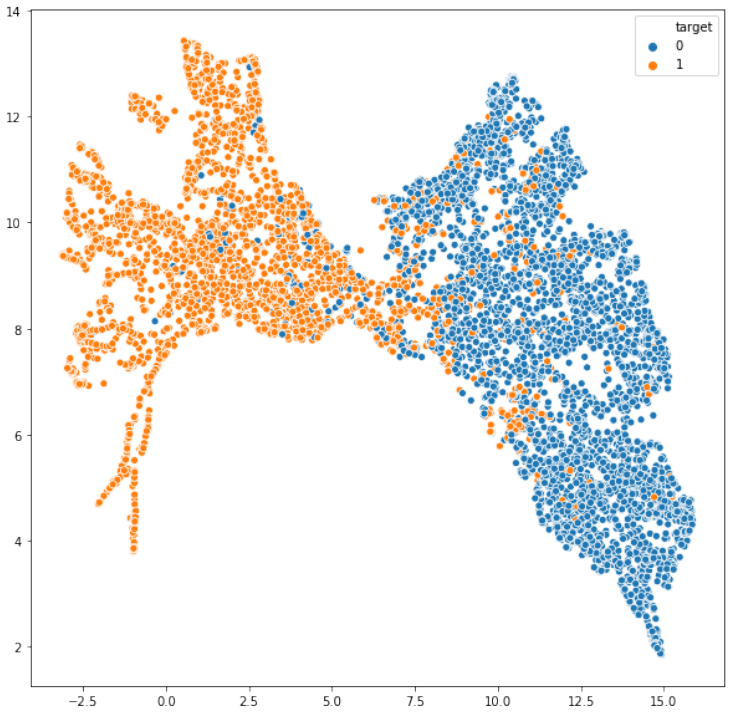

Figura 16: UMAP aplicado al set de datos del training set

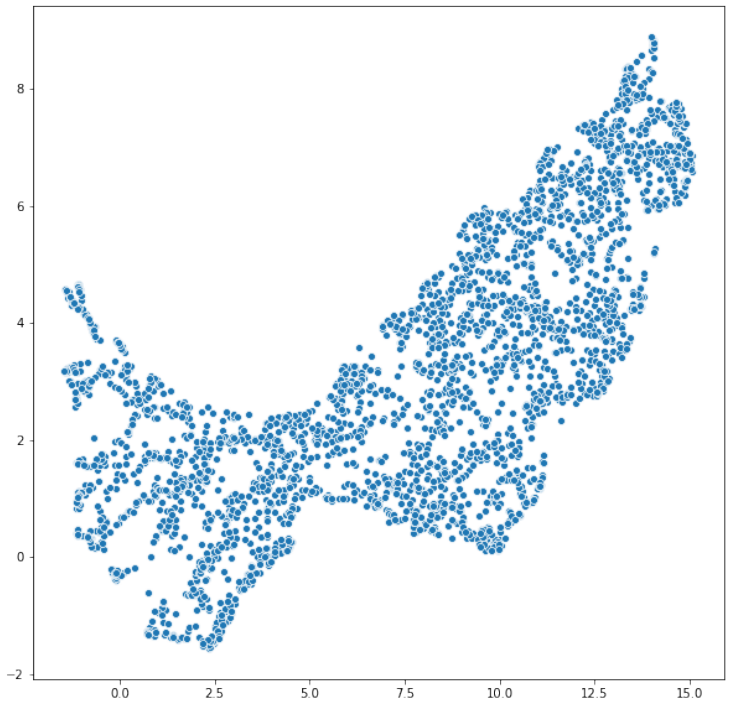

Figura 17: UMAP aplicado al set de datos del testing set

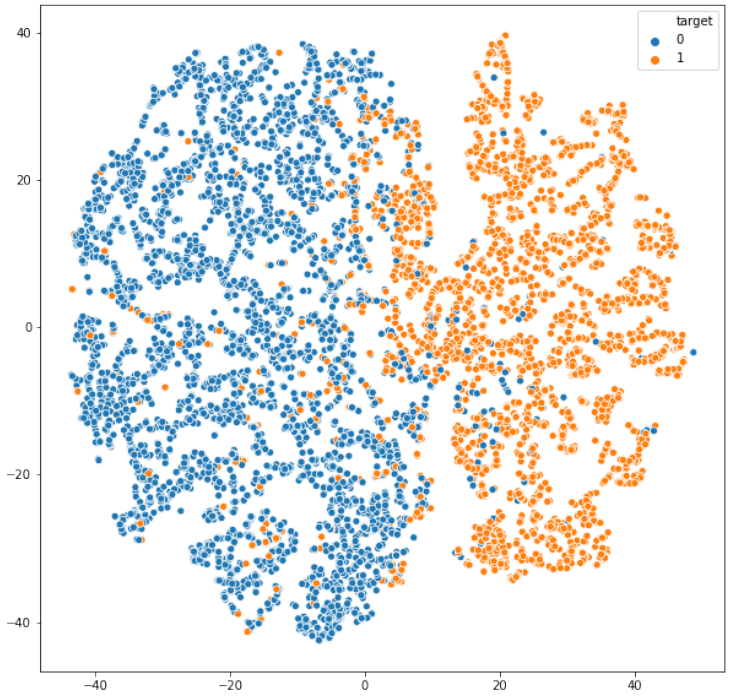

Figura 18: t-SNE aplicado al set de datos del training set

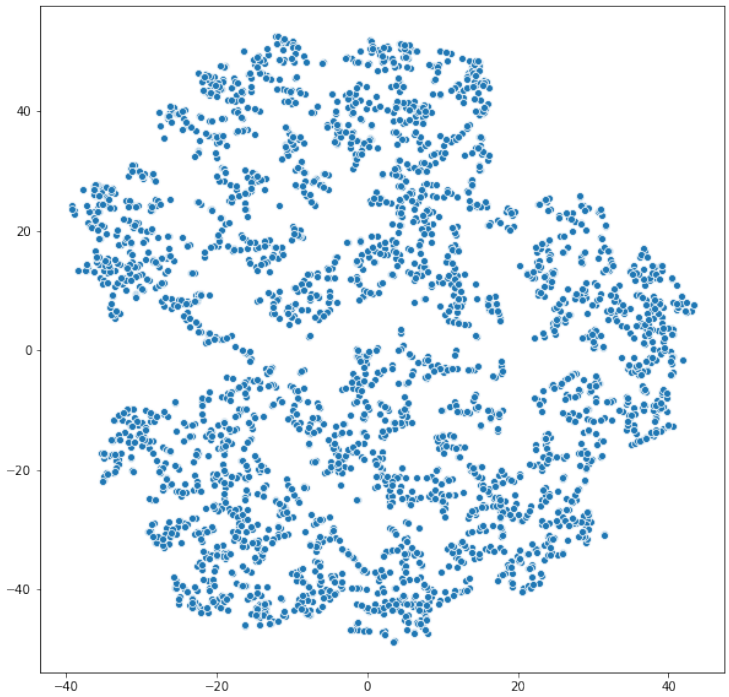

Figura 19: t-SNE aplicado al set de datos del testing set

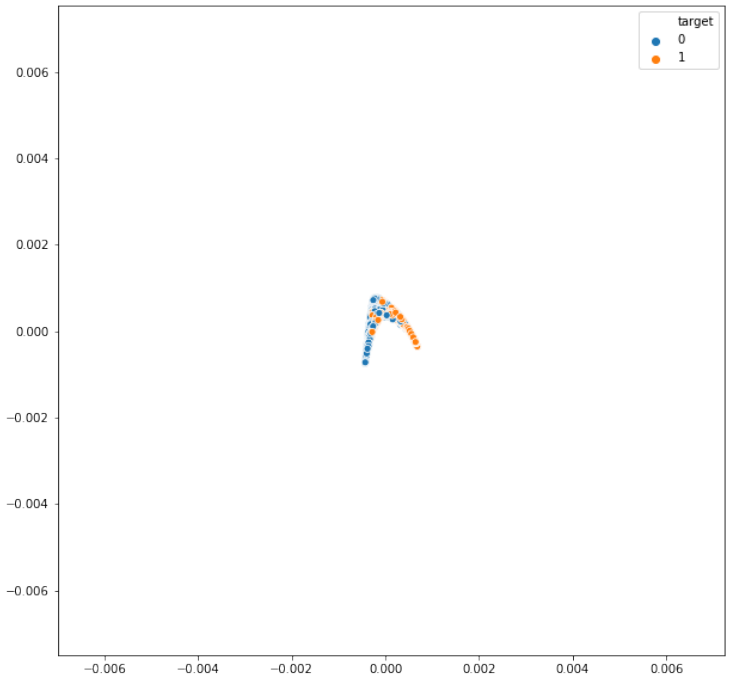

Figura 20: Laplacian Eigenmaps aplicado al set de datos del training set

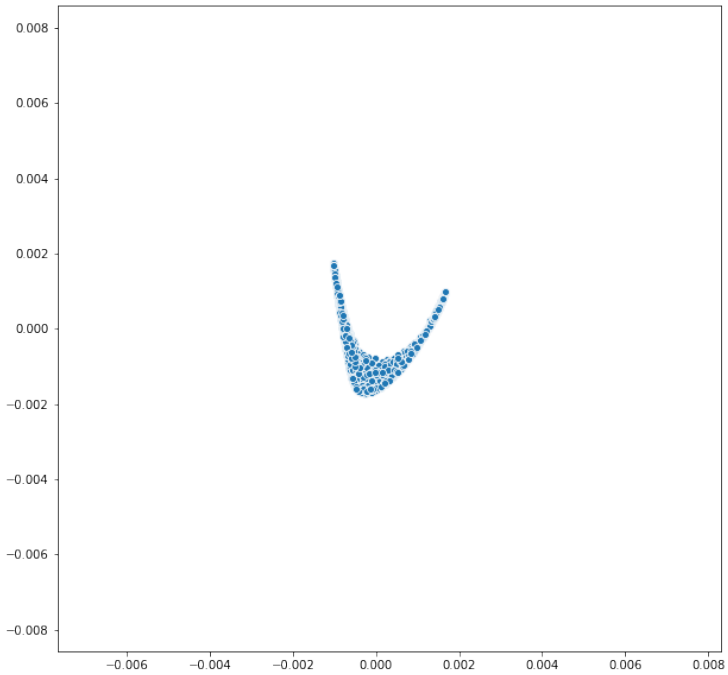

Figura 21: Laplacian Eigenmaps aplicado al set de datos del testing set

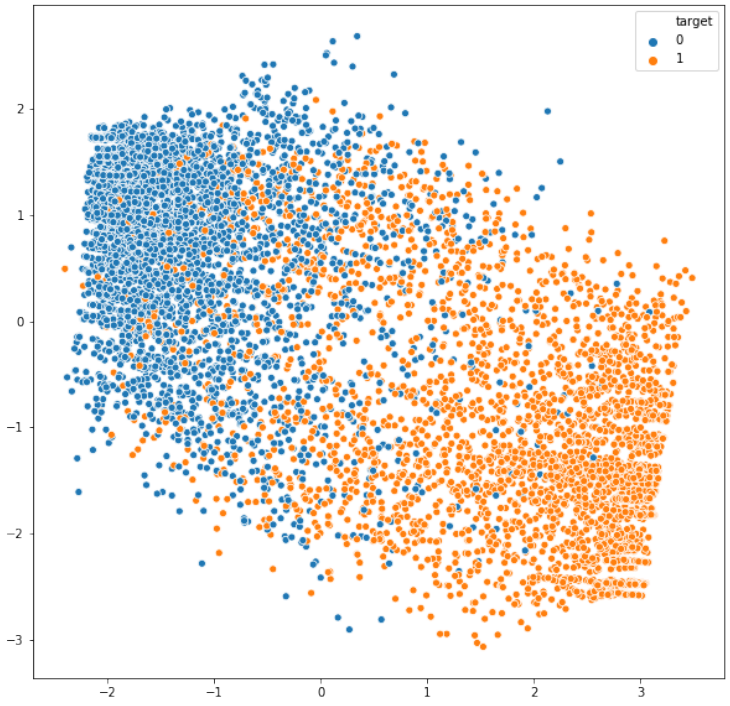

Figura 22: MDS aplicado al set de datos del training set

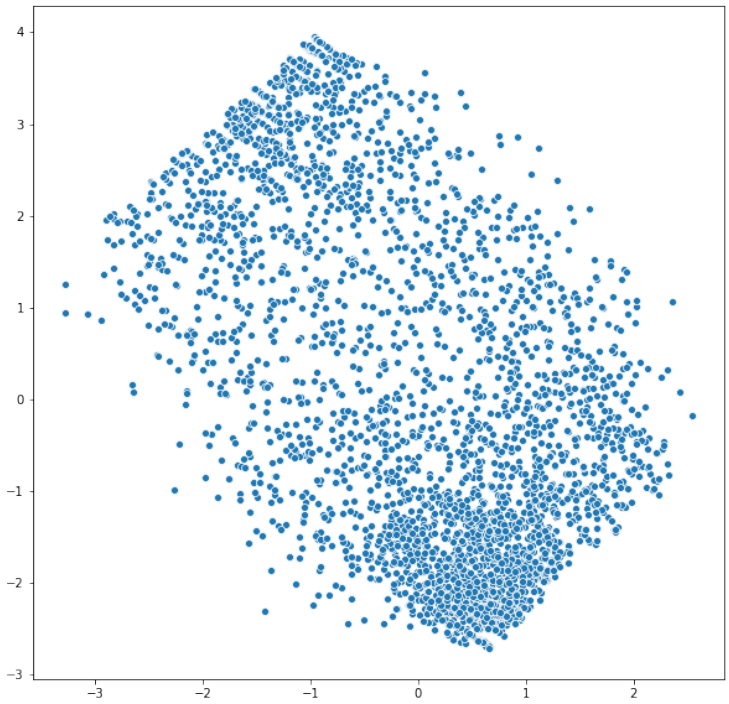

Figura 23: MDS aplicado al set de datos del testing set

5.9. El modelo esperanzador: *BERT* pre-entrenado

El descubrimiento de *BERT* e implementación en conjunto con los otros modelos trajo mejoras notables en el rendimiento. Dadas las dimensiones de la red neuronal que involucra no quedo otra alternativa que ejecutarlo en un *Notebook* de *Kaggle* acelerado por GPU, la desventaja de esto último es que se pierde la posibilidad de reproducir los resultados del entrenamiento debido a que la implementación de los algoritmos que hacen uso de la GPU para acelerar el proceso son estocásticos desde nuestro punto de vista en alto nivel.

Una solución a este inconveniente es guardar el modelo, o los pesos de cada neurona en nuestro caso, cuando se llega a un estado en el cual el modelo genera buenos resultados.

5.10. Averaging, la clave para el mejor modelo[1]

Como ya se ha mencionado, se probó con redes neuronales y ya desde un principio se notó que tenían mejor rendimiento que cualquier algoritmo o ensamble implementado hasta el momento. Al apreciar esto, no se tardó mucho en probar combinar los resultados de varias redes con la esperanza de que mejoren el resultado.

Al realizar esto, los resultados obtenidos mejoraron sustancialmente. Primero se realizó un *averaging* de los resultados de las redes con los *embeddings*. Luego se agrego a esto los resultados de las redes entrenadas con *TF-IDF*. Seguido a esto, se agrego el resultados de Transformers y *BERT*. En todos estos casos el score fue incrementando gradualmente.

Finalmente se realizó un Grid Search para encontrar la combinación que maximizaba el *Accuracy* en el set de entrenamiento. Se encontró que la combinación entre la red de transformers y *TF-IDF* con trigramas lo hacía, siendo el resultado obtenido fue de 0.84339 (el mejor score del equipo a la fecha de la presentación del trabajo).

Es destacable que este método haya tenido tal eficiencia pues los modelos solos, si bien no mostraban un desempeño terrible tampoco mostraron uno tan bueno a excepción de la red con BERT que fue el que mejor desempeño tuvo individualmente.

[1] Al día 6 de agosto de 2020

6. Análisis de los resultados y conclusiones

A lo largo de este trabajo, se han puesto a prueba múltiples metodologías con el fin de elaborar un robusto modelo de *machine learning* capaz de predecir correctamente la naturaleza de cada tweet. A partir de los modelos elaborados se extrajo una serie de conocimientos que, conforme se avanzaba con la actividad, sirvieron de guía para el grupo para identificar de mejor manera la utilización de qué algoritmos resultaría más viable que otros.

De esta manera, se puede afirmar que un modelo decente de esta índole sería prácticamente imposible de elaborar sin la utilización de *NLP* y, además, si bien los algoritmos basados en árboles de decisión son una herramienta realmente muy potente, su desempeño en este campo en particular resulta inferior al de una red neuronal.

En conclusión, el presente trabajo proporcionó una excelente oportunidad para poder poner en práctica muchos de los temas vistos a lo largo de la materia. Permitió que el equipo viva en carne propia como es el proceso de desarrollar un modelo de *machine learning* y todos los desafíos que eso involucra.

Durante la realización del trabajo se evidenció la importancia que tiene el trabajo en equipo en competencias de *machine learning* pues cada uno de los integrantes aportó ideas de gran valor para llegar al mejor resultado. La ultima instancia del trabajo se alcanzó gracias a estos aportes, que si bien muchos fueron dejados atrás ya que no conducían hacia una mejora en la *performace*, todos terminaron aportando de una u otra manera a lo que fue el resultado final. Es difícil pensar se hubiera podido llegar al mismo resultado en la misma cantidad de tiempo trabajando individualmente.

Es importante tener en cuenta que siempre quedan incontables opciones con las que se puede llegar a trabajar, tanto algoritmos de *machine learning* como de ensambles. Es ahí donde recae la responsabilidad de un *data scientist* de identificar cuales son las mejores opciones para afrontar el problema y llevarlas a cabo.

7. Referencias externas

- Link al repositorio de Github:

  https://github.com/RiedelNicolas/TP2-Organizacion-de-Datos

- GloVe Embeddings
- Modelo pre-entrenado de BERT con la arquitetura de Transformers
- Competencia de *Kaggle*
- Hyperparameter Tuning of Keras Deep Learning Model in Python
- Geocode with Python
- Text Classification using K Nearest Neighbors
- *Embedding* de W2V
- What is LightGBM, How to implement it? How to fine tune the parameters?
- LightGBM's documentation
- Understanding Random Forest
- Open Sourcing BERT: State-of-the-Art Pre-training for Natural Language Processing
- Documentacion de Sklearn
- Why is twitter so important
- Keras API Reference
- Transformers
- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding