75.06/95.58 Organización de Datos - 1C 2020

Trabajo Práctico 2

Competencia de Machine Learning

Grupo 33: "DataTravellers"

Integrantes:

- Andrés Pablo Silvestri: 85881 (silvestri.andres@gmail.com)
- Juan Manuel González: 79979 (juanmg0511@gmail.com)
- Patricio Pizzini: 97524 (pizzinipatricio@yahoo.com.ar)

Link a repositorio de GitHub:
https://github.com/patopizzini/Organizacion_Datos_1C2020/tree/master/TP2

Fecha de entrega: 10/08/2020

Contenido

1 - Introduccion

En este informe se describe el trabajo realizado para poder llegar a los distintos resultados que se fueron entregando a la competencia de Kaggle: Real or Not? NLP with Disaster Tweets[1].

La competencia consiste en predecir si un texto (tweet) estaba anunciando un desastre o no.

A partir de los datos proporcionados en el set de la competencia, se fueron extrayendo distintos features y probando diversos algoritmos de machine learning, a fin de obtener las mejores predicciones posibles.

1.1 - Organización del repositorio

En la carátula del informe se encuentra el *link* al repositorio de GitHub donde se encuentra el código utilizado en el TP. Los notebook con los distintos algoritmos y métodos ensayados se encuentran en la raíz de la carpeta TP2, en cada sección se aclarará cuál es el archivo que corresponde. Por otro lado, existe una serie de directorios con archivos adicionales, que detallamos a continuación:

Directorio	Propósito
arboles_def	Notebooks con el tratamiento de árboles.
data	Agrupa los archivos originales del dataset, los tratados y los submits.
ensambles	Notebooks con los ensambles ensayados.
informe	Copia de este informe.
models.backup.81	Backup de modelos de redes entrenadas.
tools	Herramientas auxiliares utilizadas.

[1] https://www.kaggle.com/c/nlp-getting-started/overview

Localmente, cada integrante del grupo cuenta además con los siguiente directorios:

Directorio	Propósito
bert.modules	Modelos BERT pre-entrenados.
embeddings	Embeddings GloVe utilizados para las CNN.
models	Modelos entrenados, CNN y BERT.

Estos directorios no fueron subidos al repositorio dado su elevado tamaño. Se utilizó para trabajar el *branch* ***master*** exclusivamente.

1.2 - Metodología de trabajo

Para organizar el trabajo, partimos de una lista donde anotamos los temas que nos interesaba explorar para resolver el problema planteado. La metodología consistió entonces, para esos temas, en complementar lo visto en clase con investigación en diversas fuentes online. Dichos artículos se encuentran citados a lo largo del informe en las secciones correspondientes, así como también los archivos de *embeddings* o modelos pre-entrenados utilizados.

Durante el desarrollo del trabajo, se utilizó el lenguaje de programación Python y algunas librerías extras como Scikit-learn, Pandas y Keras para facilitar el manejo de datos. El trabajo se dividió en tres ramas:

- Soluciones basadas en árboles.
- Soluciones basadas en redes neuronales.
- Ensambles.

En los dos primeros casos, el código se separó en dos secciones, la primera fue el *feature engineering* (árboles) o el pre-procesamiento de los textos (redes neuronales) y la segunda consistió en la prueba de algoritmos de machine learning y su “tuneo”.

Finalmente, en la etapa de ensambles, intentamos combinar varios de estos algoritmos, siguiendo diversas estrategias, a fin de mejorar el score obtenido.

Por último, mencionamos que esta división se aplicó también a nivel de codificación, dejando varios notebooks con distintos *feature engineering* / preprocesamiento, para luego probar los algoritmos en sus respectivos archivos, tomando como entrada la salida de los notebooks anteriores.

2 - Soluciones basadas en árboles

Podemos usar árboles de decisión para problemas en los que tenemos entradas continuas pero también categóricas. La idea principal de los árboles de decisión es encontrar aquellas características descriptivas que contienen la mayor parte de la “información” con respecto a la característica de destino y luego dividir el conjunto de datos a lo largo de los valores de estas características, de modo que los valores de la característica de destino para los sub_datasets resultantes sean lo más puros posible.

Para empezar tenemos un problema de clasificación en el que necesitamos trabajar sobre el texto que es la parte que mayor contenido nos da en relación al csv brindado, de esto se desprende que podemos usar árboles de decisión para nuestra clasificación pero que será necesario hacer un fuerte trabajo sobre los features para así obtener columnas numéricas que pueden ser entendidas por los árboles y que extraigan la mayor cantidad de información posible del texto que manejamos.

Si bien podíamos llegar a una solución aceptable con los árboles de decisión nuestra mayor esperanza era encontrar los features que nos marcaran el rumbo a seguir o por lo menos tener una idea general que nos oriente, optamos aplicar en general dos algoritmos: Uno enfocado a detectar los features más importantes que fue Random Forest y otro más enfocado en tratar de conseguir un mejor resultado o performance más acertada este fue el caso de XGBoost.

El archivo con estos modelos puede encontrarse en el repositorio, bajo el nombre:

```
arboles_def/TP2 ALGORITMOS - ARBOLES.ipynb
```

2.1 - Feature Engineering

Lo primero a plantear es que tanto usamos para este punto lo que vimos en el TP1, porque si bien teníamos muchas columnas no necesariamente eran suficientes o estaban bien, para empezar partimos con el siguiente grupo de columnas para la primera ejecución de los algoritmo de árboles:

- Cantidad de palabras.
- Longitud del texto.
- Quienes contaban o no con una keyword.
- Keywords (usando encoding)
- Cantidad de hashtags.
- Cantidad de coincidencias con un set de palabras relacionadas a desastres.
- Quienes contaban o no con ubicación.
- Cantidad de signos de exclamación y/o interrogación.

Tomando como base esos features nos tomamos un tiempo para pensar que nuevos features numéricos podíamos sacar del texto el cual era la gran fuente de información en este trabajo, pero como era necesario trabajar sobre el texto tuvimos que usar la librería de NLTK y diferentes tokenizadores para por ejemplo ir quitando stopwords, pasando a minúscula, lematizando, aplicando stemming, etc.

Entre los features que fuimos agregando tenemos los siguientes:

- Cantidad de números.
- Cantidad de letras (sin números ni símbolos).
- Cantidad de stopwords.
- Longitud del tweet sin stopwords.
- Cantidad de palabras sin stopwords.
- Longitud promedio de cada palabra.
- Longitud promedio de cada palabra sin contar stopwords.
- Módulo de la diferencia entre la cantidad de palabras y la cantidad de stopwords.
- Total de palabras sobre el total de stopwords (su relación)
- Cantidad de menciones (@)
- Cantidad de signos de puntuación.

Para este punto ya teníamos una selección bastante variada de columnas, pero sentíamos que faltaba algo fundamental y era el poder definir la calidad o sentimiento del tweet, entendimos que desde un comienzo uno podría pensar que un tweet que verdaderamente quiere expresar o informar que hay un desastre difícilmente este pueda ser un tweet alegre o positivo, es por eso que utilizamos twitter_samples y sus json positive_tweets.json / negative_tweets.json para poder definir todas aquellas palabras que se pueden asemejar a un tweet positivo o negativo y de esta manera poder hacer una mejor discriminación, terminamos entrenando un pequeño Clasificador Naive Bayes con un objetivo sencillo, poder recibir un texto y predecir si es positivo o negativo, con esto pudimos agregar dos columnas nuevas que nos defeinían lo siguiente:

- Tweet con sentimiento positivo. (probablemente no haría mención a desastre)
- Tweet con sentimiento negativo (de ser un desastre hay más chances que sea negativo)

Con respecto a las dos columnas categóricas que tuvimos es decir Keywords y Location, pensamos en aplicar algún encoding, pero para el caso de Location hacer un simple one hot encoding era impensado por lo que optamos por hacer un binary encoding, lo mismo para keywords, de todas maneras no llegamos a hacer el one hot encoding en location pero si en keywords, luego tuvimos la posibilidad de elegir entre si usar o no el binary encoding de location, como también elegir si usar el binary encoding, el one hot encoding o no usar ninguno para el feature keywords.

El archivo con este tratamiento puede encontrarse en el repositorio, bajo el nombre:

`arboles_def/TP2 PROCESAMIENTO.ipynb`

2.2 - Random Forest

Como primer medida se decidió iniciar el trabajo para la predicción usando Random Forest, entre otros motivos podemos decir que la elección está basada en el simple hecho

de que cuando un atributo es un buen predictor sus árboles van a tener mejores resultados que aquellos que usan un conjunto de atributos que no son buenos predictores, como así también sabemos que son invariantes a la escala de los atributos es decir que no necesitamos normalizarlos, lo cual entre otras cosas nos facilita la operatoria, con todo esto partimos con la idea inicial de utilizarlo para ir descubriendo qué features pueden resultar positivos y cuáles no.

Es decir la principal idea y motivación de usar Random Forest fue poder obtener o detectar aquellos features que podían darle valor a la clasificación, si bien fuimos avanzando y probando los diferentes features en un primer momento teníamos una puntuación de 0.70027 como primer submit para RF, luego lo fuimos mejorando quedándonos con los features que más nos iban aportando valor y sumado al Grid Search que hicimos para conseguir un último valor de 0.74593, este podría decirse que fue el mejor resultado que conseguimos con Ranfom Forest y ya luego dejamos de trabajar en este salvo para agregarlo en algún ensamble de prueba.

2.2.1 - Tuneo de los algoritmos y resultados obtenidos

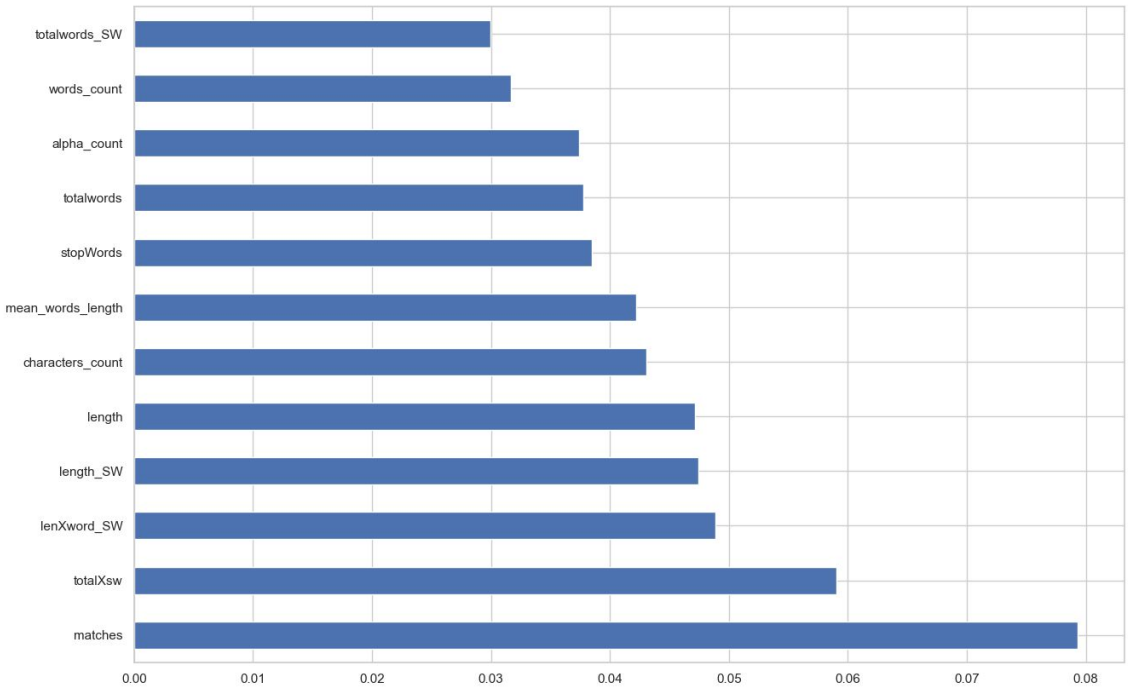
Para esta altura habíamos llegado al tope que nos podía dar Random Forest aplicando: RandomForestClassifier (porque es una clasificación lo que estamos haciendo), y los siguientes hiper-parámetros en base al grid search limitado que hicimos, y cuando decimos limitado es porque primero hicimos un random search para acotar el rango de parámetros a usar y luego sobre ese margen aplicamos un grid search siempre usando K-Fold Cross Validation, esta lógica la aplicamos en base un artículo[2] que consultamos online.

Los resultados obtenidos para nuestro caso fue el siguiente, teniendo la siguiente lista para aplicar el grid search y los hiper parámetros en verde siendo los que mejor resultado dieron:

[2] https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

- `'bootstrap': [True],`
- `'max_depth': [60, 70, 80, 90, 100, 110],`
- `'max_features': [2, 3],`
- `'min_samples_leaf': [2, 3, 4, 5],`
- `'min_samples_split': [6, 8, 10, 12],`
- `'n_estimators': [50, 100, 200, 300, 1000]`

Con esta combinación de hiper parámetros y las siguientes 12 columnas obtuvimos lo que fue el mejor resultado con Random Forest:



Vale aclarar que todo el resto de parámetros empezaba a caer por debajo del 0.01 teniendo muchos que aportan más ruido que beneficio, por lo que se optó por eliminarlos ya que a priori estábamos trabajando arriba de los 245 features los cuáles la mayoría no aportan nada, para mayor visibilidad aclaramos que los features que tienen la nomenclatura SW significa que hacen mención a las StopWords, lo que nos muestra la importancia de las mismas o que tanto inciden.

2.3 - XGBoost

XGBoost significa eXtreme Gradient Boosting. Es el algoritmo que ha estado dominando recientemente los problemas Machine learning y las competiciones de Kaggle con datos estructurados o tabulares. XGBoost es una implementación de árboles de decisión con Gradient boosting diseñada para minimizar la velocidad de ejecución y maximizar el rendimiento.

Internamente, XGBoost representa todos los problemas como un caso de modelado predictivo de regresión que sólo toma valores numéricos como entrada. Si nuestros datos están en un formato diferente, primero vamos a tener que transformarlos para poder hacer uso de todo el poder de esta librería. El hecho de trabajar sólo con datos numéricos es lo que hace que esta librería sea tan eficiente.

En nuestra primera prueba con este algoritmo conseguimos la siguiente puntuación 0.64603 lo cual era realmente muy baja en comparación a lo que nos había dado en Random Forest aplicando los mismos features, obviamente que en este punto nos faltaba aplicar todos los otros features como así también hacer un grid search para obtener los hiper parámetros óptimos. Luego ya habiendo aplicado los features más importante y quitando aquellos que generaban ruido, como así también aplicando un grid search a los hiper parámetros conseguimos un puntaje final de 0.75819 el cual no es muy alto, pero por lo menos nos sirvió para tener una aproximación aceptable con los árboles de decisión.

Si bien este resultado no fue bueno, decidimos como luego explicaremos en la parte de ensamble resguardar estas salidas y poder aplicarlas en conjunto con las otras soluciones que implementamos luego para de esta manera tener mayor diversidad y poder implementar algoritmos de manera desacoplada que aportar su “visión” de la solución.

2.3.1 - Tuneo de los algoritmos y resultados obtenidos

Para obtener los hiper parámetros aplicamos una lógica similar a la que aplicamos en Random Forest, para este caso hicimos un Random Search con cierto margen de

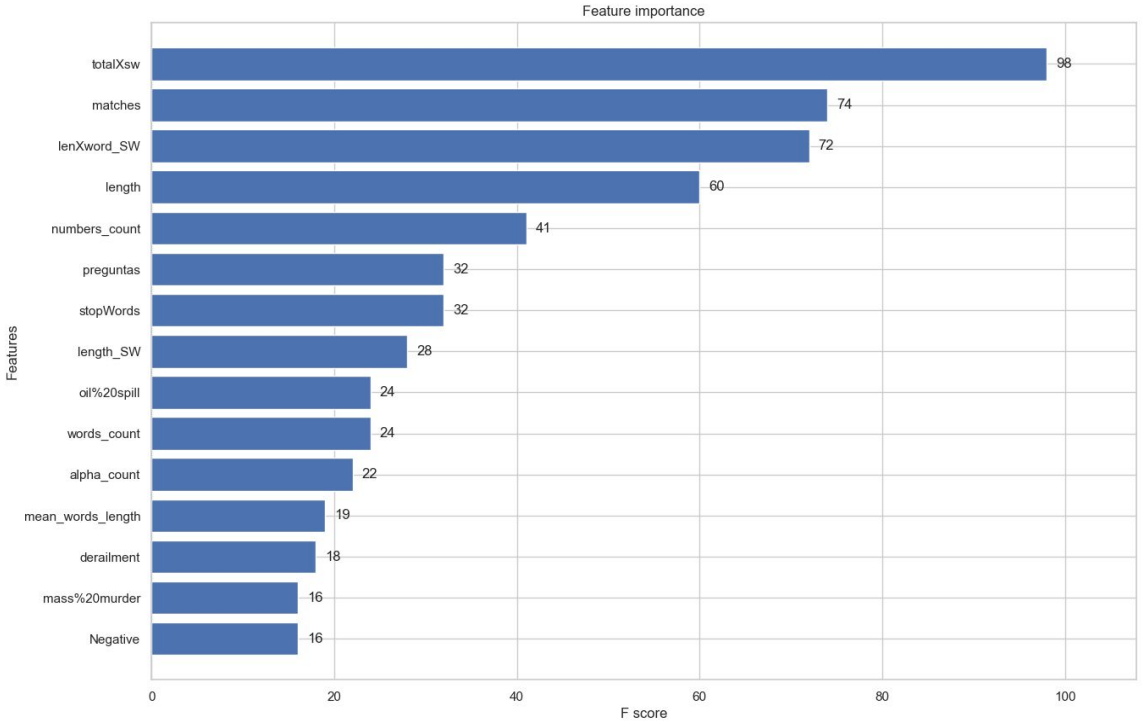
parámetros y nos quedamos con los mejores, luego sobre estos márgenes aplicamos un Grid Search, obviamente para todo esto usamos K-Fold Cross Validation como mostramos en la parte de Random Forest.

- `max_depth: [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`
- `min_child_weight: np.arange(0.0001, 0.5, 0.001)`
- `gamma: np.arange(0.0,40.0,0.005)`
- `learning_rate: np.arange(0.0005,0.3,0.0005)`
- `subsample: np.arange(0.01,1.0,0.01)`
- `colsample_bylevel: np.round(np.arange(0.1,1.0,0.01))`
- `colsample_bytree: np.arange(0.1,1.0,0.01)`

Teniendo como mejor performance la siguiente combinación de hiper parámetros:

- `objective = binary:logistic`
- `n_estimators = 500`
- `min_child_weight = 5`
- `learning_rate = 0.05017181127931773`
- `gamma = 10`
- `reg_lambda = 3`
- `max_depth = 9`
- `colsample_bytree = 0.7585033814547916`
- `subsample = 0.9779760690574663`

Luego procedemos a mostrar el gráfico que nos indica la importancia de los features que teníamos, para saber cuáles eran los más importante o que mayor peso tenían a la hora de hacer la clasificación.



Luego de estos features tenemos valores muy bajos que no aportan nada, como comentamos en la sección para Random Forest, decidimos quitar aquellos features que no nos aportan nada y nos generaban más ruido que beneficio.

3 - Soluciones basadas en redes neuronales

En esta sección del informe trataremos los métodos basados en redes neuronales que fueron ensayados. Para todos ellos utilizamos Keras, corriendo sobre Tensorflow.

3.1 - Preprocesamiento de la entrada

Dada la importancia del formato de los datos para el caso de las redes neuronales, y su tratamiento previo para trabajar con machine learning en general, decidimos crear un notebook especial para el procesamiento de los set de train y test, antes de utilizarlos con los algoritmos. Dicho notebook se puede encontrar en el repositorio, con el nombre:

`TP2 pre-procesamiento para redes.ipynb`

Esencialmente, el archivo se alimenta de los archivos originales provistos en la competencia, 'train.csv' y 'test.csv', y devuelve una copia de cada uno de ellos, con el procesamiento aplicado.

Dicho procesamiento es ejecutado en varias etapas, y fue ideado para ser configurable, de modo de poder combinar distintas opciones y hacer pruebas con ellas. La configuración es controlable mediante una serie de variables booleanas, cada una de ellas asociada a una operación distinta. El orden en que se aplica el tratamiento no puede ser variado.

A continuación detallaremos los pasos aplicados:

Etapa	Acción
Pasaje a minúsculas	Convierte el texto completamente a minúsculas, utilizando la función de Python *lower*. Aplica a las columnas keyword, location y text.

Limpieza básica	Realiza una limpieza de los textos utilizando una serie de *regular expressions*. Remueve signos de puntuación, URLs y caracteres no imprimibles, entre otros. Aplica a las columnas keyword, location y text.
Agregado de keywords	Realiza un *append* del contenido de la columna keyword a la columna text.
Agregado de location	Realiza un *append* del contenido de la columna location a la columna text.
Remoción de stopwords	Quita stopwords de la columna text, utilizando nltk, para el idioma inglés.
Stemming	Aplica esta operación al contenido de la columna text, utilizando el *porter stemmer* de nltk.
Lematización	Aplica esta operación al contenido de la columna text, utilizando el *wordnet lemmatizer* de nltk.
Aumento del set	Realiza *data augumentation* sobre el set de train, mediante el uso de la biblioteca *nlpaug*.
Chequeo final	Remueve espacios duplicados en la columna text. Si algún registro de la columna text queda vacío, lo completa con un 0.

En el notebook puede apreciarse que la primera operación que realizamos es el manejo de NaN para las columnas keyword y location, en ambos casos reemplazando por un string vacío, a fin de poder realizar las operaciones de concatenación subsiguientes.

Respecto a las operaciones hechas mediante el uso de nltk, las realizamos solamente para el idioma inglés, dado que determinamos en el TP1 que el 100% del set se encuentra en este idioma.

Sobre el aumento del set de datos, nos pareció interesante ensayar esta variante, dado que con los algoritmos de redes neuronales conviene tener sets grandes. Según pudimos investigar[3], el espíritu de esta técnica consiste en agregar tweets ficticios, generados en

[3] https://towardsdatascience.com/data-augmentation-in-nlp-2801a34dfc28

base a los existentes, pero sin alterar su categoría. Utilizamos para ello, dos estrategias provistas por la biblioteca utilizada, logrando duplicar su tamaño:

- Reemplazo de alguna palabra al azar por un sinónimo.
- Swap de dos palabras en un tweet, con las posiciones determinadas al azar.

Sobre el procesamiento final, nos gustaría aclarar que algunas de las operaciones más agresivas, como la remoción de stopwords, nos dejaba algunos registros vacíos, ya que por ejemplo el contenido eran 2 stopwords y una URL, por lo que decidimos rellenar esas instancias con el número 0, para permitir el correcto funcionamiento de los algoritmos.

Finalmente, mostramos un ejemplo de salida de este notebook, que es de la forma:

```
Operación finalizada!

Pasaje a minúsculas:   True
Limpieza básica:       True
Agregado de keywords:  True
Agregado de location:  False
Remoción de stopwords: False
Stemming:              False
Lematización:          False
Aumento del set:       False
Chequeo final:         True

Generado train: 'data/processed/train.2020.08.08T03.14.12.423304.csv' - (7613)
registros.
Generado test:  'data/processed/test.2020.08.08T03.14.12.423304.csv' - (3263)
registros.
```

Listamos un resumen de las operaciones realizadas, así como también la cantidad de registros en cada archivo (esto es importante por el uso de *data augumentation*). Cada archivo tratado tiene asociado al nombre de archivo la fecha de creación, para permitirnos hacer un mejor seguimiento de los archivos. Se aclara que solo están en el repositorio los sets asociados a algún submit significativo.

3.2 - Modelo secuencial con CNN

Para iniciar nuestro trabajo con redes neuronales, luego de investigar[4] su uso y realizar algunas pruebas preliminares, probamos un modelo secuencial implementado en Keras, utilizando embeddings pre-entrenados y sendas capas Conv1D/Maxpooling. Este modelo se puede consultar en el repositorio, en el archivo:

`TP2 KERAS Conv1D sin pre-procesamiento.ipynb`

Los embeddings utilizados con este modelo fueron tomados de GloVe, en particular la fuente es de Twitter, con las siguientes características:

Twitter (2B tweets, 27B tokens, 1.2M vocab, uncased, 200d vectors, 1.42 GB download)[5]

En principio esto nos pareció óptimo, por lo que decidimos utilizar el set de embeddings de 200 dimensiones. Tomando como punto de partida los archivos devueltos por el preprocesador expuesto en la sección anterior, se tokenizan los textos de los tweets y se genera una matriz de embeddings, a fin de utilizarlos en la capa de entrada de la red. Una métrica interesante que se desprende de este tratamiento es conocer cuántas palabras tenemos representadas en nuestro set por los embeddings:

`Cobertura de vocabulario: 0.7219241982507288`

Por ejemplo, en este caso tenemos aproximadamente el 72% del vocabulario representado por los embeddings. A fin de maximizar este número, probamos varias combinaciones de configuración en nuestro preprocesador, hasta encontrar el valor óptimo del ejemplo.

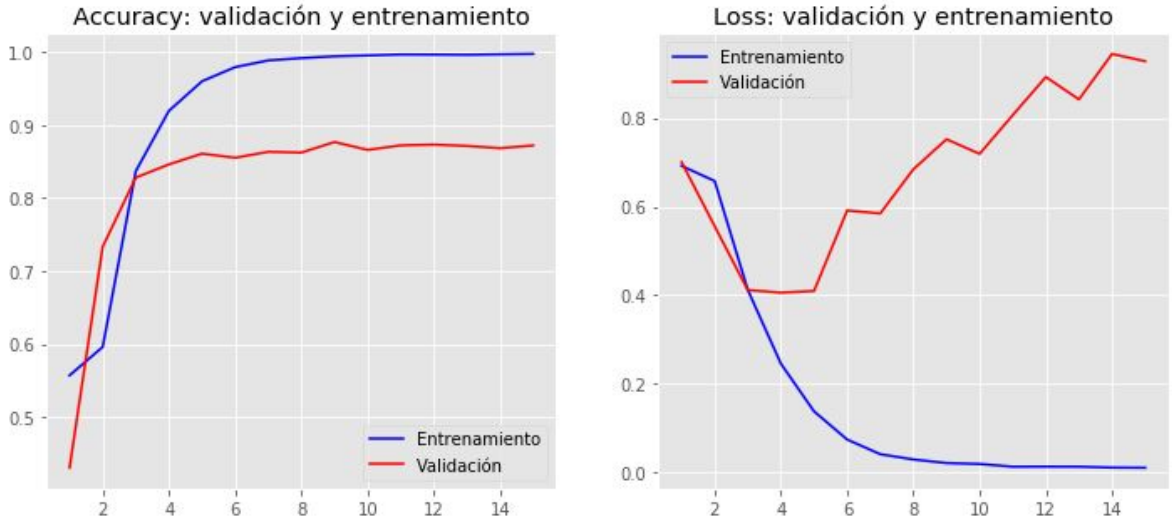
[4] https://realpython.com/python-keras-text-classification/
[5] http://nlp.stanford.edu/data/glove.twitter.27B.zip

El resumen del modelo utilizado es:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 200, 200)          3430000
_________________________________________________________________
dropout_4 (Dropout)          (None, 200, 200)          0
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 199, 64)           25664
_________________________________________________________________
global_max_pooling1d_3 (Glob (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout_5 (Dropout)          (None, 64)                0
_________________________________________________________________
predictions (Dense)          (None, 1)                 65
=================================================================
Total params: 3,459,889
Trainable params: 3,459,889
Non-trainable params: 0
_________________________________________________________________
```

Contamos como input con la capa de Embeddings de Keras, y como salida una capa densa de 1D, ya que tenemos que clasificar 2 posibles categorías. Las capas ocultas que forman esta red profunda están dadas por dos capas de dropout intercaladas, a fin de prevenir overfitting, una capa Conv1D, seguida una de Maxpooling luego de las convoluciones y una capa densa. En este caso, la capa de convolución fue configurada para trabajar con bigramas (mediante el hiper-parámetro *kernel_size*). Como algoritmo de optimización utilizamos *Adam*, y la función de loss es *binary_crossentropy*. La métrica utilizada es *accuracy*.

Entrenamos este modelo siguiendo un esquema de K-Fold cross validation, con k=10. Esto nos permitió ir monitoreando a medida que avanzaba el entrenamiento, cómo se alteraban estos parámetros. Para ello utilizamos una función que nos permitió graficar esto por *epoch,* como se muestra en el siguiente ejemplo (los gráficos representan un fold con el set de datos aumentado, de 15 *epochs*):



Esto nos permitió ver, para cada fold, la progresión del entrenamiento, y optimizar algunos parámetros. Por ejemplo, sobre la cantidad de *epochs*, vemos que no tiene sentido ir más allá de los 5 epochs, dado que el valor de *accuracy* de validación se estabiliza, y se dispara el de *loss*.

Para realizar la predicción, tomamos las siguientes estrategias:

- En cada iteración del cross-validation guardamos el modelo con el resultado, y generamos al final la predicción con el que tuvo mejor loss.
- Al finalizar el proceso predecimos nuevamente con el set de train completo.

Esto nos genera dos submits, que subimos a Kaggle a fin de evaluar los resultados. Luego de cada generación, imprimimos por pantalla un resumen con el nombre de los archivos generados, de la forma:

```
Operación finalizada!

Generado submit: 'data/submits/submission.2020.08.03T00.03.46.261681.csv' - (3263)
registros.
```

3.2.1 - Tuneo de los algoritmos y resultados obtenidos

Entre las posibles mejoras que planteamos al modelo descripto en la sección anterior fue la optimización de hiper-parámetros mediante el uso de grid search, como detallamos a continuación:

Se realizó un grid search para ver cual era el tamaño óptimo de *batch_size* y cual de *epochs*, *entre* 1 y 10, igualmente se había notado anteriormente que después de 5 *epochs* la curva tendía a estabilizarse. Se utilizó para las pruebas un fold de 2 y el texto limpio de stopwords y transformado a minúsculas.

Concluímos que los mejores valores a utilizar fueron:

```
epochs = 1

batch_size = [4, 7]
```

Este tratamiento puede consultarse en el notebook:

`TP2 KERAS Conv1D - AUG - GridSearch.ipynb`

También intentamos agregar más capas a la red y cambiar las funciones de activación de las distintas capas, pero sin éxito. En cuanto al preprocesador, las opciones que dieron los mejores resultados fueron:

```
Pasaje a minúsculas:   True
Limpieza básica:       True
Agregado de keywords:  False
Agregado de location:  False
Remoción de stopwords: False
Stemming:              False
Lematización:          False
Aumento del set:       False
```

```
Chequeo final:          True
```

Sobre esto, nos gustaría comentar que el uso de tanto la remoción de *stopwords*, *stemming*, *lemming* y *data augumentation* nos bajaron, aunque en distinta medida, los puntajes. En particular, el uso de un set aumentado hizo que el puntaje cayera en promedio un 4%, por lo que fue descartado en pruebas con los otros modelos. Nos hubiese gustado poder investigar un poco más cómo implementar esta estrategia en forma exitosa, dado que por lo que pudimos ver se usa bastante en el ambiente.

De entre todas las combinaciones probadas, tanto a los archivos de entrada como al modelo, comenzamos con un *score* de 0.76494, y luego de las optimizaciones logramos llegar a un puntaje de 0.79497.

3.3 - Modelo funcional con CNN

Siendo que no podíamos mejorar el máximo score obtenido en la sección anterior, decidimos buscar la forma de implementar un modelo más complejo. Llegamos a un modelo funcional[6], basado también en Keras. El archivo con este modelo puede encontrarse en el repositorio, bajo el nombre:

`TP2 KERAS 3xConv1D, embeddings.300d.ipynb`

Aprovechando todo lo realizado en la sección anterior, reemplazamos el modelo y los embeddings, por otros de GloVe, pero de 300 dimensiones:

Common Crawl (42B tokens, 1.9M vocab, uncased, 300d vectors, 1.75 GB download)[7]

[6] https://towardsdatascience.com/another-twitter-sentiment-analysis-with-python-part-11-cnn-word2vec-41f5e28eda74

[7] http://nlp.stanford.edu/data/glove.42B.300d.zip

Como vemos, la fuente no es más Twitter pero tenemos más tokens y mayor vocabulario. En este caso, la cobertura de vocabulario nos dio:

```
Cobertura de vocabulario: 0.77944649669957
```

La configuración del preprocesador de texto se mantuvo igual a la sección anterior. El resumen de este modelo es:

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 100)]        0
__________________________________________________________________________________________________
embedding (Embedding)           (None, 100, 300)     4953900     input_1[0][0]
__________________________________________________________________________________________________
conv1d (Conv1D)                 (None, 99, 100)      60100       embedding[0][0]
__________________________________________________________________________________________________
conv1d_1 (Conv1D)               (None, 98, 100)      90100       embedding[0][0]
__________________________________________________________________________________________________
conv1d_2 (Conv1D)               (None, 97, 100)      120100      embedding[0][0]
__________________________________________________________________________________________________
global_max_pooling1d (GlobalMax (None, 100)          0           conv1d[0][0]
__________________________________________________________________________________________________
global_max_pooling1d_1 (GlobalM (None, 100)          0           conv1d_1[0][0]
__________________________________________________________________________________________________
global_max_pooling1d_2 (GlobalM (None, 100)          0           conv1d_2[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 300)          0
global_max_pooling1d[0][0]

global_max_pooling1d_1[0][0]

global_max_pooling1d_2[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 256)          77056       concatenate[0][0]
__________________________________________________________________________________________________
dropout (Dropout)               (None, 256)          0           dense[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 1)            257         dropout[0][0]
__________________________________________________________________________________________________
```

```
activation (Activation)         (None, 1)            0           dense_1[0][0]
==================================================================================
Total params: 5,301,513
Trainable params: 5,301,513
Non-trainable params: 0
__________________________________________________________________________________
```

Esencialmente este modelo tiene tres capas convolucionales que trabajan en paralelo, con bigramas, trigramas y cuatrigramas (cada capa tiene un kernel_size de 2, 3 y 4 respectivamente). Luego del maxpooling de cada capa convolucional, se concatenan los resultados de estas tres capas. Como algoritmo de optimización utilizamos *Adam*, y la función de loss es *binary_crossentropy*. La métrica utilizada es *accuracy*.

3.3.1 - Tuneo de los algoritmos y resultados obtenidos

Sometimos a este modelo a las mismas estrategias de optimización que a la red secuencial. De entre todas las combinaciones probadas, tanto a los archivos de entrada como al modelo, comenzamos con un *score* de 0.80753, y luego de las optimizaciones logramos llegar a un puntaje de 0.81428.

Este modelo se encuentra guardado entrenado en el repositorio, se puede reproducir el mejor submit ejecutando el notebook:

`TP2 KERAS .81 cargando modelo guardado.ipynb`

3.4 - BERT

Investigando en internet posibles soluciones y otros usos de redes neuronales, llegamos al algoritmo open-source BERT creado por Google. Encontramos varias referencias donde se

mencionaba este algoritmo como el estado del arte en NLP, por lo que decidimos experimentar con él, tomando como referencia un artículo[8] de Kaggle.

En pocas palabras, lo que hace el algoritmo es aprender el contexto de las palabras en forma bidireccional, analizando cada una de ellas.

Se entrena de dos formas, primero se enmascaran algunas palabras y las predice según el contexto en el que se encuentran, luego predice si dos oraciones son consecutivas o no.

El modelo queda definido de la siguiente forma:

```
Model: “model”
_________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
=========================================================================================
input_word_ids (InputLayer)     [(None, 160)]        0
_________________________________________________________________________________________
input_mask (InputLayer)         [(None, 160)]        0
_________________________________________________________________________________________
segment_ids (InputLayer)        [(None, 160)]        0
_________________________________________________________________________________________
keras_layer (KerasLayer)        [(None, 1024), (None 335141889
input_word_ids[0][0]
                                                     input_mask[0][0]
                                                     segment_ids[0][0]
_________________________________________________________________________________________
tf_op_layer_strided_slice (Tens [(None, 1024)]       0           keras_layer[0][1]
_________________________________________________________________________________________
dense (Dense)                   (None, 1)            1025
tf_op_layer_strided_slice[0][0]
=========================================================================================
Total params: 335,142,914
Trainable params: 335,142,913
Non-trainable params: 1
_________________________________________________________________________________________
```

[8] https://www.kaggle.com/friskycodeur/nlp-with-disaster-tweets-bert-explained

Destacamos que este es un algoritmo que llevó mucho tiempo en ejecutar, cada epoch demoró en promedio unas 5 horas de entrenamiento. Las predicciones tardaron en promedio unos 45 minutos.

Esto nos llevó a investigar[9] algunos métodos de optimización, como usar por ejemplo la GPU para acelerar los cálculos, pero sin éxito. Entendemos sin embargo, que este tipo de algoritmos está diseñado para correr en otro tipo de hardware, por lo que nos conformamos con la performance obtenida en nuestros equipos de portátiles y de escritorio.

Como algoritmo de optimización utilizamos *Adam*, y la función de loss es *binary_crossentropy*. La métrica utilizada es *accuracy*.

El archivo con este modelo puede encontrarse en el repositorio, bajo el nombre:

`TP2 Bert final.ipynb`

3.4.1 - Tuneo de los algoritmos y resultados obtenidos

Dada la cuestión de performance planteada en el punto anterior, vimos limitadas nuestras opciones para hacer optimizaciones, por ejemplo grid search, o cross-validation como en los casos anteriores.

Basamos nuestra configuración en los valores recomendados por las fuentes consultadas, y separamos con un 20% el set de train para hacer validación. Si probamos en cambio variar el valor del batch size, utilizando potencias de dos, probando para: 2, 4, 8 y 16, seleccionando este último valor. Estas pruebas las corrimos con 1 epoch y la configuración del preprocesador de las secciones anteriores. Los valores utilizados fueron:

```
maxlen = 160
```

```
validation_split=0.2
```

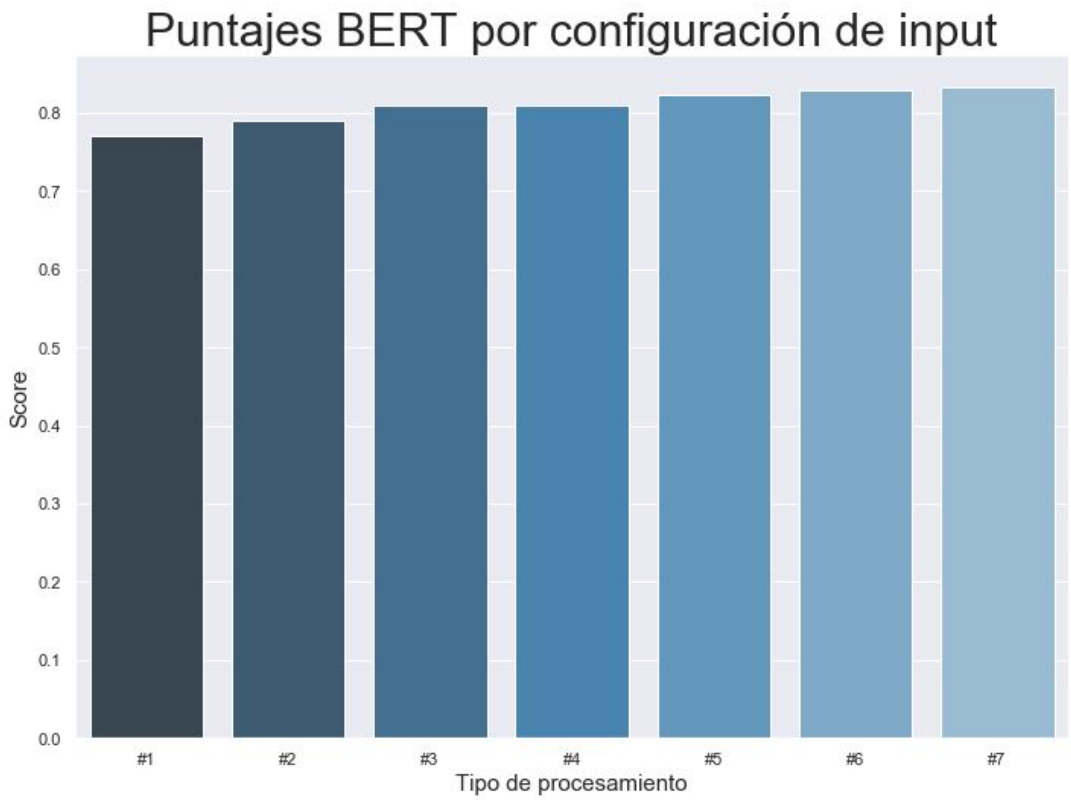
[9] https://developer.nvidia.com/blog/training-and-fine-tuning-bert-using-nvidia-ngc/

```
epochs=[1, 2, 5]

batch_size=[2, 4, 8, 16]
```

Para intentar compensar esto, probamos todas las combinaciones de nuestro preprocesador (salvo el aumento del set de datos) con un *epoch*, para luego correr la mejor combinación por 5 *epochs*. Esto nos llevó, por ejemplo, a probar 2 modelos pre-entrenados: *cased*[10] y *uncased*[11] según el caso.

En el siguiente gráfico podemos ver cómo afectó el score las principales opciones ensayadas:



El tipo de procesamiento se representa en la siguiente tabla:

[10] https://tfhub.dev/tensorflow/bert_en_cased_L-24_H-1024_A-16/2
[11] https://tfhub.dev/tensorflow/bert_en_uncased_L-24_H-1024_A-16/2

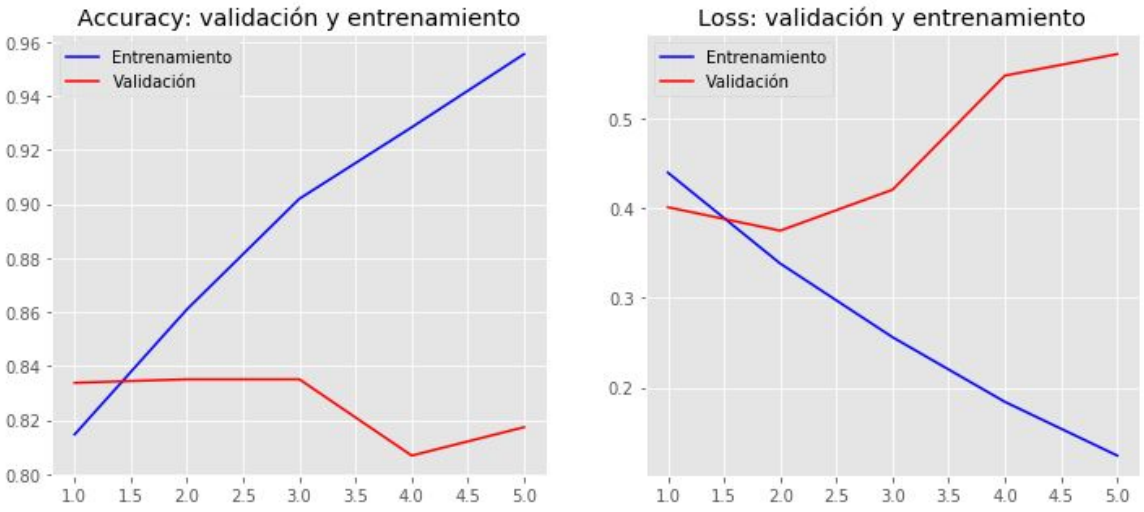
Número	Operaciones involucradas
1	Pasaje a minúsculas Limpieza básica Agregado de keywords Lemming
2	Pasaje a minúsculas Limpieza básica Agregado de keywords Stemming
3	Limpieza básica
4	Pasaje a minúsculas Limpieza básica Agregado de keywords Remoción de stopwords
5	Pasaje a minúsculas Limpieza básica Agregado de keywords Agregado de location
6	Pasaje a minúsculas Limpieza básica
7	Pasaje a minúsculas Limpieza básica Agregado de keywords

De inmediato notamos que el resultado del algoritmo era bueno, ya que la primera prueba arrojó un resultado de 0.82991. Al agregar los keywords, el *score* subió a 0.83266. Finalmente, pudimos obtener nuestro mejor resultado en la competencia al ejecutar el algoritmo por 5 *epochs*, con un batch size de 16, y la siguiente configuración de entrada:

```
Pasaje a minúsculas:   True
Limpieza básica:       True
Agregado de keywords:  True
Agregado de location:  False
Remoción de stopwords: False
Stemming:              False
```

```
Lematización:          False
Aumento del set:       False
Chequeo final:         True
```

El resultado del entrenamiento puede verse en el siguiente plot:



Como puede observarse, el *loss* de validación comienza a aumentar luego del segundo epoch, por lo que nos quedamos con ese resultado para predecir. Obtuvimos un resultado de 0.83634 y la ejecución demoró unas 25 horas.

Este modelo se encuentra guardado entrenado (no lo tenemos en el repositorio debido a su tamaño), pero de tenerlo se podría reproducir el mejor submit ejecutando el notebook:

`TP2 Bert .83 cargando modelo guardado.ipynb`

4 - Ensambles

Para esta instancia tenemos varios algoritmos funcionando y produciendo un resultado para esta clasificación, por un lado algunos algoritmos que son árboles de decisión y otros que son redes neuronales, y sin entrar en detalle en cada resultado en particular quisimos hacer una lógica de ensambles que nos permita producir una solución en la que cada algoritmo pueda dar su "punto de vista" aportando de esta manera su resultado a un resultado global general.

Hacer esto también nos permitió juntar por un lado algoritmos que tenían puras columnas numéricas (muchos features) y algoritmos que tenían al texto como su gran fuente de información, si bien es cierto que podríamos haber dejado de usar Conv1D y usar Conv2D para mezclar el texto con features numéricos tuvimos la complejidad de tener que reestructurar gran parte de todo lo que ya teníamos hecho y sumaba mucha complejidad y restaba claridad, por lo que optamos por una salida un poco más prolija y así trabajar con los algoritmos de manera desacoplada y que cada uno aporte su información o punto de vista por separado pero siempre aprovechando toda la información que nos presentaba el set de datos.

Teniendo algoritmos tan diferentes optamos por dos alternativas que nos parecieron las más claras y accesibles para "juntar todo" y poder manejar los algoritmos de manera completamente desacoplada y producir un resultado en común que representara esa idea de dar su opinión o punto de vista. La investigación que realizamos para hacer estos algoritmos de ensamble estuvo basada en un artículo[12] que encontramos online.

[12] https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/

Como comentamos en el párrafo anterior tomamos dos alternativas:

Max Voting: Este método se utiliza generalmente para problemas de clasificación. En esta técnica, se utilizan varios modelos para hacer predicciones donde cada predicción se termina considerando como un "voto". Las predicciones que obtienen la mayoría de los votos se utilizan como predicción final.

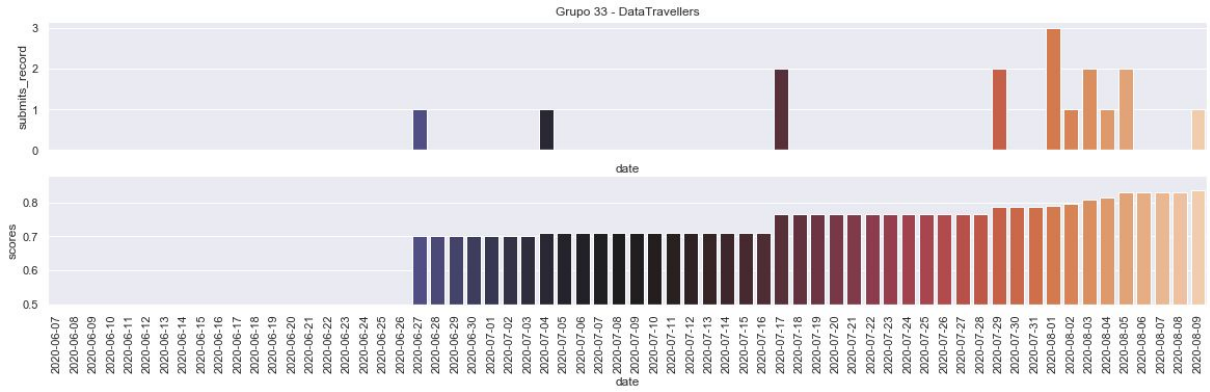
Weighted Average: Ésta es una extensión del método de promedios, a todos los modelos se les asignan diferentes pesos que definen la importancia de cada modelo para la predicción, aquí la idea es simplemente asignarle un peso según algún criterio que le atribuya a un modelo u otro un mayor peso, en nuestro caso decidimos utilizar los pesos en BERT y CNN porque fueron los algoritmos que más daban, mientras que para los árboles pusimos un peso menor.

De todas maneras estos ensambles los fuimos probando a medida que íbamos mejorando y agregando nuevos algoritmos a nuestro haber, empezamos con un resultado de 0.72908 lo cual en general nos había hecho bajar la performance general del trabajo, para luego llegar como máximo a una performance de unos 0.81489, vale aclarar que no siempre aplicamos todos los algoritmos sino que fuimos variando y alternando, para esto teníamos la limitación de los submits de *kaggle* por lo que nos tuvimos que basar en una herramienta para poder tener nuestras propias predicciones y así poder probar muchas más combinaciones de ensambles.

Los archivos con este tratamiento pueden encontrarse en el repositorio, bajo la carpeta `'ensambles'`.

5 - Desarrollo de la competencia

En esta sección describiremos nuestro progreso en la competencia. Para ello nos basaremos en la herramienta de *leaderboard* realizada por nuestros compañeros y compartida en Google Colaboratory, donde, además de obtener información sobre la posición, pudimos ver el crecimiento de nuestro score:



Como se puede ver, arrancamos con un score muy bajo, cercano a 0.70 y desde ahí fuimos incrementando nuestra performance hasta llegar al *score* definitivo.

En la segunda mitad de julio fue cuando probamos por primera vez el algoritmo de redes neuronales de la sección 3.2 y se puede ver que superamos el 0.75. Estuvimos estancados durante dos semanas aproximadamente, donde probamos varias estrategias, sobre todo tuneo de hiperparametros, no pudiendo sin embargo superar el *score* de 0.80.

Este valor fue alcanzado, y luego superado al probar y luego tunear la red de la sección 3.3, a fines de Julio.

Las últimas mejoras en el score se dieron a partir de Agosto, gracias a la implementación de BERT, algoritmo que nos permitió obtener nuestro mejor puntaje en la competencia, 0.83634.

5.1 - Herramientas adicionales

Utilizamos dos herramientas adicionales, para ayudarnos durante el desarrollo del trabajo práctico.

5.1.1 - Leaderboard

Esta herramienta fue compartida en Google Colaboratory[13] por nuestros compañeros, y nos permitió conocer nuestro lugar en la tabla, como así ver otros datos interesantes de su desarrollo y la performance de otros grupos. Hicimos una copia local del notebook en nuestro repositorio para poder usar nuestras API keys en forma segura.

5.1.2 - Resultado en Kaggle

La segunda herramienta que usamos fue una creada por nosotros y nos permitió predecir el resultado que nuestro submit tendría en Kaggle. Esto fue posible dado que durante nuestra investigación, encontramos un artículo[14] en Kaggle donde se explicaba cómo inferir los resultados de la competencia de un dataset disponible en otra fuente.

Tomando como base ese artículo, generamos un notebook que toma por entrada un posible submit y ese dataset. El notebook calcula un submit perfecto y lo compara con el nuestro, arrojando el resultado esperado en Kaggle. Este notebook se encuentra en la carpeta 'tools' del repositorio, bajo el nombre:

`resultado_kaggle.ipynb`

Esto nos ayudó a sortear el límite de 5 submits por 24h de Kaggle, ya que hubo varias instancias, por ejemplo cuando hicimos grid search para el tuneo de hiperparametros, en

[13] https://colab.research.google.com/drive/1bcf_jgWjzx2OlRP4m8rIEn-UgQe51pV2?usp=sharing#scrollTo=DOTP3-6aBwhS

[14] https://www.kaggle.com/aaroha33/a-real-disaster-leaked-label/data

donde generamos muchos archivos CSV con submits, sobrepasando dicho valor. Con la ayuda de esta herramienta, pudimos ver el resultado sin tener que subirlo a Kaggle. Aclaramos que este recurso fue usado solamente cuando no teníamos submits disponibles y necesitábamos tomar alguna decisión.

Un ejemplo de salida de este notebook es:

```
Cantidad de tweets predecidos correctamente: 2729
Cantidad de tweets en total para test: 3263
El resultado es: 0.8363469200122586
```

Entendemos que nuestra cantidad de submits en la competencia bajó por el uso de esta herramienta, ya que no fue necesario subir todos los resultados a Kaggle, pero justificamos su uso en períodos de alta actividad y pruebas, ya que nos permitió en las situaciones especificadas conocer el resultado de una optimización dada, por ejemplo, cuando de otra forma tuvieramos que hacer esperado.

6 - Conclusiones

Durante el desarrollo de este trabajo, se utilizaron distintas herramientas de machine learning adquiridas durante la cursada, probando diferentes algoritmos a fin de ir mejorando el *score* de predicción en la competencia.

Dividimos el planteo de nuestra investigación en 2, arboles y redes: empezamos con la primer alternativa, donde pudimos obtener un primer score y empezar a crecer a partir de ahí, obteniendo diferentes resultados con ambas soluciones.

Respecto a los árboles, si bien fueron las primeras predicciones, nos sirvieron para poner un piso (a modo de *baseline*) y saber desde donde teníamos que empezar, no queríamos dejar de hacerlo, para probar la mayor cantidad de alternativas posibles, aunque supiéramos que trabajar solamente con features numéricos no era lo óptimo para el desarrollo de esta competencia.

Las soluciones basadas en redes, por el contrario, nos permitieron mejorar el *score* logrado con las técnicas descriptas en el párrafo anterior. Armamos un preprocesador de textos, configurable, a fin de ensayar diversas alternativas con los datos de entrada, y probamos 3 alternativas: un modelo secuencial con una capa convolucional, un modelo funcional con 3 capas convolucionales, y por último BERT, un algoritmo que representa el estado del arte en problemas de NLP. Aplicamos con cada uno de ellos diversas técnicas de optimización, como grid search para la optimización de hiper-parámetros y k-fold cross validation. Si bien estas técnicas nos permitieron mejorar el puntaje obtenido, pudimos verificar que al mejorar los *features*/entrada las ganancias eran por lo general más importantes, lo que destaca la importancia del *feature engineering*. Para cerrar, es importante mencionar que mediante el uso del algoritmo BERT logramos nuestro mejor *score* en la competencia.

Finalmente, a modo de cierre, nos gustaría agregar que tuvimos la oportunidad de probar un número interesante de algoritmos, cada uno de ellos de diferentes características, que nos permitieron corroborar las ventajas de usar soluciones basadas en redes neuronales a

la hora de trabajar con problemas de NLP. Asimismo, destacamos que el haber participado en la competencia nos dio la experiencia de investigar y trabajar con un algoritmo como BERT, que representa el estado del arte en su campo de aplicación.