Trabajo Práctico 2

[7506/9558] Organización de Datos
Primer cuatrimestre de 2020

Unidos por los datos

Alumno	Número de Padrón	Email
Minino, Nahuel	99599	nahuel_minino@hotmail.com
Peña, Alejandro	98529	alee.pena.94@gmail.com

Repositorio GitHub: https://github.com/alepenaa94/TP2-2020

Índice

1. Introducción

El objetivo de este Trabajo Práctico fue intentar predecir la veracidad de un conjunto de tweets sobre desastres en base al texto del mismo, la ubicación desde donde se realizó el tweet y una palabra clave asociada.

Dentro de los problemas de Machine Learning, este se trataba de un problema de aprendizaje supervisado que podía ser abordado tanto como un problema de clasificación, teniendo las clases 0 (el tweet es sobre un desastre falso) y 1 (el tweet es sobre un desastre verdadero); así como un problema de regresión, obteniendo valores entre 0 y 1 indicando "que tan probable es para el modelo que el tweet sea verdadero". En este trabajo, se abordo el problema de las 2 formas dependiendo del algoritmo con el que se realizó la predicción.

Al igual que en el primer Trabajo Práctico, los tweets sobre los que se trabajó fueron los aportados por la competencia de Kaggle: https://www.kaggle.com/c/nlp-getting-started/overview/description, en la cual se propone analizar tweets para poder monitorear emergencias en tiempo real. Debido a esto, se utilizaron los conocimientos adquiridos en dicho TP a la hora de limpiar la información y generar los features para obtener mejores resultados.

Para realizar las predicciones se utilizaron distintos algoritmos de Machine Learning, principalmente de las librerías sklearn y keras de Python, con los cuales se realizaron predicciones en base a un entrenamiento sobre la información, y una posterior validación de dichas predicciones. Además, se utilizó como herramienta Python3 con las librerías de Pandas y Numpy, entre otras. Luego de realizar las predicciones, se hicieron submits en la competencia para verificar la precisión de las mismas.

El trabajo fue dividido en 2 partes:

- Limpieza de los datos y creación de los features y los dataframes a utilizar.
- Entrenamiento de los algoritmos y validación de las predicciones.

Cada una de estas partes fue realizada en un Notebook distinto (llamados Features y Algoritmos).

Se utilizó un repositorio en GitHub (https://github.com/alepenaa94/TP2-2020) como herramienta de integración, donde se pueden encontrar los notebooks utilizados para la creación de los features y para la propia predicción con los algoritmos.

2. Limpieza de los datos

Al leer los archivos csv con los tweets, especificamos los tipos de datos que contenían los campos *ID* y *TARGET* para reducir el espacio que ocupaba la información del set. Convertimos el campo *ID* al tipo entero sin signo de 16 bits (ya que el id comprendía valores del 0 al 10873) y *TARGET* al tipo booleano (ya que es el campo que indica si el tweet es verdadero o falso). Luego de leer los archivos de train y test, lo primero que se hizo fue concatenarlos para unificar la limpieza y trabajar sobre todo el universo de tweets cuando se crean los distintos features.

Utilizando lo aprendido en el TP1, se realizaron las mismas modificaciones a los campos. Para el campo de "keyword", reemplazamos en donde aparecía " %20" (que representa al espacio en código URL) por el espacio para una mayor claridad en la información. Además, se se completaron los valores nulos con la palabra "None" para considerarlo como otra categoría en los análisis de este campo. Para el campo "location", se realizó un tratamiento similar completando los valores nulos con "Unknown".

Además, tanto para el cálculo del TF-IDF como la creación de los embeddings, se realizó un pre-procesamiento básico en el texto de los tweets. Se creó un campo temporal llamado "clean_text" el cual consistía de una versión limpia del texto del tweet (eliminando links, emojis y puntuación por ejemplo) para dejar únicamente las palabras. Esto demostró mejorar ligeramente los resultados comparado con el manejo de estos campos sobre el texto original de los tweets.

3. Features

3.1. Creación de features

En primer lugar, comenzamos creando features basados en el análisis realizado en el TP1 en los casos donde consideramos que podría aportar información importante para la determinación de la veracidad del tweet.

En un principio, comenzamos con unos pocos features sencillos para poder comprobar el correcto funcionamiento del algoritmo y familiarizarnos con los submits de la competencia. A medida que los fuimos desarrollando, buscamos alternativas en los algoritmos en los que los probábamos.

Algunos de estos features contenían valores numéricos, pero en cambio otros eran categóricos. Debido a esto, en los casos en los que había valores categóricos, se optó por utilizar el método Smoothing como método de conversión de variables categóricas a numéricas. Se eligió este método para evitar crear múltiples columnas por cada feature categórico, a la vez que podemos transformar a valores numéricos no arbitrarios para evitar que el modelo realice transformaciones lineales entre las categorías sin sentido o asigne un orden sin sentido.

Para los features con valores booleanos, se completó con 0 y 1 para evitar tener problemas con los algoritmos que solo aceptan valores numéricos en los features.

3.2. Features elaborados

3.2.1. Cantidad de palabras en el tweet

Campo generado para el TP1, se cuentan las palabras como la longitud del vector generado por la función split.

3.2.2. Longitud del tweet

Campo generado para el TP1 y que habíamos visto en el Finger que cuanto más largo fuera el tweet, más probable era que fuera verdadero.

3.2.3. Longitud promedio por palabra

División de la longitud del tweet por la cantidad de palabras del mismo.

3.2.4. Cuartiles de longitud

En base a las estadísticas sobre la longitud de todos los tweets, los dividimos en 4 categorías dependiendo si están por debajo del cuartil inferior de las longitudes, entre el cuartil inferior y la media, entre la media y el cuartil superior o por encima del cuartil superior.

Como es una variable categórica, usamos el método de Smoothing para transformar los valores a numéricos haciendo uso del label de los tweets del set de entrenamiento evitando el filtrado del mismo.

3.2.5. Cantidad de menciones

Indica la cantidad de menciones realizadas en el tweet, considerando como mención el caso en el que aparece el caracter “@” en el texto.

3.2.6. Cantidad de hashtags

Variable numérica que indica la cantidad de hashtags considerando un hashtag por cada caracter “#” que aparece en el texto.

3.2.7. Cantidad de links

Consideramos un link cada vez que aparecen los caracteres “http”.

3.2.8. Cantidad de caracteres no alfanuméricos

Contamos la cantidad de caracteres no alfanumericos usando el método “punctuation” de string.

3.2.9. Ubicaciones no alfanumericas

Variable booleana que, haciendo uso de la función “isalnum”, indica si la ubicación tiene caracteres de puntuación o no.

3.2.10. Location único

Variable booleana que indica si desde la ubicación en la cual se realizó el tweet se realizaron otros tweets del conjunto analizado o solo el tweet en cuestión.

3.2.11. Encoding del campo keyword

Para el caso del campo "keyword", cómo habían 222 keywords distintas (contando la categoría "None" creada en los casos donde hubiera keyword nulo) se utilizó el método de Smoothing para codificarlas en base al valor de sus labels en el set de train.

Como se verá en la sección 3.3, este resultó ser el feature con mayor importancia de entre los analizados en el algoritmo de Random Forest.

3.2.12. TF-IDF

Variable que representa la similitud entre el tweet vs todo los documentos del set train. Previo a calcular la frecuencia e importancia de cada termino se utilizó algo realizado en el práctico anterior, donde se realiza un "pre-procesamiento" del texto del tweet.

La etapa de "pre-procesamiento" consiste en tokenizar las palabras, aplicar filtros como por ejemplo de urls y cacracteres especiales, realizar una lematizacion en el cual se neutraliza la palabra quitando conjugación genero y/o pluralidad , y por último un filtrado de "stop words" (articulos, preposiciones, pronombres, etc). De esta forma se consigue que los textos de los tweets esten "neutralizados" de forma que en todos los tweets se tengan lematizacion de la palabra y sin stop-words que generan un alto ruido en nuestro calculo de TF-IDF.

Una vez concluida esta etapa se continuó con el armado de la matriz TF-IDF con el uso de la libreria sklearn , puntualmente de "TfidfVectorizer". En la creacion de esta matriz se especificaron parametros que mejoran nuestro objetivo de buscar la similaridad entre los textos de 2 tweets, algunos de los más importantes son "min_df" y "max_features". El primero de estos nos sirvió para ignorar palabras que aparezcan en N o menos documentos, en nuestro caso se usó 5. Por último "max_features" es para indicar la cantidad maxima de palabras en el vocabulario a considerar, siempre considerando el top_n donde n es el valor indicado por parametro, esto nos da la posibilidad en enfocarnos en las n palabras más consideradas importantes por TF-IDF.

La última etapa en la cual realmente se crea el feature es donde por cada tweet se le calcula su correspondiente valor de TF-IDF segun el vocabulario de la matriz generada, dejandonos así un valor numérico que representa la similitud entre tweets, entre más cercano sea el valor de esta columna para 2 o más tweets significa que son más similares.

3.2.13. Embeddings

Usamos los embeddings pre-entrenados de Glove. Como se mencionó previamente, tanto para TF-IDF como para los embeddings, se realizó un pre-procesamiento del campo de texto para trabajar únicamente sobre las palabras que aparecen en el mismo.

La creación de los embeddings estuvo dividida en 2 partes:

- Creación de paddings de los textos de los tweets.
- Creación de la matriz de embeddings.

Para la primer parte, comenzamos creando una lista de listas de palabras por cada tweet con los que contamos. Para esto, usamos la función "word_tokenize" de la librería nltk para separar las palabras del tweet considerando únicamente las palabras con caracteres alfabéticos y excluyendo las "stopwords" (consideramos las "stopwords" en inglés de la librería nltk). A las palabras obtenidas las convertimos a minúscula.

Luego, usando la clase Tokenizer de la librería keras, pasamos la lista de palabras por tweet a listas de números enteros de igual longitud (en este caso, listas de 50 dimensiones), completando con 0 las listas con menos de 50 enteros y truncando las listas con más de 50 enteros.

Para la segunda parte, usamos el archivo de Glove con los embeddings en 100 dimensiones. Por cada palabra en el universo de palabras de los tweets, obtenemos su embedding generando una matriz de "cantidad de palabras" x 100 dimensiones. En los casos en donde no se encuentra la palabra entre los embeddings de Glove, se completa con ceros.

Una vez generados los paddings y la matriz de embeddings, los guardamos en archivos .csv para utilizarlos en las redes neuronales del Notebook de Algoritmos.

3.3. Feature Importance

Para determinar el nivel de importancia de cada uno de los features creados, utilizamos los algoritmos de Random Forest y XGBoost. Sabiendo que se realizan los split que dan mayor ganancia de información primero, estos algoritmos

pueden dar un buen indicio de que features pueden aportar más información al entrenamiento de los algoritmos y que features no aportan mucho o hasta generan ruido y empeoran el entrenamiento.

En los siguientes gráficos podemos ver la importancia de los features según cada algoritmo:

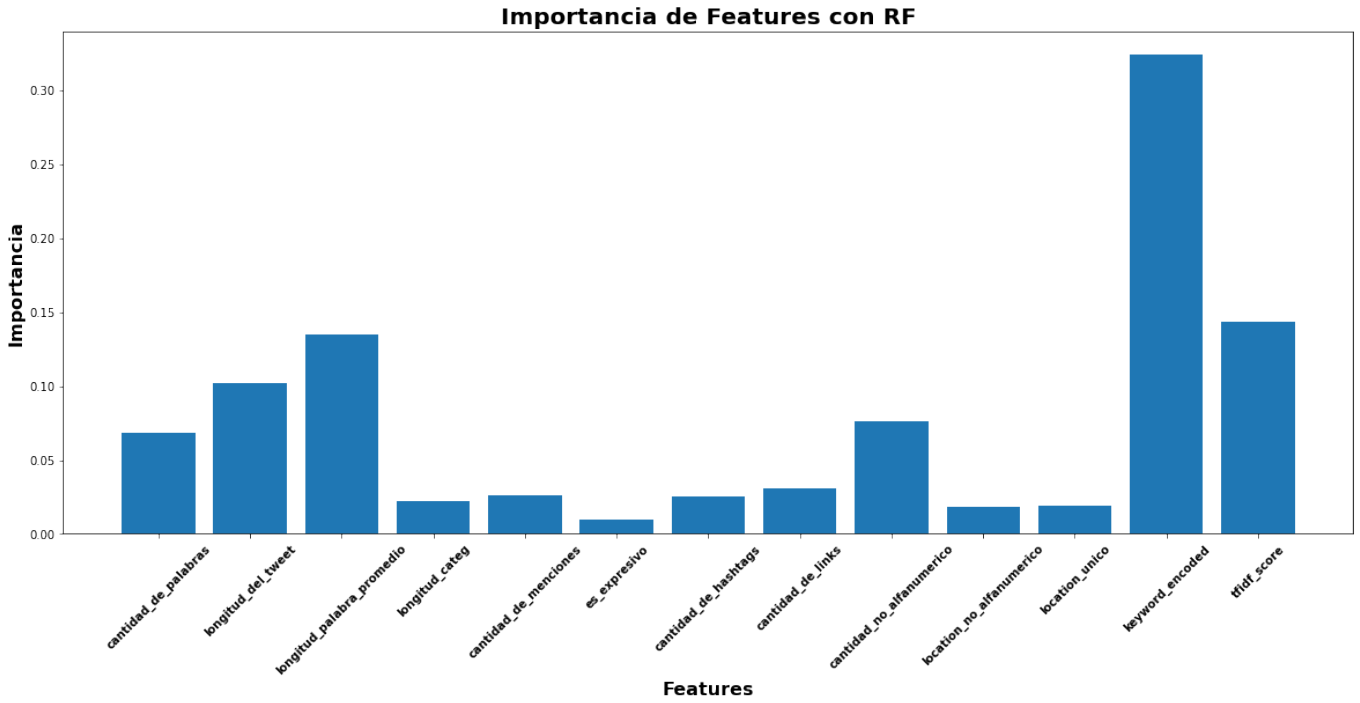

Figura 1: Feature importance obtenido usando Random Forest.

Para el caso de Random Forest, vemos que el campo más importante resulta ser "keyword_encoded" con casi el doble de importancia que el segundo campo más importante. A este le sigue "tfidf_score", "longitud_palabra_promedio", "longitud_del_tweet", "cantidad_de_palabras" y "cantidad_no_alfanumerico". Los demás features parecen tener mucha menos importancia.

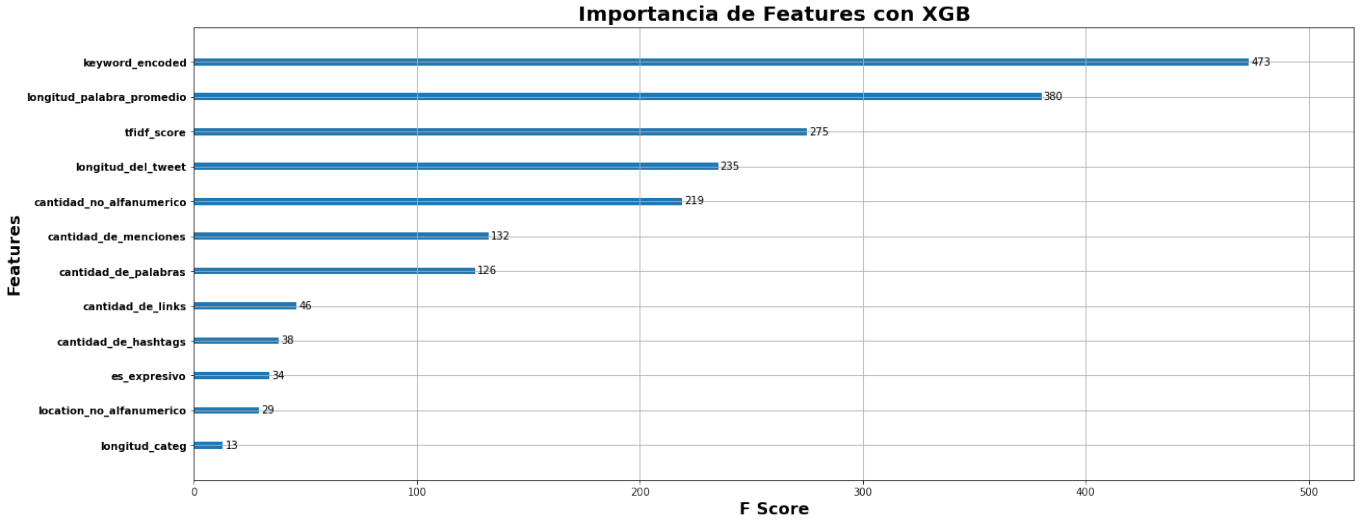

Figura 2: Feature importance obtenido usando XGBoost.

Para el caso de XGBoost, podemos observar que los features más importantes son los mismos que para Random Forest variando un poco las posiciones en las que aparecen, y con la inclusión de "cantidad_de_menciones". A su vez, se puede ver que los demás features tienen una importancia similar entre ellos pero mucho menor a la de las más importantes en ambos modelos.

Vamos a tener en cuenta esto a la hora de probar distintas variaciones de los algoritmos para identificar si los mismos mejoran usando solo features con importancia alta.

4. Algoritmos probados

En esta sección vamos a detallar los distintos algoritmos con los que generamos predicciones indicando los resultados obtenidos y las variaciones en los modelos.

Para los entrenamientos utilizamos la función "train_test_split" de la librería sklearn para separar los tweets del archivo de train en un set de train y un set de test, tomando el 20 % de los registros como set de test. También se utilizaron las funciones "classification_report" y "accuracy_score" para analizar la precisión de las predicciones.

Para los casos en los que fuera necesario, además, utilizamos la clase "StandardScaler" de la misma librería para normalizar los valores de los features antes de realizar el entrenamiento y la predicción.

4.1. Random Forest

El primer algoritmo de Machine Learning probado fue Random Forest. La idea, como se mencionó anteriormente, fue comenzar con un algoritmo relativamente sencillo para generar una predicción con features básicos para poder con el mismo predecir y generar un submit que sea válido para poder familiarizarnos con la competencia. Además, nos permite analizar la importancia de cada feature.

Para este modelo se utilizó la clase "RandomForestClassifier" de la librería de sklearn.

Este algoritmo junto con XGBoost, fue el que peores resultados nos dio, obteniendo un score de alrededor de 0.74-0.75. En un principio, se intentó realizar tuning de los hiperparámetros para ver si esto mejoraba los resultados de manera notable, pero estas siguieron dando scores similares.

Debido a esto, decidimos dejarlo únicamente como algoritmo para analizar la importancia de los features creados.

4.2. XGBoost

Para este modelo se utilizó la clase "XGBClassifier" de la librería xgboost.

Al igual que con Random Forest, los resultados de las predicciones eran bastante malos en comparación a los otros algoritmos. Por lo tanto, nuevamente lo dejamos como algoritmo para analizar la importancia de los features y no nos tomamos la molestia de modificar los hiperparámetros del mismo para ver si mejoraban los resultados.

4.3. Perceptrón Multicapa

Para este modelo se utilizó la clase "MLPClassifier" de la librería sklearn.

Este algoritmo fue una primera aproximación a redes neuronales. Para realizar las predicciones, tanto en esta como en las demás redes neuronales, tuvimos que normalizar los features antes del entrenamiento y la predicción.

Los resultados de las predicciones en todos los casos fueron mejores a los de los algoritmos anteriores por lo que se mantuvo como modelo principal durante las primeras predicciones. También se realizó tuning de hiperparámetros sobre este modelo sin grandes mejoras respecto a los hiperparámetros originales.

A pesar de todo esto, los submits no superaban el score del 0.76 por lo que terminamos buscando otro modelo de redes neuronales más avanzado.

4.4. Redes Neuronales con Keras

Luego de ver los buenos resultados que daba Perceptrón Multicapa, decidimos buscar redes neuronales que se ajustaran mejor al problema. Debido a esto, usamos la librería keras de python para crear redes neuronales especificando las distintas capas y pudiendo hacer uso de los embeddings creados.

Estas redes son las que nos dieron mejores resultados hasta el momento con gran diferencia (dandonos en los peores casos un score superior a 0.79) y sobre las que más centramos nuestro análisis y nuestras modificaciones.

Para estos modelos se trabajó como un problema de regresión, obteniendo valores en el intervalo [0,1] y clasificando: los que dieran un resultado entre 0 y 0.5 como 0 (el tweet es falso), y los que dieran valor mayor a 0.5 como 1 (el tweet es verdadero).

A grandes rasgos usamos 2 modelos:

- Un modelo con un único input de embeddings.
- Un modelo con 2 inputs, por un lado los embeddings y por otro los features.

4.4.1. Modelo 1: solo embeddings

Para hacer uso de los embeddings creados, utilizamos la clase "Sequential" de keras, con la cual se puede ir agregando las distintas capas de la red en un orden secuencial.

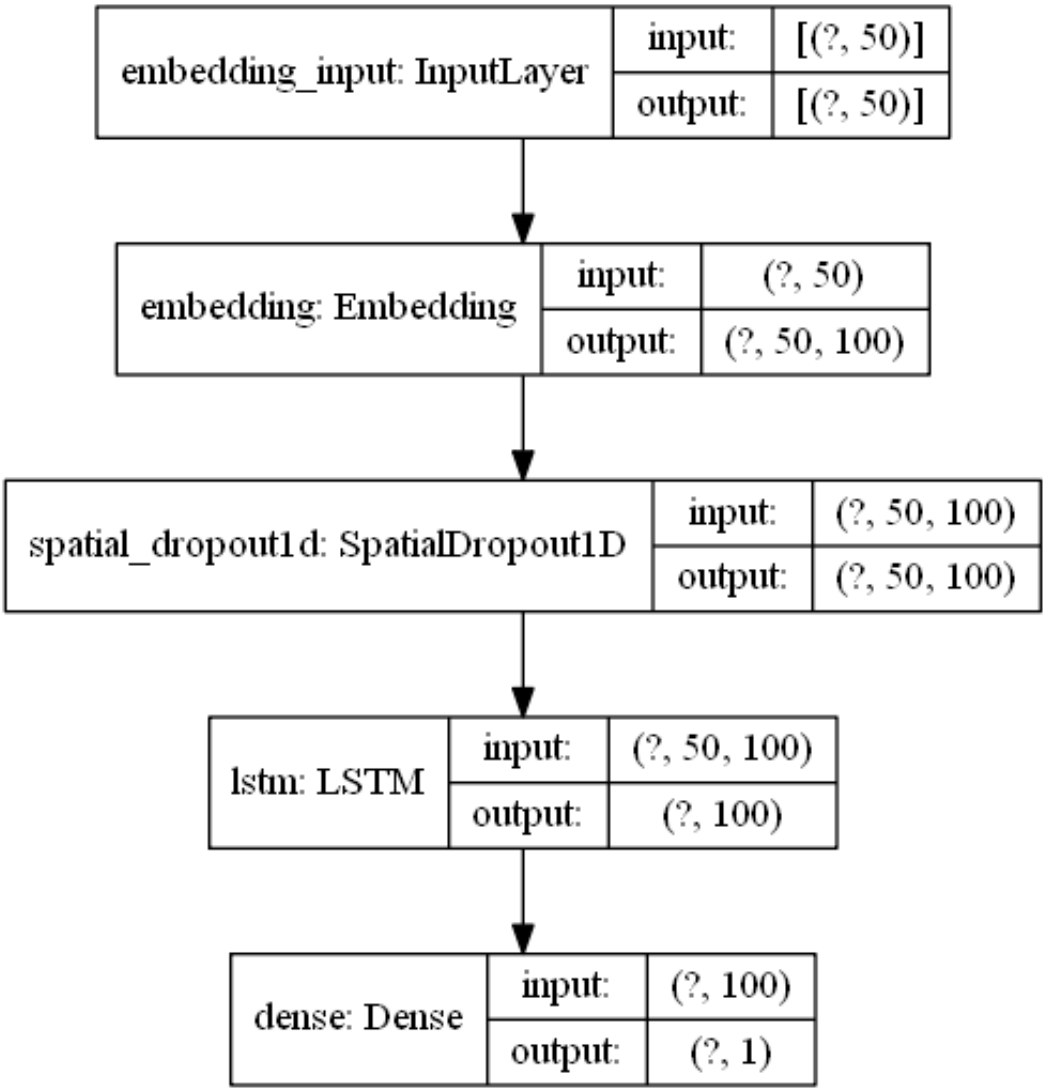

Figura 3: Diagrama del modelo 1.

La primer capa de este modelo recibe los paddings de los tweets creados previamente como inputs, los cuales entran en una capa de embeddings la cual hace uso de la matriz de embeddings creada. Luego, hay una capa de dropout en donde se apagan algunas neuronas al azar en cada iteración para así evitar el overfitting, seguida por una capa LSTM. Finalmente, termina con una capa fully-connected que usa como función de activación la sigmoide para obtener valores en el rango [0,1].

A lo largo de los submits se realizaron algunas modificaciones a esta estructura, como por ejemplo a sus parámetros, se probaron capas Bidirectional-LSTM, CONV1D, entre otras cosas; siendo este modelo el que mejores resultados mostró.

Una vez que vimos lo mucho que mejoraban los resultados, buscamos la forma de implementar además de los embeddings, los features que usamos para el resto de los algoritmos, cosa que no se podía hacer con la clase "Sequential".

4.4.2. Modelo 2: embeddings + features

Debido a que la clase "Sequential" no permitía el uso de múltiples Inputs, utilizamos la clase "Model" de keras, con la cual se puede especificar las capas a usar por cada input y concatenar los inputs para usarlos en conjunto.

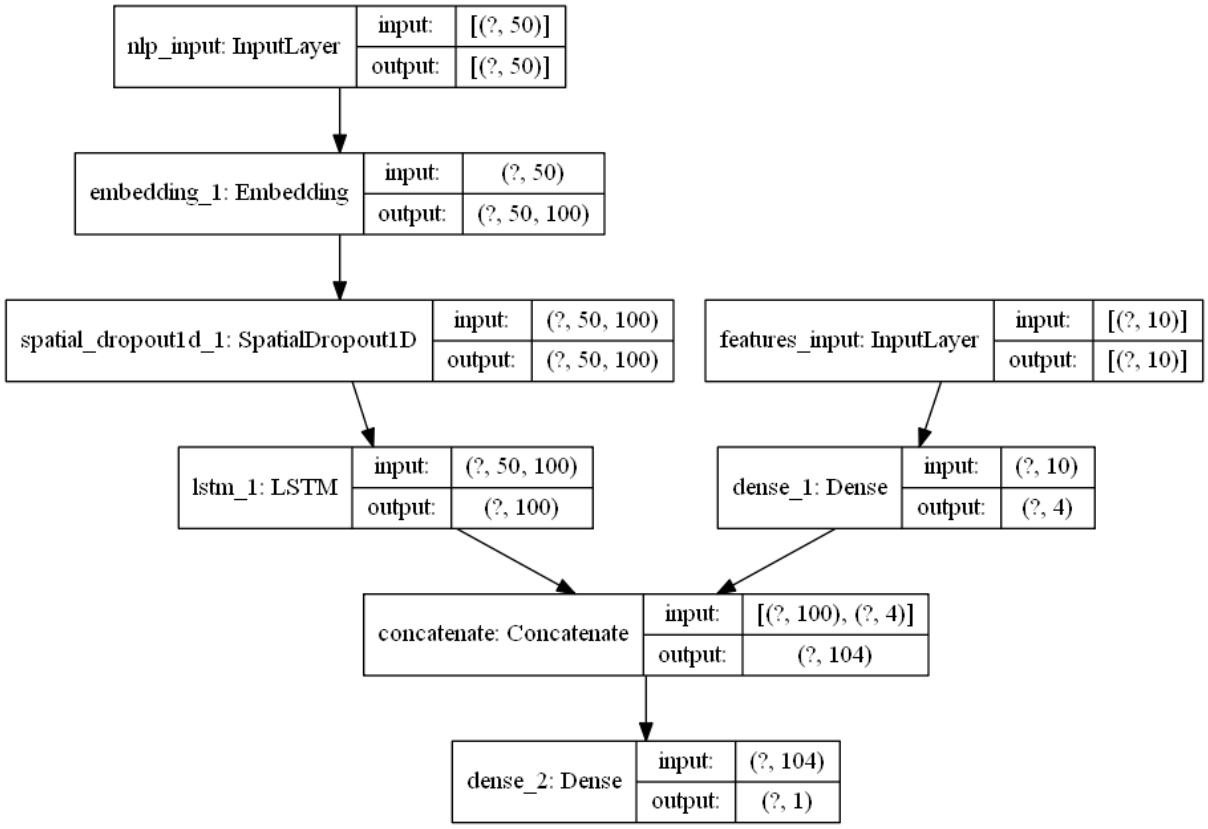

Figura 4: Diagrama del modelo 2.

Para la rama de inputs de los embeddings, decidimos usar la misma estructura que para el modelo 1 hasta la caoa LSTM. Para la rama de los otros features, únicamente utilizamos una capa dense con función de activación relu; cuyo output son 4 neuronas las cuales se concatenan con las 100 neuronas del output de la capa LSTM. Finalmente, se realiza otra capa fully-connected con la función de activación sigmoide para obtener la predicción.

Al igual que con el modelo 1, se probaron múltiples alternativas para esta estructura siendo este modelo el que mejores resultados dio. El cambio más significativo en ambos modelos fue la modificación del "Learning Rate" del optimizador Adam. Originalmente usabamos un learning rate de 1e-5, pero al cambiarlo a 3e-4 los resultados mejoraron notablemente, pasando de scores en el rango 0.79-0.80 a 0.81-0.82.

Otro cambio interesante, fue que originalmente usabamos un "batch_size" igual a 4 y eventualmente lo cambiamos a 32. Este cambio mejoró los resultados muy ligeramente, pero lo más importante fue que se pudo pasar de un entrenamiento de 20 minutos aproximadamente a uno de 5 minutos, permitiendonos comenzar a probar ensambles como se hablará más adelante.

Algo a destacar, es que este modelo siempre dio resultados un poco mejores que el modelo que utiliza únicamente los embeddings. Además, se probó este modelo considerando menos features (dándole prioridad a las que nos dieron mayor feature importance), pero los mejores resultados se obtuvieron usando todos los features creados.

Otra modificación realizada fue que, usando la clase "Checkpoint" de keras, guardamos la iteración que generar el menor "val_loss". De esta forma, podemos ejecutar entrenamientos con múltiples epochs sin preocuparse por el overfitting del modelo ya que se guardarán los pesos óptimos.

Analizando los resultados que fuimos obteniendo de las predicciones, notamos que estas variaban bastante, es decir, que variaba la clase predecida en diversos ids. Debido a esto, pensamos probar realizar un Majority Voting de las predicciones del modelo. De este ensamble resultaron los mejores scores que obtuvimos hasta el momento en la competencia.

5. Optimización de Hiper parámetros

La optimización de hiper-parámetros consiste en la busqueda de los hiper-parámetros que generan el mejor score para un modelo. Se tiene que establecer un conjunto de valores por cada hiper-parámetro del modelo para que se busque con que combinacion es la que mejor score genera el modelo. En nuestro caso, se analizó principalmente con los siguientes metodos de optimización.

5.1. Grid Search

Este método consiste en pruebas a "fuerza bruta" probando cada combinación para los conjuntos de valores pre-establecidos, en nuestro caso partimos primero con pocos parametros y se fueron agregando más y generando mayor rango de testeo. Cabe destacar que este método si bien encuentra la mejor combinación segun los datos proporcionados, es un método poco práctico y nada eficaz ya que prueba absolutamente **toda** combinación, lo que consume mucho tiempo.

En nuestro caso práctico no se hizo la búsqueda completa de todos los hiper-parámetros deseados sino que se hizo con algunos acotados para familiarizarnos con la optimizacion de los hiper-parámetros y profundizar más con Random Search.

5.2. Random Search

En este caso, este método soluciona la problemática de Grid Search, acotando la búsqueda de los conjuntos a "k" combinaciones aleatoriamente, de esta forma no probamos todo pero lo importante es que reducimos tiempo.

Tanto para Grid Serch como para Random Search hicimos uso de la libreria "sklearn model_selection " "GridSearchCV" y "RandomizedSearchCV", en ambos casos se creo una funcion en la cual se creaba el modelo a usar haciendo esta funcion parametrizable con los hiper-parámetros del modelo. La idea es generar el modelo con "KerasClassifier" pasando como creacion esta misma función y así usar Random Search con estimador el modelo creado y pasando el conjunto de hiper-parámetros y su conjunto de valores por medio de un diccionario a ser entrenado.

Los hiper-parámetros y sus conjuntos de valores con los que se hicieron pruebas para el modelo de RN con embeddings son:

- batch_size : [32,64]
- epochs : [32,64]
- lstm_p1 : [64,100]
- dropout_rate : [0.0, 0.1, 0.2, 0.3]
- activation : ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid']

Y se obtuvo como mejor combinación:

- 'lstm_p1': 64,
- 'epochs': 64,
- 'dropout_rate': 0.2,
- 'batch_size': 64,
- 'activation': 'tanh'

Tanto la implementación de Grid Search como de Random Search tienen métodos para obtener el mejor estimador según el score obtenido, consultar cual fue el mejor score y con que hiper-parámetros se logró.

Una vez que obtenemos la mejor combinación se podría realizar más pruebas con valores mas cercanos a los que fueron probados si es necesario o se busca aun una mejor optimización.

Con estos hiper-parámetros se mejoro el "Test accuracy" pero no superó lo obtenido para el modelo con embeddings+features. Vale hacer la aclaración que no se pudo hacer la optimización de "embeddings+features" debido a que nos encontramos con un problema con la libreria sklearn por tener el set de datos dividido los embeddings y features por otro lado y no pudimos resolverlo, debido a eso es que la optimizacion se realizó para el modelo 1 de Redes Neuronales comentado anteriormente.