Organización de Datos - TP2

August 10, 2020

Apellido, Nombres	Padrón	E-mail
Tomas Fanciotti	102179	tfanciotti@fi.uba.ar
Gomez hobbs Fernando	94811	fgomezh@fi.uba.ar
Leguizamón Agustín	99535	aleguizamon@fi.uba.ar
Sosa Cristian Martín	98741	csosa@fi.uba.ar

Link al repositorio de GitHub:
`https://github.com/AgustinLeguizamon/datos_tp_2`

Contents

1 Introducción

Links utilidades utilizadas en los notebooks:

`https://nlp.stanford.edu/projects/glove/`

`https://gist.githubusercontent.com/Sirsirious/ c70400176a4532899a483e06d72cf99e/raw/ e46fa7620c4f378f5bf39608b45cddad7ff447a4 /english_contractions.json`

1.1 Objetivos

El objetivo principal de este trabajo práctico es usar los conocimientos estudiados en la materia sobre procesamiento de datos y generar un modelo de Machine Learning (ML) que se entrene usando dichos datos para encontrar relaciones entre las variables que se disponen (features) y el target (label) que clasifica cada entrada.

Para esto, participamos en la competencia de ML donde se propone crear un modelo que permita, a partir de un tweet de entrada, determinar si esta basado en un desastre real o si habla de cualquier otra cosa. Dicho modelo será calificado por la cátedra y por la plataforma de la competencia que lo pondrá a prueba con un set de datos de prueba obteniendo un "score" como métrica.

La competencia se desarrolla en la plataforma de Kaggle

`https://www.kaggle.com/c/nlp-getting-started.`

1.2 Set de datos

El set de datos dado para el análisis consta de registros que representan cada uno de los tweets, con la siguiente estructura:

- id - identificador único para cada tweet
- text - el texto del tweet
- location - ubicación desde donde fue enviado (podría no estar)
- keyword - un keyword para el tweet (podría faltar)
- target - en train.csv, indica si se trata de un desastre real (1) o no (0)

Utilizamos el train.csv para entrenar nuestros algoritmos y luego el test.csv son aquellos tweets que no poseen un 'target', este set es el utilizado para realizar las predicciones con nuestros algoritmos. Los "targets" predecidos son publicados en la pagina de la competencia, de esta publicación obtenemos un porcentaje de aciertos para saber que tan efectivo es nuestro algoritmo al momento de predecir los desastres.

2 Pre-procesamiento de la data

En el primer trabajo practico (TP1) se hizo un análisis del set de datos para obtener información en cuanto a la relación entre el tweet y si el hecho era real o no. Ahora con esta informacion buscamos procesarla de manera de que pueda ser utilizada por un algoritmo de machine learning. Nos enfocamos en el texto dada que las columnas "location" y "keyword" aportan poco en comparación a la información que se puede extraer del texto.

Los tweets se consideran **unstructured data**, dado que no son valores numéricos sino una tira de caracteres sin ningún tipo de orden en común. Para poder manejar esta información en forma de texto utilizamos en los conceptos de **NLP (Natural Lenguaje Processing)**

2.1 Texto

Los tweets estan escritos por distintos usuarios y al tratarse de una red social los textos son "ruidosos" es decir que poseen todo tipo de faltas de ortografia, exageraciones, abreviaciones, excesivo uso de puntuaciones y mas. Lo que significa que es imposible trabajar con el texto tal como se nos presenta para esto primero el texto debe ser procesado. Existen varias tecnicas de pre-procesamiento de texto listadas a continuación.

- Lowercasing: convertir en minúscula cada palabra
- Stemming: llevar la palabra a su forma raíz (no necesariamente real dado que corta el final de cada palabra esperando llevarlo al formato raiz)
- Lemmatization: a diferencia de Stemming, no corta la palabra si no que intenta llevarlo a su forma raiz.
- Stop-word removal: remoción de palabras comúnmente usadas que no aportan información
- Normalization: general y no representa una técnica en si, sino varias dependiendo de la tarea. Incluye correcciones ortográficas (2moro por tommorrow), expandir contracciones(e.g i'm por i am), simplificar puntuaciones; normalizar espaciados, saltos de linea; etc.
- Noise Removal: Remocion de urls, caracteres especiales, numeros
- Text enrichment /Augmentation: Agregar informacion al texto que previamente no habia.

La utilización de cada técnica depende del tipo de data que tengamos. En este caso al tener una cantidad de data moderada y tratarse de texto ruidoso decidimos hacer un pre-procesamiento sustancial pero sin enriquecimiento del texto. A continuación indicamos las funciones utilizadas

```
#Elimino corchetes, links, <>, puntuaciones, saltos de linea
#y remuevo las palabras que contienen numeros
#remuevo todas las puntuaciones salvo # y @
#no remuevo urls porque se reemplazan mas adelante
def clean_text(text):
def remove_emoji(text):
def _lemmatize_text(sentence, nlp):
def simplify(sentence):
def _replace_urls(sentence)
def simplify_punctuation(sent)
def _normalize_whitespace(sent)
def normalize_contractions(text, contractions):
def spell_correction(text, spellchecker):
def _reduce_exaggerations(text):
def is_numeric(text):
```

Todas estas funciones fueron utilizadas para procesar el texto de manera de remover la mayor cantidad de ruido posible, notar que no se opto por remover los stopwords (sino que fueron "lemmatizados") además se obtuvieron mejores resultados al dejarlos que simplemente removiéndolos como resulta verdad en muchos casos de "Sentiment Analysis" los stopwords juegan un papel importante y removerlos completamente puede reducir la performance. Entonces nuestra función de pre-procesamiento de texto queda definida de la siguiente manera.

```
def text_preprocessing(text, nlp, contractions, spellchecker):
    text = simplify(text)
    text = normalize_contractions(text, contractions)
    text = spell_correction(text, spellchecker)
    #la limpieza la hago al final dado
    #que remueve todas las puntuaciones y muchas son importantes
    #para la normalizacion del texto como (')
    text = clean_text(text)
    text = _lemmatize_text(text, nlp).strip()
    return text
```

Damos un ejemplo de como afecta esto al texto. Texto sin procesar

Texto procesado

```
'-PRON- deed be the reason of this earthquake may allah forgive -PRON- all'
```

3 Algoritmos de Machine Learning

Dado que existen muchas formas de encarar la solución a este problema de **clasificación** (los targets son 0 o 1) dividimos los algoritmos utilizados en varios notebooks en donde cada uno tiene un enfoque distinto para la creación y entrenamiento del algoritmo de predicción. Cada subsección tiene el título del notebook que se encuentra en el repositorio de github

3.1 Pre-process + Classifiers + Stacking

En este notebook utilizamos el pre-procesamiento de texto ya mencionado, luego utlizando la libreria **sklearn** probamos los distintos algoritmos de clasificación (naive bayes, modelos lineales, svm, knn, etc)

3.1.1 Feature extraction

Utilizamos BOW y TF-IDF para extraer features a partir del texto, volviendo la colección de textos en una matriz con la cantidad de apariciones de cada palabra.

```
from sklearn.feature_extraction.text import CountVectorizer,TfidfVectorizer

#Feature extraction - BOW
count_vectorizer = CountVectorizer()
```

```
train_bow = count_vectorizer.fit_transform(df_train['text'])
test_bow = count_vectorizer.transform(df_test['text'])

#TF IDF
tfidf_vectorizer = TfidfVectorizer()
train_tfidf = tfidf_vectorizer.fit_transform(df_train['text'])
test_tfidf = tfidf_vectorizer.transform(df_test['text'])
```

Se probaron los distintos parametros que sklearn nos ofrece:

- strip_accents
- stop_words:'english': remover stopwords del ingles
- tambien se probaron listas de stop_words personalizadas
- ngram_range: Se usaron unigramas, bigramas y trigramas y combinaciones (e.g unigramas y bigramas)
- max_df: ignora terminos que tienen una frecuencia de documento mayor a lo indicado.
- min_df: ignora terminos que tienen una frecuencia de documento menor a lo indicado.

Se probaron varias combinaciones de estos parámetros pero resultó mas eficiente el utilizar los que vienen por defecto dado que dieron mejores resultados.

3.1.2 Modelos

Se utilizaron los siguientes modelos de clasificación:

- LogisticRegression(max_iter=1000, class_weight='balanced', C=1e-1)
- RidgeClassifier()
- SGDClassifier()
- KNeighborsClassifier()
- DecisionTreeClassifier()
- LinearSVC(max_iter = 1500)
- NuSVC()
- MultinomialNB()
- ComplementNB()
- Perceptron()

Utilizando 5-fold cross validation obtenemos una idea del score que podríamos obtener. Esto se utilizo tanto para BOW como para TF-IDF obteniendo los siguientes resultados.

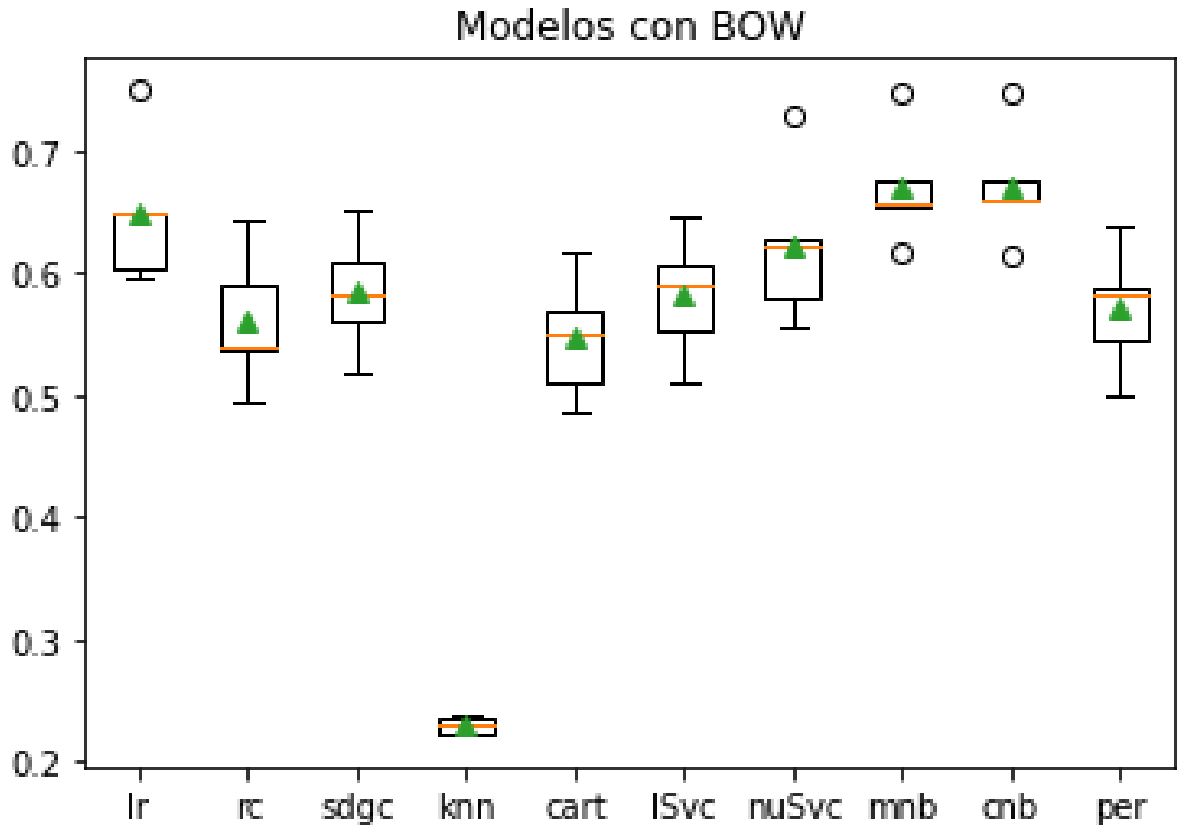

Figure 1: Modelos con BOW

De la figura 1 vemos que la mayoría de los modelos que tienen un mayor puntaje y a su vez una menor desviación estándar son Logistic Regression, Multinomial Naive Bayes, Complement Naive Bayes.

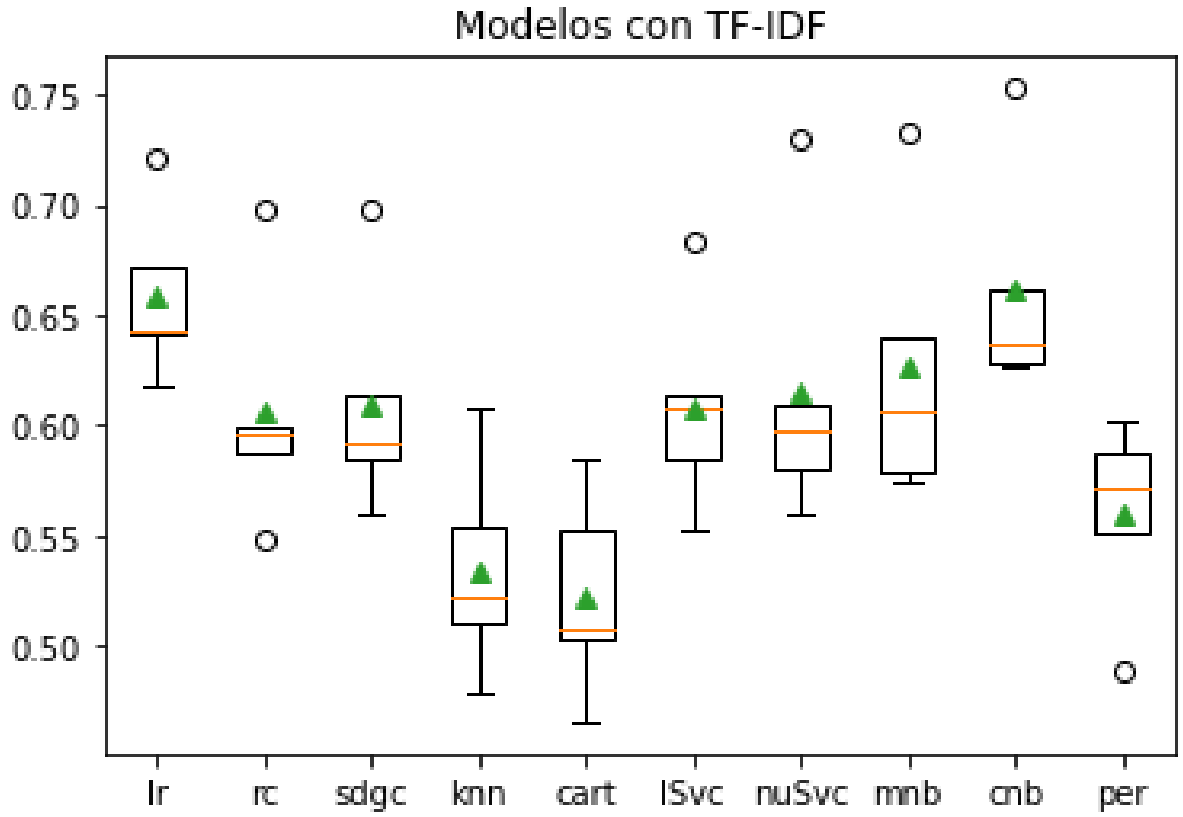

Figure 2: Modelos con TFIDF

De la figura 2 se nota que en la mayoría de los casos el puntaje es menor y la desviación estándar es mayor comparado con los modelos que utilizan BOW. Aun asi los modelos que presentan mejores resultados son nuevamente Logistic Regression, Multinomial Naive Bayes, Complement Naive Bayes y NuSVC.

Para mejorar los resultados se decidio usar StackingClassifier de la libreria de sklearn.ensemble de manera de agrupar las salidas de cada uno de los estimadores y usar un clasificador para computar la predicción final. Elegimos los mejores modelos antes mencionados para crear nuestro Stack.

```
def get_best_stacking():
level_0 = list()
level_0.append(("lr", LogisticRegression(max_iter = 150, class_weight='balanced', C=1e-1)))
level_0.append(("cnb", ComplementNB(alpha=1)))
level_0.append(("mnb", MultinomialNB(alpha=1)))
level_0.append(("nuSvc", NuSVC()))
level_1= LogisticRegression()
model = StackingClassifier(estimators=level_0, final_estimator=level_1)
return model
```

Evaluando este nuevo modelo tanto con BOW como con TFIDF con respecto a los demás obtenemos una ligera mejora en el puntaje, mayor en el caso de BOW y siendo el modelo con el valor mas alto comparando con el Stack con TFIDF y los modelos base. En la figura 3 se agrega el modelo "stacking" de manera de poder comparar con el resto de los modelos base y aquellos no utilizados en el stack de estimadores.

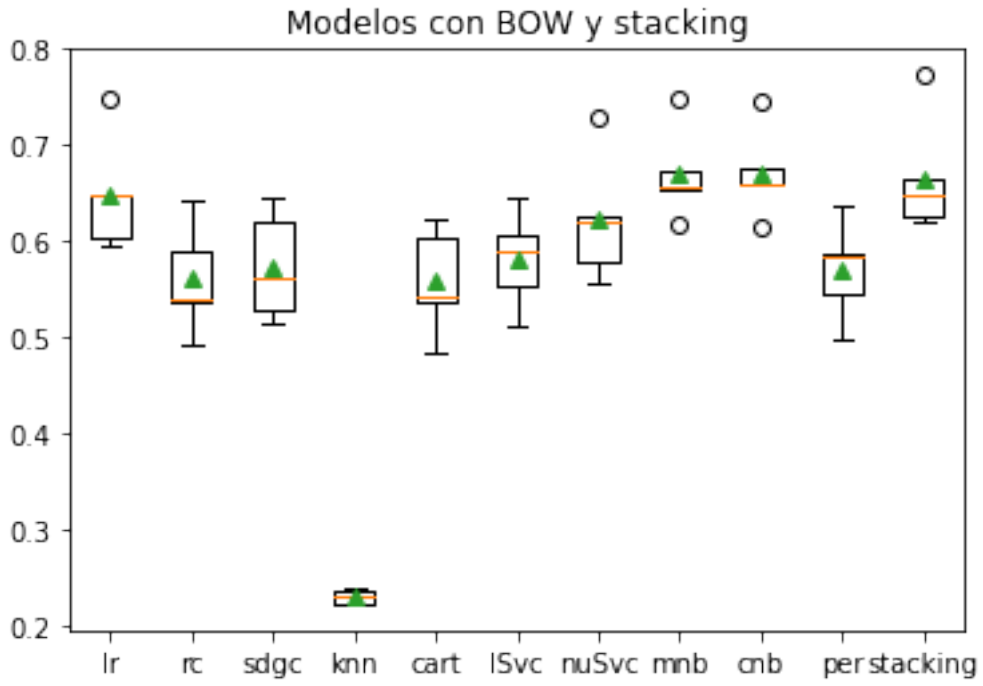

Figure 3: Modelos con BOW y Stacking

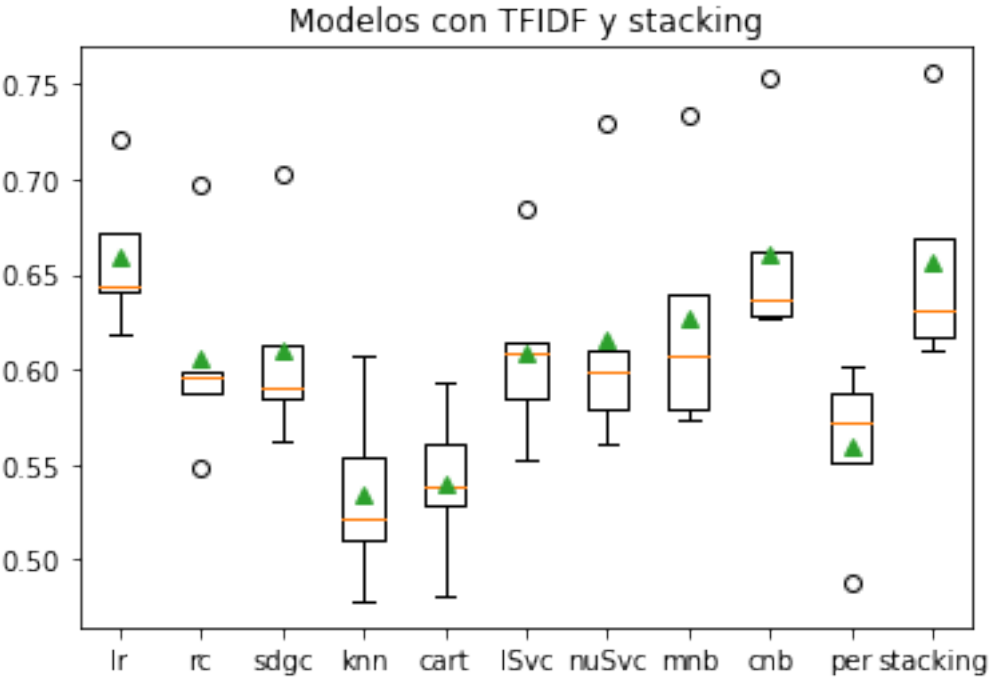

Figure 4: Modelos con TF-IDF y Stacking

3.1.3 Parámetros

Tanto para CountVectorizer() como TfidfVectorizer() se probaron distintos parámetros (mencionados en 3.1.1) para mejorar estos resultados. A pesar de distintas combinaciones se comprobó que los utilizados por defecto tienen el puntaje mas alto.

En el caso del Stack se fue ajustando la regularización (C o alpha) de cada modelo eligiendo aquel que de un mejor resultado. En el modelo LogisticRegression se ajusto el peso de cada clase (1 y 0) con el parámetro class_weight que modifica los pesos de las clases a medida que se va entrenando.

3.1.4 Submit

Con el siguiente score de cross validation, entrenamos el modelo basado en StackingClassifier.

```
[0.6481149  0.6192733  0.66403785 0.62543554 0.77286585]
```

El score de la competencia es: 0.80140

3.1.5 Pre-process + XGBoost

Utilizando el mismo pre-procesamiento de texto pasamos a utilizar el algoritmo de machine learning XGBoost. En un principio trabajamos con solo el texto, extrayendo features numericos a partir de texto con CountVectorizer() y TfidfVectorizer(). Dado que es un problema de clasificación usamos XGBClassifier con los siguientes hiper parámetros.

```
clf = XGBClassifier(objective = 'binary:logistic',
                random_state=42,
                seed=2,
                colsample_bytree=0.5,
                subsample=0.7,
                learning_rate=0.1,
                )
```

Dado que pensamos usar features de distinto tipo (numérico y texto) formamos un Pipeline para que en el caso de que sea texto, se extraigan los features numéricos a partir de las funciones de utilidad; y en el caso de ser un numero simplemente se devuelva ese numero. Una vez formado el Pipeline de features y modelo de clasificación, utilizamos un grid search (sklearn.model_selection.GridSearchCV()) para obtener la cantidad óptima de arboles a usar (n_estimators), los parámetros fueron 50,100 y 300. Haciendo un "train_test_split" separamos el train set en uno de entrenamiento y validación para los cuales se obtuvieron los siguientes puntajes para los estimadores mencionados.

Split de entrenamiento

```
[0.86539216, 0.90980392, 0.97401961]
```

Split de validación

```
[0.74372549, 0.74921569, 0.74509804]
```

Los mejores resultados se obtienen con n_estimators = 100 (0.74921569) Resulta común que en el de entrenamiento se tenga una exactitud mayor que en el de validación, esto nos permite ver si nuestra modelo tiene o no overfitting en el caso de que la diferencia sea mayor, como se ve en el caso que corresponde a 100 y 300. Definiendo la cantidad de arboles a construir y los hiper parámetros de antes entrenamos nuestro modelo de XGB y realizamos las predicciones.

El score de la competencia es: 0.76371

XGboost texto + features numericos Al train set se le agregaron las siguientes columnas para así tener mas features, numéricos, para usar en el algoritmo. Todos estos se calcularon antes de realizar la limpieza de texto ya que la idea es que se tenga en cuenta la diferencia que existen entre los tweets reales y los que no lo son; ya que una vez normalizados pierden sentido como el conteo de puntuaciones.

- word_count
- unique_word_count
- stop_word_count
- url_count
- mean_word_length
- char_count
- punctuation_count
- hashtag_count
- mention_count

Generando un Pipeline con cada feature y usando sklearn.pipeline.FeatureUnion unimos los features basados en texto y numéricos. Se probaron distintas combinaciones de estos features, al final los que mostraban mejoras en el puntaje fueron los siguientes:

- unique_word_count
- stop_word_count
- mean_word_length
- mention_count

Entrenando nuevamente el modelo con features mixtos y los mismos n_estimators del grid search [50, 100, 300].

Split de entrenamiento

```
[0.79598039, 0.82848039, 0.88593137]
```

Split de validacion

```
[0.75117647, 0.76215686, 0.78039216]
```

El mejor resultado lo tenemos con n_estimators=300 y la diferencia con el split de entrenamiento es menor por lo que nos aseguramos de no tener overfitting.

El score de la competencia es: 0.78455

LightGBM + Limpieza Se utilizó este modelo simplemente para probar otro modelo de Gradient Boosting, el cual resultó ser más rápido y sencillo pero no obtuvo mejores resultados que el modelo XGBoost con la misma limpieza del texto y utilizando los mismos vectores de Glove.

El score de la competencia es: 0.80692

XGBOOST + Limpieza + CV

Al efectuar una limpieza simple del texto, eliminando stopwords, tomando los vectores de GloVe (6 billones de palabras y vectores de 200 dimensiones) se utilizaron estos features para el entrenamiento para clasificacion de XGBoost, utilizando a su vez cross validation para el tuneo de hiperparametros.

```
alg = xgb.XGBClassifier(learning_rate=0.1,
                    n_estimators=1000,
                    max_depth=15,
                    min_child_weight=3,
                    gamma=0.2,
                    subsample=0.6,
                    objective='binary:logistic',
                    nthread=4)
xgb_param = alg.get_xgb_params()
cvresult = xgb.cv(xgb_param,
                    xgtrain, num_boost_round=alg.get_params()['n_estimators'], nfold=5,
                    early_stopping_rounds=40)
alg.set_params(n_estimators=cvresult.shape[0])
alg.fit(xtrain_gv, ytrain, eval_metric='auc')
```

El score de la competencia es: 0.81274

3.1.6 Pre-process + neural network

En este notebook utilizamos vectores pre entrenados de GloVe (6 billones de palabras y vectores de 100 dimensiones) y redes neuronales de Keras.

Transformaciones

Utilizando la clase Tokenizer() de keras, vectorizamos el cuerpo del texto, convirtiendo el texto en una secuencia de palabras separadas por espacios y luego estan secuencias se separan en listas de tokens y al final son indexados. Con el método fit_on_texts actualizo el vocabulario basado en una lista de textos (strings). Con el método text_to_sequence transforma cada texto en una secuencia de enteros. Al final usamos el método pad_sequences para transformar la lista de secuencias en una matriz (2D Numpy array) y completar con ceros aquellas secuencias que tengan una longitud menor a la definida.

```
MAX_NUM_WORDS = 25000
MAX_SEQUENCE_LENGTH = 50

tokenizer = Tokenizer(num_words=MAX_NUM_WORDS)
tokenizer.fit_on_texts(df_train['text'])
sequences = tokenizer.texts_to_sequences(df_train['text'])

word_index = tokenizer.word_index

data = pad_sequences(sequences, maxlen = MAX_SEQUENCE_LENGTH)
```

Modelo

```
embedding_layer = Embedding(num_words, EMBEDDING_DIM,
embeddings_initializer=Constant(embedding_matrix),
input_length=MAX_SEQUENCE_LENGTH, trainable=False)
```

```
model = Sequential()
model.add(embedding_layer)
model.add(SpatialDropout1D(0.2))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
```

El score de la competencia es: 0.61906

Para obtener mejores resultados fueron utilizados vectores preentrenados de GloVe, de 200 dimensiones. Adicionalmente fueron utilizados 2 capas LSTM bidireccionales, en conjunción con otras capas fully connected.

```
model.add(Bidirectional(LSTM(700,input_shape=(1,200), return_sequences=True)))
model.add(Dense(1000, ))
model.add(Dropout(0.2))
model.add(Bidirectional(LSTM(700,input_shape=(1,200), return_sequences=True)))
model.add(GlobalMaxPooling1D())
model.add(Dropout(0.5))
model.add(Dense(800))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(650))
model.add(Dense(1, activation='sigmoid'))
```

El score de la competencia es: 0.80692

Observación: También fueron submitteados muchos otros modelos que poseían RNN's (LSTM o GRU) o CNN's o una combinación entre ambas, pero ninguna combinación logró superar éste último valor.

3.2 BERT

En éste caso fue utilizada la librería ktrain la cual ayuda a crear, entrenar y utilizar modelos de machine learning usando Keras. Para éste caso fue utilizado usando el modelo BERT (o Bidirectional Encoder Representations from Transformers)

Con el método lr_find() es posible hallar el learning rate adecuado para la minimización del loss en el entrenamiento.

En base a ésta Figura se decide que un learning rate de 2e-5 es un valor adecuado para realizar el entrenamiento.

Luego el modelo se entrenó con el método fit_onecycle, entrenando con solo una época.

El score de la competencia es: 0.84002

Pero al utilizar el método autofit, que utiliza una "triangular learning rate policy" el cual, durante el entrenamiento, forma un ciclo triangular aumentando y disminuyendo el learning rate hasta su máximo, especificado por el parámetro del método.

Modificando además un par de hiperparámetros como el la cantidad máxima de palabras por Tweet o la cantidad máxima de features a utilizar (el tamaño del vocabulario utilizado). Se consiguió luego de un entrenamiento de 7 épocas el mejor puntaje actual en la competencia.

El score de la competencia es: 0.84094

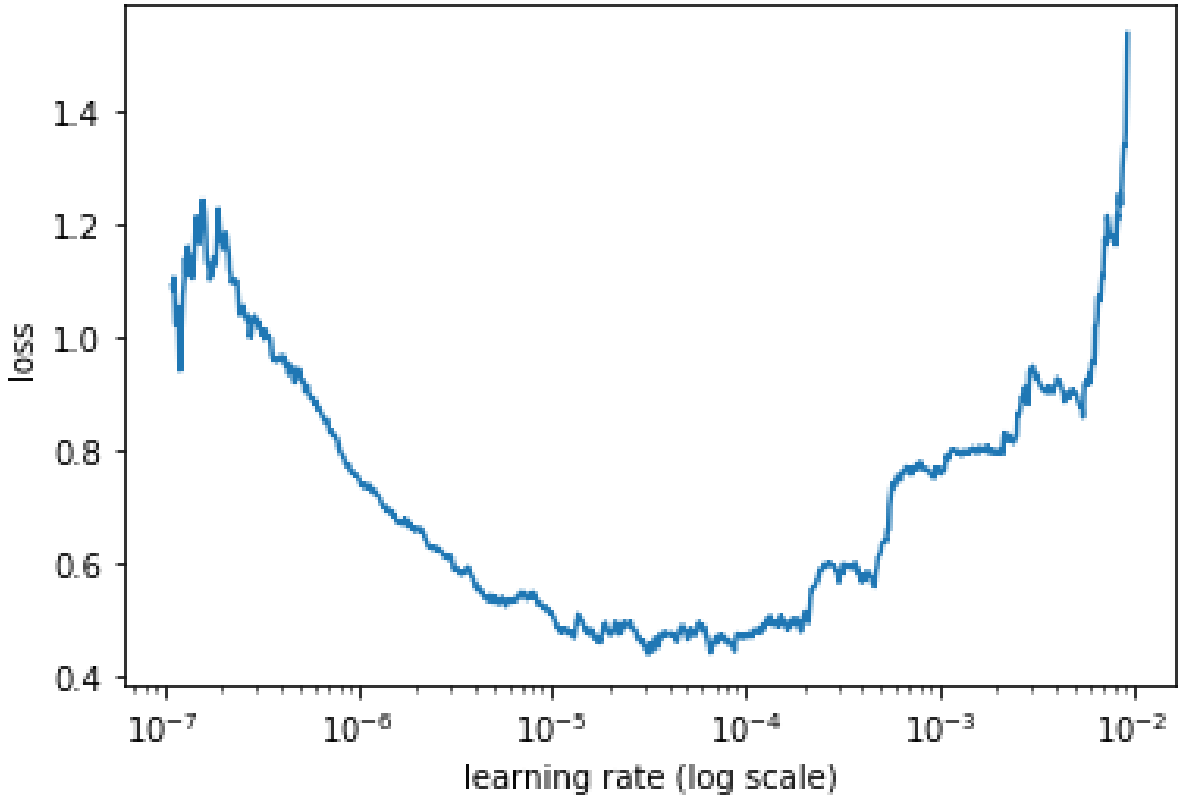

Figure 5: loss vs learning rate

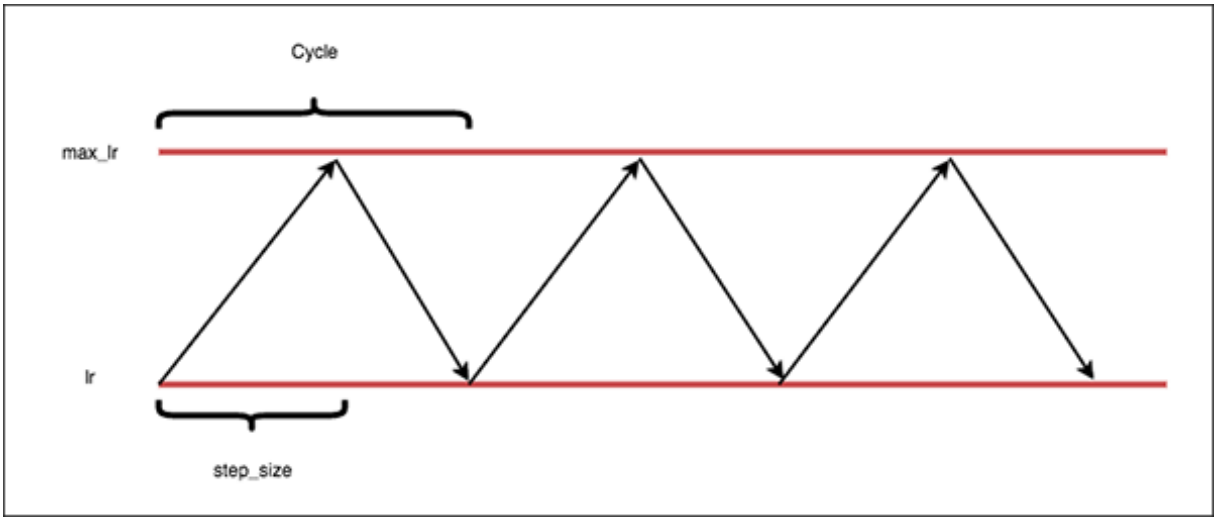

Figure 6: Ciclo triangular utilizado por autofit

4 Conclusiones

De los modelos probados para este trabajo practica se pueden obtener las siguientes conclusiones.

4.1 Algoritmos tradicionales (sklearn)

Estos modelos de clasificación obtenidos a partir de una librería son muy sencillos de utilizar y entrenar y a su vez permiten obtener un score decente en muy poco tiempo, el problema es que tienen un limite en cuanto a la relación de aprendizaje y cantidad de data utilizada. Es decir que no importa cuanta cantidad de data se incorpore estos algoritmos no podrán aprender mas, tienen un limite. Este marca su principal diferencia con las redes neuronales

4.2 Redes neuronales

Los datos necesitan transformaciones y estas pueden variar según el tipo de red a utilizar, además es mucho mas complicado generar un modelo efectivo ya que existen miles de combinaciones de capas (layers) de manera que se pueden obtener resultados variados, desde muy malos comparados con los algoritmos más simples a otros superiores. Una ventaja es el hecho de que a medida que se suministran más datos una red neuronal es capaz de seguir aprendiendo a diferencia de los algoritmos tradicionales de clasificación.

4.3 Gradient Boosting

Son modelos fáciles de utilizar y entrenar, y dependen fuertemente de sus hiperparámetros, sin embargo han dado resultados decentes rápidamente. Pero, sin embargo fue complicado mejorar dichos resultados.

4.4 BERT

Se puede concluir que el uso de embeddings BERT es el mejor para este tipo de problemas de lenguaje natural, dado a que es el que mejor resultados ha logrado. Además el uso de la librería ktrain facilitó en gran medida la cantidad de código requerida para realizar la tarea. Sin embargo, el modelo es fácil de overfittear.