TP2 Críticas Cinematográficas - Grupo 07

Introducción:

El dataset de train provisto consta de 50 mil observaciones de reseñas cinematográficas, con 3 columnas detallando id, la reseña en sí y su respectivo valor de sentimiento. Este dataset fue bastante limpio en comparación al del tp1, debido a que se nos fue entregado sin nulos o datos erróneos de por medio. El único detalle a tener en cuenta fue que había reseñas en otros idiomas por lo cual las hemos filtrado, solo quedándonos con las críticas en español, para ser consistentes y no tener un vocabulario disperso. Se nos fue solicitado para este trabajo, trabajar con PLN (Procesamiento de Lenguaje Natural) y entrenar modelos para predecir si una reseña, es positiva o negativa, es decir, su valor de sentimiento. Los modelos seleccionados fueron Bayes Naive, Random Forest, XGBoost, una Red Neuronal con Tensorflow y Keras, y un Ensamble de mínimo 3 modelos.

Por otra parte, el dataset de test provisto consta de 8599 observaciones de reseñas cinematográficas, con 2 columnas detallando id y la reseña en sí. En este caso, sus respectivos valores de sentimiento, no fueron otorgados debido a que es la gracia de la competencia en Kaggle. Nosotros no sabemos los valores verdaderos para no poder overfittear en exceso a nuestros modelos o incluso hacer trampa.

Las técnicas de PLN que fueron exploradas fueron varias. Desde el más simple preprocesamiento como solo una simple tokenización hasta preprocesamientos muy complejos que utilizaban muchas técnicas a la vez. Comenzamos utilizando un clean_text, tokenización, stopwords y el Count Vectorizer. Le pasamos ciertas funciones para que se encargue el Count Vectorizer. Donde, también, jugábamos con los N-Gramas. Siempre utilizando las siguientes combinaciones: palabras simples (ngram range(1,1)), palabras y Bi-gramas (ngram range(1,2)) y, por último, sólo Bi-gramas (ngram range(2,2)). Estas tres combinaciones eran las que daban mejores resultados, más específicamente, utilizando ambas a la vez. El único detalle de esa opción, era que a veces se eliminaban palabras que uno no quería debido a que el vectorizador, primero limpiaba, tokenizaba y luego eliminaba stopwords, por lo que se eliminaban palabras todas las palabras por separado. Por ejemplo, si en tu lista de stopwords tenías la palabra “buenos aires”, se eliminaban ambas palabras por separado y quizás quitaba información en el vocabulario; ya que la palabra “buenos” suele ser útil para analizar sentimientos. Con este preprocesamiento, teníamos resultados aceptables pero no tan buenos, no conforme con ellos, seguimos explorando.

Luego, comenzamos a trabajar con preprocesamientos más complejos. Agregamos un cálculo de Scores en las palabras para realizar un análisis de sentimientos, exploramos Stemming, la Lematización, ésta última con diferentes librerías; etc. Ya que el lematizador de la librería NLTK (Natural Language Toolkit) no funciona del todo bien en Español; probamos con el de Spacy, que parece funcionar mejor (Aunque tarda mucho más en ejecutarse). Sin embargo, de forma anti-intuitiva, dieron mejores resultados los modelos con una Lematización hecha por NLTK. También, hemos probado utilizar varios Regex, como por ejemplo, para eliminar caracteres especiales, diacríticos, signos de puntuación, convertir todas las palabras a minúsculas, etc.

Y luego de varios días de arduo trabajo y exploración en todas las combinaciones posibles para el preprocesamiento, llegamos a la conclusión de que cuanto más complejos sean los análisis, peores resultados nos daban, iban decreciendo en Kaggle. Los mejores resultados en Kaggle se alcanzan con la función preprocesar_texto_lemm para algunos modelos (llamese Random Forest), y en otros, preprocesar_texto_lemm_regex dónde agregamos dos simples regex para reemplazar signos de puntuación por espacios y luego eliminar los espacios adicionales. Ambas con Lematización de NTLK, por los motivos antes mencionados. ¡Cuanto más simple sea el análisis, mejor!

Adicionalmente, como pequeña conclusión, notamos que los mejores resultados en Kaggle han sido modelos que su predicción fue por amplia medida, mayoritariamente valores de sentimiento “positivos”, es decir, planteamos como suposición que el kaggle test a predecir, está completamente desbalanceado incluso un 75% de las críticas cinematográficas deben ser “positivas”. Este dato será muy útil para las reseñas, ya que ante la duda, decidiremos que una reseña será positiva, como es el caso de las redes neuronales, las cuales los datos de salida son valores en el intervalo entre 0 y 1. Podemos utilizar un round de numpy o también, podremos crearnos una función para jugar con ese valor de salida, seleccionando un umbral a nuestro gusto para determinar si una crítica es “positiva” o “negativa”. Por ejemplo, se han logrado muchos mejores resultados cuanto menor sea ese umbral, como 0,2 o 0,4.

Para poder mejorar el rendimiento en la competencia de kaggle, utilizamos el vocabulario del archivo de test. Vamos a calcular las matrices de ocurrencias de términos usando sklearn, las dos opciones fueron: Count Vectorizer nos devuelve la frecuencia absoluta de cada término por cada documento. Y, el TF-IDF: calcula la frecuencia de cada término por documento, y normaliza por el total de documentos donde el término aparece.

Además, concluimos que, según sus métricas, la opción adecuada para maximizar la performance de los modelos en general fue la de utilizar el Tfidf Vectorizer por sobre el

Count. En literalmente todos los modelos, el Tfidf Vectorizer daba mejores resultados.Por ende, en el caso del Random Forest, le realizamos un Cross Validation creando un pipeline, con el RF y el Tfidf Vectorizer, para optimizar los hiperparámetros del modelo y los parámetros del vectorizador. Una vez hallados estos parámetros para el Tfidf vectorizer, hemos utilizado ese vectorizador encontrado para casi todos los modelos, ya que las notebooks de colab colapsaron consumiendo toda la memoria RAM al hacer Cross Validation de un pipeline con un modelo y el vectorizador, en los modelos restantes. Además, nos parece importante comentar, que las performance de todos los modelos han incrementado en gran medida gracias a utilizar un Tfidf Vectorizer junto al vocabulario del kaggle test, pero truncado con min_df = 3, lo cual ese vocabulario de alrededor de 57 mil palabras, pasaba a tener 23 mil palabras aprox. Salvo el Random Forest, que dió mejores performance sin ese truncamiento, es decir, con el vocabulario completo.

Resumiendo, se nos comentó que en PLN, no hay una bala de plata para mejorar las performance, es decir, a cada modelo le puede aumentar sus métricas distintos preprocesamientos. Sin embargo, cuando dimos en la tecla en el preprocesamiento, absolutamente todos nuestros modelos se despegaron en cuanto a performance en kaggle, y en el test de la notebook, manteniéndose en números similares.
Concluyendo, nuestro preprocesamiento utilizado ha sido, como hemos comentado, en algunos casos la función preprocesar_texto_lemm para algunos modelos (llámese Random Forest), y en otros, preprocesar_texto_lemm_regex dónde agregamos dos simples regex para reemplazar signos de puntuación por espacios y luego eliminar los espacios adicionales (ambas lematizaciones con NLTK). Y refiriéndonos al vectorizador, siempre el Tfidf, con vocabulario del kaggle test, a veces completo (Random Forest) y a veces truncado mediante el parámetro min_df.

Construcción de Modelos:

Bayes Naive:

Primero hemos realizado un Bayes Naive 100% default sin ningún hiperparámetro optimizado. Sus números han sido correctos, las cuatro métricas rondaban por 0.82 aproximadamente y en kaggle 0.70 aprox. Para ser un modelo sin ninguna optimización, parecían ser números normales. Se nos fue informado que este modelo se debía utilizar como “base”, ya que los demás modelos debían superarlo.

Luego, hemos decidido jugar y buscar optimizar los hiperparámetros mediante Randomized Search Cross Validation. Utilizamos la métrica “F1 Score” para optimizar los

hiperparámetros, ya que es la métrica a mejorar y target en kaggle. Con esta situación, con los hiperparametros recomendados, su Score juntos al de las demás métricas aumentaban hacia los casi 0.84 aproximadamente y en kaggle subía muy poquito, 0.71. Sin embargo como hemos mencionado, cuando dimos en la tecla en el preprocesamiento y utilizando el tfidf vectorizer con vocabulario del kaggle test truncado (min_df = 3), sus números en el test de la notebook se mantuvieron, pero en kaggle se despegó hacia 0.76. De esta forma, ya pasó a ser un gran modelo, apto para ser tenido en cuenta.

Random Forest:

Primero hemos realizado un random forest arbitrario sin ningún hiperparámetro optimizado, tratando de crearlo, lo más parecido al default posible (los hiperparámetros que se han elegido han sido el random_state para eliminar la aleatoriedad entre ejecuciones, oob_score, etc). Sus números han sido levemente mejores y un tanto irregulares, las cuatro métricas se separan en distintos números, algunas cerca de 0.82 y otras por 0,84, aproximadamente; y en kaggle 0.70 también. Para ser un bosque sin ninguna optimización, son números bastante acertados e incluso un poquito mejores que los del bayes naive básico, como debería esperarse, más no así en kaggle.

Luego, hemos decidido jugar y buscar optimizar los hiperparámetros mediante Randomized Search Cross Validation. Utilizamos la métrica “F1 Score” para optimizar los hiperparámetros, ya que es la métrica a mejorar y target en kaggle. Con esta situación, con los hiperparametros recomendados, su Score juntos al de las demás métricas aumentaban hacia los casi 0.83 y otras manteniéndose; luego en kaggle subiendo a 0.71 también muy poquito. En cambio repitiendo la explicación, cuando dimos en la tecla en el preprocesamiento y utilizando el tfidf vectorizer con vocabulario del kaggle test (esta vez completo), sus números en el test de la notebook se mantuvieron, pero en kaggle se despegó hacia 0.76 al igual que el bayes. De esta forma, ya pasó a ser un gran modelo, apto para ser tenido en cuenta y llevándonos al 1er lugar de la competencia por unos cuantos días.

XGBoost:

Primero hemos realizado un XGBoost arbitrario sin ningún hiperparámetro optimizado, tratando de crearlo, lo más parecido al default posible (salvo el random_state para eliminar la aleatoriedad entre ejecuciones, etc). Sus números han sido más correctos, las cuatro métricas rondan por 0.83 aproximadamente. Para ser un modelo sin ninguna optimización,

son números acertados e incluso un poco mejores que los del bayes naive base, aunque un poco peor que el bosque básico.

Luego, hemos decidido jugar y buscar optimizar los hiperparámetros mediante Randomized Search Cross Validation. Utilizamos la métrica “F1 Score” para optimizar los hiperparámetros, ya que es la métrica a mejorar y target en kaggle. Con esta situación, con los hiperparametros recomendados, su Score juntos al de las demás métricas aumentaban hacia los 0.84, de forma regular. Pero, como no nos cansaremos de decirlo, cuando dimos en la tecla en el preprocesamiento y utilizando el tfidf vectorizer con vocabulario del kaggle test truncado (min_df 3), sus números en el test de la notebook aumentaron levemente, pero en kaggle se despegó hacia 0.75, siendo un modelo más respetable. Aunque tristemente siendo un poquito peor que los anteriores modelos, incluido el bayes naive.

Red Neuronal:

Primero hemos realizado una Red Neuronal medio básica a ojo y redondeando sus resultados, utilizando numpy. Sus números han sido muy buenos, las cuatro métricas rondando por 0.85 aprox, quizás un poquito más. Para ser un modelo sin ninguna optimización, son números excelentes, nos atreveremos a decir. Lógicamente, números que superan todos los modelos anteriores. A diferencia en kaggle, su Score fue 0.70, a priori decepcionante. Lamentablemente, no hemos podido jugar y buscar optimizar los hiperparámetros mediante ningún Cross Validation en esta instancia. Esto es debido a la carga computacional de las mismas, colapsan las notebooks de colab porque consumen toda la memoria ram. Asique lo que hemos hecho, fue jugar a mano con los distintos optimizadores y diferentes activaciones de las redes.

Continuando, para sorpresa de nadie ya a esta altura, cuando dimos en la tecla en el preprocesamiento y utilizando el tfidf vectorizer con vocabulario del kaggle test truncado (esta vez, min_df 10, debido a la carga computacional de las redes), sus números en el test de la notebook aumentaron levemente, pero en kaggle se despegó hacia 0.75, siendo un modelo más respetable. Pero, aquí viene lo bueno. No conforme con esta situación, como hemos dicho quizas varias veces en este informe, comenzamos a jugar con ese “redondeo”. Seleccionando a nuestro gusto un umbral para determinar si una crítica es “positiva” o “negativa”, obtuvimos la mejor red. En este caso, el umbral fue de 0.3, y su Score en kaggle se disparó hacia 0.79220. Ahora sí, una red neuronal en condiciones y, obviamente, el mejor modelo de todo este trabajo. Llevándonos al primer lugar de la competencia, de

nuevo, durante los días finales de la competencia.

Ensamble de Modelos:

Y llegando al último modelo, tenemos que comentar una particular situación que transitamos. Comenzamos creando un Ensamble de tipo Voting, con los mejores representantes de cada modelo. Es decir, nuestro mejor Bayes Naive, nuestro mejor Random Forest y nuestro mejor XGBoost, en cuanto a performance en kaggle se refiere. Intuitivamente, este modelo debería ser unos de los mejores a simple vista o por lo menos pelear con los mejores, pero no fue así. Comenzamos a realizar muchas pruebas con muchos modelos y absolutamente todos los ensambles, tanto Voting como Stacking, rondaban entre 0.45 y 0.60 de puntuación en kaggle, en cambio, en el test de la notebook se llegaban a métricas realmente altas. Incluso algunos modelos, tenían todas sus métricas alrededor de los 0.89, números extremadamente altos y jamás vistos. Luego de varios días de nerviosismo y desesperación, dudando en casi todos los aspectos de nuestra implementación, nos percatamos que había un pequeño error en la exportación de la predicción del kaggle test. Es decir, absolutamente todos los modelos que fuimos probando, pensando que eran malos modelos, podían ser muy buenos, pero a esa altura el error ya estaba hecho.

Haciendo borrón y cuenta nueva, una vez hallado ese error nuestro, arrancamos desde el principio otra vez. También nos parece importante comentar que los modelos Voting, los fuimos creando ensamblando Pipelines, es decir, cada modelo tiene su propio vectorizer empaquetado con él, para no perder sus vectorizadores que maximizan sus mejores performances. No resta mencionar que, los modelos y los pipelines los importamos gracias a la librería y en formato de joblib. En cambio, en el caso del Stacking, nos tiraba un error si queríamos ensamblar los pipelines, por ende, tuvimos que agarrar manualmente a los modelos individuales que se encontraban en las importaciones de los pipelines.

Volvimos a comenzar con un Voting (de parámetro hard) de los mejores representantes de cada modelo, igual que como mencionamos recientemente. Ya de entrada, las métricas en la notebook seguían siendo buenas, 0.86 aprox; pero lo tranquilizador fue ver su puntuación en kaggle, 0.71661. Pensándolo en frío, no era el mejor modelo de todos, ni tampoco superaba al mejor Bayes Naive, pero con la situación que veníamos transitando, fue relajante ver esa puntuación, que para ese determinado momento, parecía alta. Prosiguiendo con este mismo modelo Voting, continuamos variando el preprocesamiento hasta hallar el ideal. Este ensamble decide con los votos individuales de cada modelo y su

hiperparámetros importante es el tipo de ‘voting’ los cuales son: “Hard”: Elige el valor dado por mayoría estricta. Y “Soft”: Pondera con cálculos como el promedio, etc. Una vez hallado el preprocesamiento adecuado para estos ensambles, y jugando con el hiperparámetro “voting”, encontramos hasta ahora el mejor ensamble de todos. El cual, sus cuatro métricas se incrementaron con respecto a los anteriores, aproximadamente a los 0.87 y 0.88, pero su Score en kaggle fue 0.77902; lo que lo convertía en un modelo que ya imponía respeto y posicionándose como el 2do mejor modelo por detrás de la red.

Cuadro de Resultados:

Modelo	F1 - Test	Precision Test	Recall Test	Accuracy Test	Kaggle
Bayes Naive	0.84869	0.85081	0.84657	0.84932	0.76100
Random Forest	0.83115	0.80871	0.85488	0.82662	0.76817
XGBoost	0.84462	0.82056	0.86514	0.83825	0.75130
Red Neuronal	0.87580	0.85809	0.89425	0.87339	0.79220
Ensamble (Voting)	0.87816	0.87376	0.88261	0.87775	0.77902

Bayes Naive: {'fit_prior': ‘False’, 'alpha': 50, 'force_alpha’: ‘True’}

Random Forest: {'criterion': 'gini', 'min_samples_split': 19, 'n_estimators': 89, ‘ccp_alpha’: 0.0010}

XGBoost: {'learning_rate': 0.2, 'n_estimators': 128, 'max_leaves': 27, 'reg_lambda': 1.0526315789473684, 'reg_alpha': 2.631578947368421, 'gamma': 2.6530612244897958}

Red Neuronal: {'epochs': 100, 'batch_size': 50, 'activation’: ‘sigmoid’, ‘optimizer’: SGD (learning_rate = 0.01), ‘loss’: binary_crossentropy’}

Ensambles: {‘voting’: ‘soft’, ‘estimators’: [‘best_bn’, ‘best_rf’, ‘best_xgb’, ‘best_red’] }
(Son los 4 mejores modelos de cada uno, es decir, los seleccionados para el cuadro de resultados)

Conclusiones Generales:

En este trabajo, realizar un análisis exploratorio fue útil, pero en gran menor medida que el tp1. Esto se debe a que en el caso del anterior tp, se nos había entregado un dataset totalmente defectuoso, lleno de nulos, datos erróneos, outliers extraños o no posibles, etc. Sin embargo, en este trabajo sirvió para detectar que habían reseñas en varios idiomas, para chequear que no hayan nulos o errores y para tener una idea general de los datasets que se nos fueron otorgados, útil pero no fundamental y necesario como en el anterior caso.

Lo que sí fue fundamental en este trabajo, han sido las técnicas de preprocesamiento las cuáles ayudaron significativamente en la performance de todos los modelos. Como hemos comentado anteriormente, se han explorado muchas técnicas para preprocesar los datasets provistos. En el momento que hemos dado con una combinación de técnicas adecuada y no tan compleja, las performances, los números de los modelos en kaggle se han incrementado bastante.

Los Bayes Naive comenzaron en 0.69, lo mejoramos un poquito y lo dejamos en standby por un tiempo, ya que volviendo a decirlo, se nos fue avisado que debía ser un modelo base y todos deberían superarlo. Luego de un tiempo, al retomarlo, con el preprocesamiento adecuado, más la optimización de sus hiperparámetros, llegó a ser, sorpresivamente, un buen modelo en kaggle con un score de 0.76100. Incluso superando al XGBoost y al ensamble de tipo Voting.

Los Random Forest, también, comenzaron en 0.69 y con el preprocesamiento adecuado, más la optimización de sus hiperparámetros, llegó a ser nuestro mejor modelo en kaggle por un buen tiempo con un score de 0.76817, incluso siendo el modelo que nos mantuvo en la 1era posición de la competencia por aproximadamente 1 semana y media. Lastimosamente, no hemos superado el desafío de superar el primer score de los profesores en kaggle (aprox 0.745), por literalmente, 15 minutos. Una vez hecho esta submit, nos mantuvimos en primer lugar.

Siguiendo con el XGBoost, los primeros modelos tenían scores en kaggle de aproximadamente 0.67. Utilizando casi el mismo preprocesamiento encontrado para los RF, con un pequeño agregado de 2 regexs simples, la performance de todos los XGBoost comenzaron a subir. Llegando a su tope, por lo menos en lo que pudimos lograr, el score de kaggle fue 0.75130, algo quizás decepcionante ya que en el primer tp, el XGBoost Classifier había sido el mejor modelo.

Continuando con las Redes Neuronales, el preprocesamiento fue fundamental. Nuestra primera red con tensorflow y keras logró un score en kaggle de 0.70. Luego habiendo hallado y utilizado, también, el mismo preprocesamiento que los XGBoost; su performance y scores se despegaron y nos pusieron en la cima de la competencia otra vez. Posicionándonos 1eros, otra vez, en la competencia durante la semana final, gracias a una red, media básica para poder correr las notebooks. Claramente, debido a la carga computacional de las mismas, ya que colapsan las notebooks de colab porque consumen toda la memoria ram. De todas formas, la mejor red la hemos conseguido seleccionando un umbral a nuestro gusto para determinar si una crítica es “positiva” o “negativa”, en este caso fue de 0.3. Y su Score en kaggle ha sido 0.79220. En cambio, los mejores valores en test de la notebook fueron alcanzados por esta misma red, cambiando el umbral a 0.5. Esto se debe a que el dataset train está balanceado con críticas 50% positivas y 50% negativas, a diferencia del test de kaggle, lo cual sospechamos que debe rondar por 75% positivas y 25% negativas.

Y, por último en el caso del Ensamble, los primeros modelos tenían scores entre un intervalo de extremos 0.45 a 0.60 en kaggle, debido a la situación ya mencionada. Luego habiendo hallado y utilizado, también, el mismo preprocesamiento que los XGBoost sumado a respetar los vectorizadores de cada modelo gracias a ensamblar los Pipelines; su performance y scores se despegaron. Aunque llegando a su tope, en score de kaggle que fue 0.75751, algo quizás decepcionante ya que al ser un ensamble de muy buenos modelos a priori, debería tener una baja varianza y como mínimo, superar al bayes naive. Sin embargo, cuando logramos sumar al ensamble nuestra mejor red, el voting se volvió a despegar para bien yéndose a un Score de 0.77902 en kaggle; cosa que tenía más sentido y ya dejándonos bastante más conformes.

Lógicamente, el modelo más sencillo de entrenar y sobre todo el más rápido, fue el Bayes Naive. Incluso fue con el único modelo que hemos podido realizar un Grid Search Cross Validation, debido a la gran cantidad de reseñas presentes en el dataset de train. Y si encima, a eso le sumamos que al vectorizar se genera una matriz esparza pero de gigantes dimensiones, los modelos con más complejidad computacional, en correr un Cross Validation tardan horas o colapsan la notebook de colab. En algunos modelos, fue posible realizar un Randomized Search CV, siendo muy paciente, y en otros como la Red Neuronal se hizo imposible.Volviendo al Bayes Naive, aunque fue el modelo más económico, terminó siendo un más que aceptable modelo, y eso que se debía utilizarlo como base o piso. Es decir, todos los modelos deben superar sus métricas y score en kaggle; XGBoost no pudo con él.

Nuestro mejor modelo, como hemos mencionado varias veces ya, ha sido la Red Neuronal, pero un tanto “sesgada”. ¿A qué nos referimos con esto? Al umbral comentado anteriormente, con el que fuimos jugando. Para nuestro objetivo que era la competencia de Kaggle, “sesgando” un poquito a la red para que, ante la duda, prediga que una reseña sea positiva. Esto mejora los scores en kaggle pero disminuirá su productividad en la vida real. Si balanceamos ese detalle, el modelo podría ser utilizado de forma productiva tranquilamente, aunque suponemos que se podría mejorar un poco más. Si tan solo resistieran las notebooks y fuéramos capaces de realizar un Cross Validation a nuestras redes, suponemos que optimizando sus hiperparámetros (no a ojo), incrementarán sus métricas y performance, generando que sean aún mejor utilizarlas de forma productiva.

Tiempo Dedicado:

Integrante	Tareas Realizadas	Prom Hs. Semana
Matias Agustin Ferrero Cipolla	Bayes Naive Random Forest XGBoost	8 hs
Franco Ricciardo Calderaro	Preprocesamiento Random Forest XGBoost Red Neuronal Armado de Reporte	12 hs
Carlos Alejandro Orqueda	Bayes Naive Ensambles Red Neuronal	7 hs
Sebastian Kraglievich	Bayes Naive Ensambles	3 hs