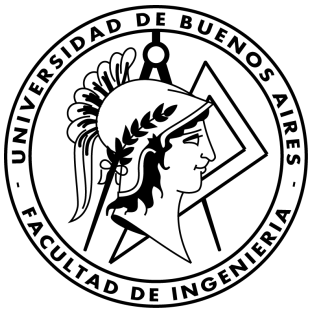

Trabajo Práctico II — Análisis de sentimientos

[75.06/95.58] Organizacion de Datos
Primer cuatrimestre de 2023

Alumno	Número de padrón	Email
Masri, Noah	108814	noahmasri19@gmail.com
Ayala, Camila	107440	cayala@fi.uba.ar
Loscalzo, Melina	106571	mloscalzo@fi.uba.ar

Índice

1. Introducción

El objetivo de este trabajo practico era el de crear modelos de prediccion para un set de datos de reseñas de peliculas. Se deseaba predecir el sentimiento detras de estas, es decir, si la reseña era positiva o negativa. En el presente informe se dara una breve explicacion de todos los modelos llevados a cabo en el trabajo y la eleccion de hiperparametros a la hora de intentar optimizar estos modelos.

Para todos los modelos se realizaron mas de una muestra; uno basico, uno con hiperparametros elegidos cuasi arbitrariamente y un tercero hecho con KFolds. En muchos de estos tambien se probaron ambas maneras de vectorizacion, es decir, usando un Count Vectorizer y luego un TFIDF.

2. Bayes Naive

Se realizaron 5 modelos de bayes naive, de los cuales 3 son hechos con Bernoulli, y dos con Multinomial. Se usaron ambos metodos de vectorizacion para cada uno de los modelos, y se realizo al final optimizacion de hiperparametros con kfolds.

2.1. Multinomial

Estos fueron nuestros primeros de muchos modelos. El primero de todos fue hecho con Count Vectorizer como generador de nuestro Bag of words para el cual, con prueba y error, decidimos poner un minimo de apariciones de 30, es decir, todas las palabras que aparecieran menos de 30 veces no quedarian en nuestro vector, se las filtraria. Con este modelo obtuvimos la siguiente matriz de confusion.

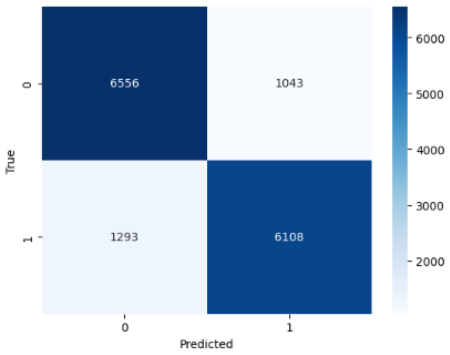

Figura 1: Matriz de confusion del modelo Multinomial de Bayes con Count Vectorizer

Se obtuvo alli, sobre las clasificaciones positivas (1) del set de testeo un F1 score de 0.84, una precision de 0.85 y un recall de 0.83, por lo que podemos decir que las metricas estuvieron bien balanceadas, y esta pudiendo generalizar bien. Este modelo, en el set de testeo obtuvo un F1 score

de 0.86, una precision de 0.87 y un recall de 0.84, lo cual nos hizo notar que esta sobreajustando un poco, pero esta brecha no era muy amplia por lo que seguiamos contando con un buen modelo. Una vez que tuvimos este modelo, decidimos hacer una prediccion en Kaggle, esperando que este diera como en nuestro Notebook, y eso no sucedio; apenas logramos que obtuviera 0.71447. Esto se siguio repitiendo en todo el resto de los modelos.

El siguiente realizado fue otro modelo multinomial, pero esta vez usando un TFIDF, que analiza la frecuencia de aparicion en vez del numero. Se decidio eliminar las palabras que tuviesen asignado un puntaje de mas de 0.5, ya que el que apareciera en la mitad de las entradas nos significaba que posiblemente era algun conector que nos habia quedado sin borrar accidentalmente. A este modelo le fue bastante peor que el anterior, obteniendo en el set de entrenamiento un F1 score de 0.81, una precision de 0.79 y un recall de 0.84, pero se redujo un tanto el overfitting, pues este en el set de entrenamiento obtuvo un F1 score de 0.82, una precision de 0.80 y un recall de 0.85, difiriendo unicamente en 0.01 del set de testing. La brecha entre este y el anterior se vio reflejada en kaggle, pues se obtuvo en las predicciones con este un 0.70498.

2.2. Bernoulli

Dado que tenemos que realizar predicciones binarias, notamos que podiamos usar tambien Bernoulli. Probamos primero con TFIDF, pero sus metricas fueron tan decepcionantes como las de la multinomial con TFIDF, pero con un sobreajuste un poco menor, y en kaggle 0.71137, por lo que volvimos a usar un Count Vectorizer y asi obtuvimos muy buenas metricas.

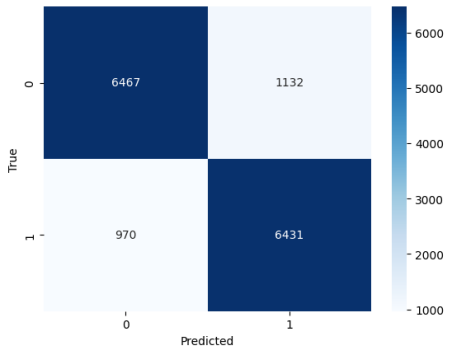

Figura 2: Matriz de confusion del modelo Bernoulli de Bayes con Count Vectorizer

Se obtuvo alli, sobre las clasificaciones positivas (1) del set de testeo un F1 score de 0.86, una precision de 0.85 y un recall de 0.87, y la brecha con el set de entrenamiento no fue tan amplia. A este modelo ademas le fue mucho mejor en kaggle que a todos los anteriores, obteniendo 0.76177, por lo que era el mejor hasta el momento.

2.3. Bayes Optimizado

Una vez hecha una especie de optimizacion a ojo, finalmente buscamos optimizar los hiperparametros. Decidimos optimizarlos a partir del modelo basico obtenido con Bernoulli con count vectorizer, ya que este habia sido nuestro mejor hasta el momento. Dado que este modelo no tiene mucho que optimizar, solo cuenta con un hiperparametros, creamos un pipeline y aprovechamos para optimizar tambien los hiperparametros del vectorizador. Pusimos para el alpha unos valores bastante variados para ver si era mejor reducir la influencia de los datos de entrenamiento o no. Para el count vectorizer, dejamos que tome las palabras por separado o de a dos, teniendo en cuenta que hay enfatizadores como por ejemplo "muy", donde cambia mucho el contexto en el que este aplicado. Luego con el filtro de palabras, decidimos que el minimo de apariciones para que entre este entre 5 y 200, pues queriamos dejar un rango amplio pero que no represente mas que un 0.01 de las entradas, y el maximo que sea desde una aparicion del 40 % hasta un 60 %.

Este modelo dio buenas metricas en el set de testeo, pero dio muchisimo mejor en el set de entrenamiento, teniendo una diferencia de mas de 0.07 entre el set de entrenamiento y el de testeo, siendo el rendimiento en entrenamiento cercano a 1. Este modelo en kaggle obtuvo 0.76061, que no estuvo tan mal, pero no supero al anterior, que ademas tardo muchisimo menos en correr.

3. Random Forest

Para este se crearon dos modelos; un modelo basico y un modelo optimizado. Para ambos se creo un pipeline que aplicaba primero el vectorizador y luego aplicaba el modelo per se. Esto facilito mucho la realizacion de la prediccion para subirlos a kaggle.

3.1. Modelo Basico

Este fue el primero creado a fines de ver como le iba inicialmente sin ningun tipo de optimizacion. A este modelo le fue bastante bien sobre el set de testing, obteniendo un F1 score de 0.83, una precision de 0.84 y un recall de 0.83, pero ahora cuando miramos las metricas del set de entrenamiento nos encontramos con lo encontrado en la figura 5. Se observa que este modelo esta realizando un sobreajuste abismal, lo cual tiene sentido pues este modelo permite que los arboles crezcan hasta donde lo requieran. Las predicciones en kaggle dieron resultados mediocres, obteniendo una puntuacion de 0.70091.

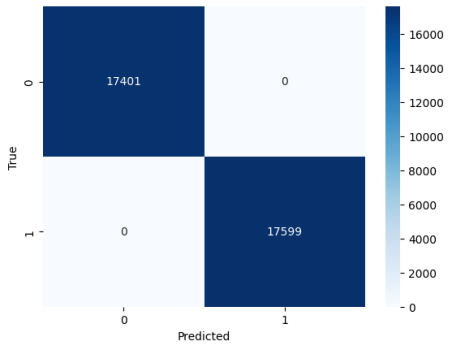

Figura 3: Matriz de confusion del modelo Random Forest basico sobre set de entrenamiento

3.2. Random Forest Optimizado

Dados los resultados del modelo anterior, debimos regular los hiperparametros para poder regular el sobreajuste de alguna manera. Al igual que en la optimizacion de bayes naive, decidimos tambien optimizar los hiperparametros del vectorizador, poniendo rangos similares a los anteriores, aunque permitiendo mas repeticiones de las palabras. Seteamos el rango de la cantidad de estimadores entre 50 y 300, intentando mantener un balance entre performance y tiempo de corrida, esperando que ademas no haya tanto sobreajuste. La profundidad de los arboles se puso en un rango de 1 a 10, para evitar que crezcan demasiado y sobreajusten. Viendo la performance una vez entrenado, notamos que al modelo le fue peor que al modelo basico en cuanto a metricas, pero cuando observamos las metricas del set de entrenamiento notamos que se redujo un monton el sobreajuste. A este modelo, si bien no le fue tan bien como a los modelos de bayes en nuestro notebook, obtuvo, sus resultados en kaggle fueron comparables con los de los modelos multinomiales.

Concluimos igualmente sobre este tipo de modelo que para el tipo de problema con el que contamos este no es muy apto; los modelos de Bayes resultaron mucho mas adecuados y eficientes. No solo corrieron muchisimo mas rapido sino que tambien dieron resultados mucho mejores.

4. XGBoost

El modelo de XGBoost nos resulto muy efectivo en el trabajo practico anterior por sus hiperparametros para controlar el overfitting y porque fue diseñado para manejar dataset de mayor tamaño. Como en todos los otros casos, se creo primero un modelo basico, usando los hiperparametros por defecto. Este modelo dio metricas regulares, pero ademas realizo bastante sobreajuste. Aprovechamos entonces la existencia de los hiperparametros de regulacion y los optimizamos usando cross-validation. Se seleccionaron los siguientes rangos de hiperparámetros:

1. 'learning_rate': (0.005,0.2): Este determina la contribución de cada árbol al modelo general. Un learning rate más alto permite que cada árbol tenga un peso más fuerte y, por lo tanto, tenga un impacto mayor en la predicción final.Al optimizarlo quedo en 0.2, el valor mas alto proveido lo cual implica que puede haber overfitting en el modelo resultante.

2. 'max_depth': (3,10): Esta es la profundidad máxima de cada árbol en el modelo. Controla la cantidad de divisiones o niveles que puede tener un árbol. Un valor más alto de max_depth permite al modelo capturar relaciones más complejas en los datos, pero también puede conducir a un sobreajuste. Al optimizarlo obtuvimos 9 lo cual es alto y es posible que haya contribuido al sobreajuste.

3. 'n_estimators': (50, 200): Este indica el número total de árboles en el modelo. Cuanto mayor sea el número de estimadores, más complejo será el modelo. Sin embargo, un número demasiado alto de estimadores puede llevar al sobreajuste. Al optimizar quedo en 72, lo cual simplifico el modelo.

4. 'subsample': (0.5, 1.0): Este determina la fracción de muestras utilizadas para entrenar cada árbol individualmente. Un valor menor que 1.0 hace que el entrenamiento sea estocástico, lo que puede ayudar a reducir el sobreajuste. En este caso, al optimizarlo obtuvimos 0.875, lo cual reducira el sobreajuste pero no es muy bajo.

5. 'gamma': (0.1, 3): Este es un parámetro que controla cuándo se realizará una división en un nodo del árbol. Un valor más alto de gamma indica que se requerirá una ganancia más grande para realizar una división, lo que puede conducir a un modelo más conservador con menos divisiones. Al optimizarlo quedo en 0.1, lo cual hizo que se construya un modelo bastante conservador.

6. 'lambda': (0.1, 3.0): Este es un parámetro de regularización que controla la penalización de la complejidad del modelo. Un valor más alto de lambda reduce la complejidad del modelo al agregar más regularización. Al optimizarlo quedo en 0.275, lo cual es bajo y deberia ayudar a simplificar el modelo.

Sin embargo, como XGBoost funciona mediante la combinación de múltiples árboles de decisión nos quedaba un modelo altamente complejo. Esto le dificulta la interpretación de los resultados y el análisis de qué características son las más influyentes en la predicción del sentimiento. En el contexto de las reseñas, es importante comprender qué aspectos específicos generan una respuesta positiva o negativa, y XGBoost no nos brindo una visión clara de esto.
Para hacer un analisis de nuestro lado de como se estaba procesando los datos en el modelo, imprimimos las importancias que le estaba asignando a las palabras de nuestro vocabulario y

encontramos que habia palabras que tenian alta importancia que estaban repetidas, por ejemplo 'mal','mala' y 'malo' tenian todas importancia alta.

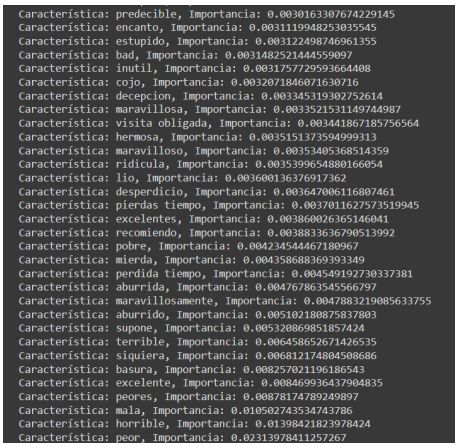

Figura 4: Importancia de las caracteristicas inicial

Encontramos una herramienta que combina las palabras similares, por lo cual las tres se combinarian en 'mal'. Los resultados construyendo el modelo con los hiperparametros calculados en la optimizacion previa y entrenandolo con el set de datos modificado con esta herramienta vimos una leve mejora en las metricas y vimos una mejor distribucion de las importancias. En el ejemplo dado, 'mala' tenia una importancia de 0,01 y cuando aplicamos el stemmer, 'mal' token cual abarca todas las anteriores, paso a tener una importancia de 0,06. Lamentablemente, estos cambios no implicaron una mejora en las metricas en kaggle.

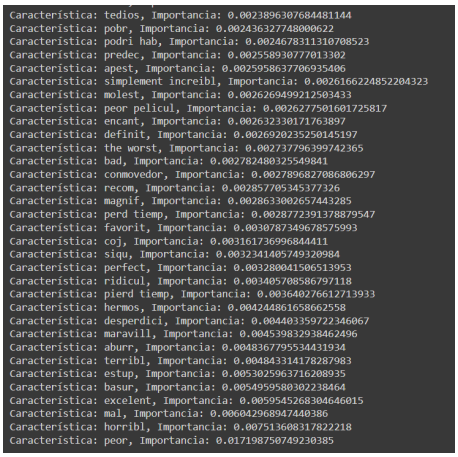

Figura 5: Importancia de las caracteristicas luego de aplicar Stemmer

Ademas, XGBoost puede ser sensible a características irrelevantes o ruidosas. Por ejemplo,

vimos que palabras que no tienen ningun sentimiento inherente como 'siquiera' y 'supone' tenian importancia bastante alta.

Aunque pudimos mejorar los resultados obtenidos con un XGBoost basico por medio de optimizacion de hiperparametros y aplicando el Stemmer, concluimos que XGBoost no es el mejor modelo para predecir lo necesitado.

5. Redes Neuronales

Con este tipo de modelos en un principio tuvimos bastantes problemas. Los modelos de redes neuronales requieren que tus datos de entrada esten en un array. Al usar la funcion to_array sobre la variable de independiente de nuestro dataset que contaba con el count vectorizer aplicado, el entorno se crasheaba, pues usaba mas RAM de la que teniamos disponible. Es por esto que en principio se opto por usar las herramientas de tensor flow para agregar capas que realizaran esta vectorizacion. El problema con estas fue que el rendimiento era muy bajo, y el tiempo de ejecucion excesivamente largo. Es por esto que se opto por un enfoque un tanto distinto.

5.1. Modelo con una capa oculta y dropout

Este fue nuestro primer modelo realizado. Se uso la funcion train_on_batch que provee keras, tomando lotes de 32 entradas, y aplicando la funcion to_array sobre estas. Esto permitio un entrenamiento gradual que no rompio nuestro entorno, y su corrida no fue tan larga. Se eligieron distintos valores de dropout pseudo arbitrariamente, apagando mas neuronas en la primera capa que en la segunda. El modelo obtuvo muy buenas metricas sobre el set de testing, obteniendo un f1 score de 0.88, una precision de 0.85 y un recall de 0.92, pero lamentablemente no podemos controlar el sobreajuste, ya que para predecir nuevamente necesitamos las entradas vectorizadas, y como dijimos anteriormente, esto no nos fue posible. En kaggle este modelo no obtuvo un f1 score comparable con el de nuestro notebook local, pero igualmente llego a obtener 0.77767, que hasta el momento habia sido el mejor.

5.2. Modelo con dos capas ocultas, dropout y early stopping

Para este modelo se agrego una capa mas, agregando mas neuronas en la capa de entrada y manteniendo igual las siguientes. Se uso nuevamente la idea del dropout gradual, apagando un menor porcentaje neuronas mientras va avanzando, y ademas se agrego early stopping, intentando evitar al 100 % el sobreajuste. Se tomo un enfoque de entrenamiento levemente distinto; se mantuvo la idea de la vectorizacion por lotes, pero ahora se construyo una funcion que unicamente vectoriza

esos lotes, y luego el entrenamiento se realizo todo junto. El entrenamiento de este modelo fue mas costoso y largo. Se creo luego un modelo muy similar, esta vez con una capa oculta mas y aumentando el dropout pero manteniendo la idea de la reduccion gradual a traves de las capas, y con este se lograron excelentes resultados, teniendo entonces nuestra mejor prediccion en kaggle: 0.78348.

6. Ensambles

Se decidio para esta parte crear varios ensambles del tipo voting, intentando buscar la mejor combinacion de modelos. Se esperaba que, usando modelos de un no tan alto rendimiento en las partes anteriores, encontrar un modelo que los una a todos y de esa manera consiga una mejor performance.

6.1. Hibrido

Se creo un primer ensamble hibrido en el que los modelos a usar fueron random forest, SVM, KNN y XGBoost. Esta seleccion diversa se baso en el hecho de que todos estos toman un enfoque distinto para la seleccion del output, por lo que juntarlos pensamos nos permitiria obtener una vision mas completa y equilibrada del problema.Se creo un pipeline en que se aplico tambien el count vectorizer, donde se pidio que el minimo de apariciones de una palabra sea de 1000, con el fin de filtrar muchas palabras irrelevantes y reducir el ruido que estas generaban. La performance de este modelo fue bastante buena sobre el set de testing local, pero fue excesivamente buena sobre el set de entrenamiento, obteniendo un f1 score de 0.96, por lo que concluimos que el modelo estaba sobreajustando mucho. En kaggle este modelo obtuvo menos de 0.7 de f1 score.

6.2. SVM

Se creo un segundo modelo de voting que solo contaba con 3 modelos de SVM, donde variamos el kernel. Esta decision se debio a que el modelo de support vector machines esta basado en bag of words y consideramos podia ser bueno para capturar las caracteristicas importantes a nivel palabras. A este modelo le fue mucho mejor que al anterior en el conjunto de test obteniendo un f1 score de 0.87, una precision de 0.83 y un recall de 0.92. Sin embargo, al observar las metricas sobre el set de entrenamiento nos encontramos con que el sobreajuste persistia, pues se obtuvo en este un f1 score de 0.97. El rendimiento en kaggle mejoro bastante igualmente, obteniendo para este un puntaje de 0.76681.

6.3. Bayes Naive

Con la misma justificacion que el voting de SVM, y agregando el hecho de que los modelos de bayes cada uno por si solos dieron buenas predicciones, decidimos hacer un modelo de voting. Para este se usaron ciertos hiperparametros tomados del kfolds hecho en la primera parte para optimizar bernoulli, mientras que a los otros dos se los puso con sus hiperparametros por defecto. Este modelo fue super rapido de correr, a diferencia de los anteriores, y obtuvo un mejor rendimiento, tanto local como en kaggle. Este modelo no conto con tanto sobreajuste, teniendo una diferencia de 0.02 entre todas las metricas de entrenamiento vs las de test, obteniendo en test un f1 score de 0.86. En kaggle, esta metrica disminuyo, obteniendo 0.76526. Esto implico una mejora de 0.004 aproximadamente con el mejor de los modelos de Bayes de la primera parte.

6.4. Optimizado

Dado que se obtuvo buenos resultados haciendo voting con los modelos de bayes, y que este modelo es bastante liviano de correr, se decidio optimizar sus hiperparametros haciendo kfolds. Se eligio un rango entre 0.1 y 1.5, equiespaciado, dividido 10 veces para el alpha de todos los modelos, ya que este es el rango en que dicho alpha se suele encontrar. Se optimizaron tambien los hiperparametros del vectorizador, poniendo los mismos rangos que al hacer la optimizacion de bayes, a excepcion del max_df, en que se cambio el rango a uno mas alto, permitiendo mas apariciones de una palabra sin filtrarla.

Se realizaron durante el proceso modelos de tipo stacking, pero estos modelos eran super pesados, muy largos de correr y sus resultados no nos parecieron proporcionales. Es por esto que decidimos eliminarlos, pero permanecen en Kaggle ciertas predicciones hechas con los mismos.

7. Conclusion

El vocabulario empleado al realizar una reseña es tan amplio como el empleado para tener una conversacion o incluso escribir una novela. Considerando que nuestro dataset contaba con unicamente 50000 reseñas, es esperable que no pueda generalizar perfectamente, y cuando se encuentre con un dataset que quizas tiene datos muy distintos, sus predicciones no sean las mejores, por lo que la brecha con las predicciones en Kaggle quedaria entonces justificada. Igualmente, logramos un puntaje de 0.78, que dadas las conclusiones, estuvo bien.