TP2: Críticas Cinematográficas - Grupo 25

Introducción

Para esta entrega, debimos crear y entrenar modelos predictivos para determinar y comprender las emociones del conjunto de datos que estaba conformado por texto. El objetivo, a partir de un dataset con reseñas de películas, fue determinar si éstas eran positivas o negativas. A partir del análisis básico del dataset, observamos que tiene 50000 registros y 3 columnas (id, reseña y sentimiento) y a partir de un análisis manual, vimos que había reseñas en idioma inglés.
Con este vistazo a simple vista, comenzamos a utilizar técnicas de procesamiento del lenguaje natural tales como: limpieza de las reseñas (convertir texto en minúsculas, sacar tildes, etc.), lematización, sacar stopwords y filtrar las reseñas en inglés.

Algo para destacar, es que tuvimos que trabajar con distintas combinaciones de técnicas de PLN para lograr mejorar cada modelo. En este link están los dataframes con los distintos preprocesamientos: drive.

Para la vectorización, en todos los modelos que lo requirieron utilizamos TfidF Vectorizer, ya que fue el que mejor rendía (aunque ralentizaba un poco el proceso), a excepción de las redes neuronales que utilizamos Tokenizer y redujimos el vocabulario a 15000 palabras.

Cuadro de Resultados

Modelo	F1-Test	Presicion Test	Recall Test	Kaggle
Bayes Naive Base	0.86	0.88	0.83	0.71292
Bayes Naive Multinomial	0.82	0.83	0.80	0.70401
Bayes Naive Bernoulli	0.79	0.83	0.76	0.70032
Random Forest Base	0.83	0.84	0.83	0.7513

Random Forest Mejores Hiper	0.82	0.83	0.82	0.69664
XGBoost base	0.83	0.85	0.82	0.6695
XGBoost Mejores Hiper	0.85	0.86	0.84	0.70381
Red Neuronal	0.87	0.84	0.90	0.72339
Ensamble (Voting)	0.87	0.87	0.87	0.71

Resumen de cada Modelo

Bayes Naive: es un clasificador que se basa en el teorema de Bayes. Asume que todas las palabras en un texto son independientes entre sí, y calcula la probabilidad de que una instancia pertenezca a una clase dada las características observadas. En el trabajo usamos dos variantes.

- **Multinomial**: asume que las características son representativas de la frecuencia de ocurrencia de palabras o eventos en el conjunto de datos.
- **Bernoulli**: se utiliza específicamente en problemas de clasificación binaria. Asume que las características son binarias y se centra en si una característica está presente o no.

Random Forest: se realiza una partición del dataset a nivel de columnas y de observaciones, y se entrena un árbol sin podar con cada una, al finalizar nos quedamos con el mejor de ellos y repetimos el proceso. Entonces tendremos n mejores modelos clasificadores. Cuando llega una nueva observación, cada modelo la clasificará. La clasificación queda determinada por la mayoría.

XGBoost: está basado en boosting y construye un conjunto de árboles de forma secuencial, y cada uno se enfoca en corregir los errores de los anteriores. Incluye términos de regularización en la función de pérdida para controlar el sobreajuste y mejorar la generalización del modelo.

Voting: se entrenan varios modelos distintos basados en el mismo dataset. La clasificación queda determinada por la mayoría.

Redes Neuronales Recurrentes: es un tipo de arquitectura de red neuronal para procesar datos de series temporales. Tienen conexiones que forman ciclos, lo que les permite mantener una memoria de la información procesada anteriormente. Se utilizan para trabajar con problemas donde el orden de los elementos es crucial, como el análisis de sentimientos o la generación de texto.
Para el trabajo utilizamos Gated Recurrent Unit, que es una variante para abordar las limitaciones que presenta el desvanecimiento del gradiente.
Además, a diferencia de los demás modelos, para poder entrenar, tokenizamos el texto usando Tokenizer, ya que presentamos problemas con el TFIDF Vectorizer. Con Tokenizer pudimos definir una cantidad específica de palabras con las que trabajar (las quince mil más frecuentes), y que cada secuencia tenga como máximo 500 palabras. Para esta etapa fue muy importante la parte de preprocesamiento del texto, como la limpieza y quitar las stopword para mejorar la calidad del análisis.

Luego, manualmente probamos distintas arquitecturas, variando el uso de capas GRU y LTSM. Estas arquitecturas tienen rendimientos muy similares, a diferencia que notamos un entrenamiento más rápido con las GRU ya que es más simple. También, experimentamos con distintos números de neuronas, número de capas, implementando también modelos piramidales. Finalmente, nos quedamos con una red neuronal con una capa de entrada Embedding, con dos capas intermedias GRU con 128 neuronas cada una y regularización Dropout. Al compilar el modelo, usamos un optimizador Adam con los parámetros vistos en la teoría.

No se realizó mejora de hiperparámetros puesto a que el tiempo de entrenamiento del random search había superado las 15 horas, sin poder concluir con la ejecución del mismo. Consideramos que, comparando el resto de modelos, al optimizar parámetros no creemos que se obtenga una mejora lo suficientemente buena que justifique todo ese tiempo de entrenamiento que conlleva. Lo principal fue definir la arquitectura y en eso metimos mucho énfasis.

Mejora de hiperparámetros

Bayes Naïve: se realizó la mejora de hiperparámetros para el TFidFVectorizer en cada modelo, y luego hiperparámetros específicos de los mismos.

Multinomial:

- *alpha*: es el hiperparámetro de suavizado de Laplace.
- *fit_prior*: indica si se deben aprender las probabilidades a priori de las clases o si se deben utilizar probabilidades uniformes.

- *class_prior*: permite proporcionar manualmente las probabilidades a priori de las clases.

Bernoulli tiene los mismos hiperparámetros que el modelo multinomial, a excepción que tiene también:

- *binarize*: establece un umbral para binarizar (convertir en 0 o 1) las características. Si se proporciona un valor, todas las características con valores menores que el umbral se establecen en 0, y las que son mayores o iguales al umbral se establecen en 1.

Random Forest: se trabajó con los hiperparámetros default del TfidF Vectorizer y se utilizó GridSearch para la búsqueda de los mejores hiperparámetros del RF, ya que a partir del modelo base que contenía parámetros bastante pequeños que rindió bastante bien, el Grid buscará también valores de bajo rango ya que tiende a evitar el sobreentrenamiento.

- *criterion:* función para medir la calidad de la división.
- *min_samples_leaf:* número mínimo de muestras requeridas en un nodo hoja.
- *min_samples_split:* número mínimo de muestras requeridas para dividir un nodo.
- *n_estimators:* cantidad de árboles

XGBoost: trabajamos con los valores default del TFidFVectorizer salvo la cantidad máxima de features y la máxima aparición de las palabras para no sobrecargar la notebook y mejoramos los siguientes hiperparámetros del XGBoost.

- *learning_rate*: tasa de aprendizaje.
- *max_depth*: máxima profundidad de cada árbol.
- *colsample_bytree*: porcentaje de features usadas para cada árbol (valores muy altos, posible overfitting).
- *n_estimators*: cantidad de árboles a construir.
- *min_child_weight:* peso mínimo de los hijos, ayuda a controlar el sobreajuste.
- *reg_alpha:* regularización para los pesos de las hojas.
- *reg_lamda:* similar alpha pero para la sintonia fina.

Detallamos el caso del mejor modelo base al Random Forest que nos ha dado un score de 0.7513 y fue el que mejor rindió de todos y su entrenamiento fue rápido.

Conclusiones generales

Tareas del trabajo práctico

Fue útil realizar un análisis exploratorio de los datos, para poder entender mejor la complejidad del problema y luego introducirnos con el preprocesamiento del mismo para poder trabajar mejor con los modelos.
Las tareas de preprocesamiento (remoción de reseñas en inglés - stopwords - limpieza - lematización) ayudaron en algunos casos:

Bayes Naïve: el modelo sin ningún tipo de preprocesamiento funciona mejor. La lematización empeoró muchísimo su rendimiento.

Redes Neuronales: para este caso, el preprocesamiento fue fundamental ya que era importante trabajar las redes con las principales palabras del dataframe, y reducir el vocabulario (en caso contrario, la red sería muy compleja computacionalmente). Las stopwords fueron eliminadas ya que son palabras muy frecuentes que no aportan significado, y se quitaron los signos de puntuación, tildes, etc. para poder tratar las palabras de forma consistente.

Random Forest: este modelo rindió su mejor performance sin sus parámetros mejorados y con el dataframe procesado completamente (stopwords + lematización + reseñas en inglés + limpieza). Sin embargo, a la hora de mejorar los hiperparámetros nos encontramos con que el rendimiento disminuyó, habiendo probado con la búsqueda por RandomSearch y GridSearch. Concluimos que con la búsqueda de los hiper, conseguimos un modelo menos pesado a la hora de ser entrenado, pero que no ayuda a mejorar el rendimiento.

XGBoost: con este modelo probamos las siguientes combinaciones de procesamiento: completo, sin lematizacion, sin lematizacion y con stopwords y sin tokenizar. Funcionó mejor el modelo base con el último, aunque no ha tenido gran rendimiento. A la hora de mejorar los hiperparámetros nos encontramos con dificultades ya que se consumía la RAM de la notebook, así que en vez de utilizar todos los parámetros default del TFidFVectorizer, regulamos los max_features (número total de palabras que se usarán) y max_df (frecuencia máxima de una palabra antes de ser ignorada) y de esa manera, pudimos mejorar los hiperparámetros del XGBoost. Conseguimos una leve mejora, pasando del 0.6829 al 0.70168, y se logró entrenando con el dataframe sin lematización.

Voting: generamos un ensamble con 3 modelos base (RF, XGBoost y un Bayes), del cual conseguimos un score bastante aceptable (0.71 aproximadamente), probamos con distintas combinaciones de modelos y esta es la que mejor funcionó.

En test, el mejor rendimiento lo tuvo la red neuronal (0.87 f1-score) seguido del modelo base de Bayes (0.86 f1-score). Luego en Kaggle, estos resultados bajaron considerablemente, a 0.72339 y 0.71292 respectivamente.

El modelo más sencillo de entrenar ha sido el Bayes Naive ya que su resultado óptimo lo obtuvimos al no hacer ningún tipo de preprocesamiento en el texto, y además, en entrenarse tardó menos de 1 minuto.

Otro modelo muy rápido y "sencillo" de entrenar es el ensamble Voting, sin contar el tiempo de armado y mejora de los modelos que utilizamos para ensamblar.

Desempeño de los modelos. ¿Pueden ser llevados a producción?

Considerando que en el conjunto de prueba de Kaggle hemos obtenido resultados buenos, podemos demostrar que los modelos tienen un buen rendimiento en datos de prueba. Pero hay cosas que mejorar.

Uno de los factores más importantes del Machine Learning es la actualización de los modelos para mantenerlos a medida que se recopilan nuevos datos. Teniendo en cuenta esto, creemos que hemos entrenados los modelos con una escasa cantidad de datos. Aún así, si hubiésemos tenido un dataframe más grande, podríamos haber abarcado un vocabulario más amplio para entrenar los modelos pero como contamos con recursos limitados, nuestros modelos hubiesen demorado muchísimo en entrenarse.

Nosotros como estudiantes contamos con recursos limitados, valoramos muchísimo que un modelo se desempeñe bien y se entrene rápido. Las redes neuronales son un modelo predictivo muy fuerte, pero particularmente para nosotros fue tediosa la tarea de poder entrenarlas. Nuestra arquitectura de red neuronal con tan solo 4 épocas tardó 1 hora en terminar de entrenarse, luego, el RandomSearch demoró casi 10 horas.

Consideramos que nuestros modelos son aceptables, pero para usarlos de manera productiva a lo mejor hay algunas cosas que ajustar que como estudiantes no tenemos a nuestro alcance.

Tareas Realizadas

Integrante	Promedio Semanal (hs)
Miranda Marenzi	15 horas
Mariana Juarez Goldemberg	12 horas
Lisandro Román	11 horas