TP 2 Críticas Cinematográficas

Grupo 17

Systeam

Integrantes

Del Rio, Juan Sebastián - 103337

Brizuela, Sebastián - 105288

Agha Zadeh Dehdeh, Lucía - 106905

Fecha de entrega: 7 diciembre de 2023

Introducción

El objetivo del presente trabajo es predecir si una crítica cinematográfica es **positiva** o **negativa**, para esto, armamos y entrenamos distintos modelos de clasificación.

En el análisis exploratorio del DataFrame se encontraron **50000 filas y 3 columnas**, las columnas corresponden al **"ID"** de cada registro, **"review_es"**, que son críticas de películas y **"sentimiento"**, que dice si la crítica es "positiva" o "negativa"; este último es nuestro target.

Preprocesamiento

En primer lugar se utilizó una librería **"langdetect"** para detectar el idioma de las críticas, agregando una columna "idioma" al DataFrame, en el cual **se observó 1817 registros que contenían la crítica ("'review_es") en inglés, por lo tanto se decidió eliminarlas**, **quedando 48183 críticas en español.** Una vez eliminadas se remueve la columna "idioma" del DataFrame ya que solo era utilizada para filtrar dichas críticas.
Se realizaron predicciones con y sin las críticas en inglés y se detectó una mejoría al no estarlas.

Como siguiente paso **eliminamos los caracteres especiales** que podían contener las críticas (*acentos, signos de puntuación, etc*).

Para realizar las predicciones reemplazamos en la columna "sentimiento" cuando era 'positivo' por un 1, y cuando era 'negativo' por un 0.

Respecto a datos faltantes y duplicados, **no se registraron datos faltantes ni duplicados.**

Se utilizó el **Tfidvectorizer** el cual consiste en una ponderación de frecuencia invertida, dando mayor importancia a las palabras de menor frecuencia. Se eliminaron los **stopwords** en español. El máximo de características a utilizar para cada modelo se iban modificando sea a mano con las redes neuronales o de forma aleatoria con RandomSearchCV utilizando el parámetro 'max_features' y también se utilizó el parámetro 'ngram_range' que determina que n-gramas (*es una secuencia contigua de n elementos de una cadena o texto*) se utilizará.

Cuadro de resultados

Modelo	F1	Precisión	Recall	Accuracy	Kaggle
Bayes Naive *(Mejor)*	0.86911	0.85916	0.87930	0.86748	0.75033
Random Forest	0.85331	0.83575	0.87163	0.85005	0.72281
XGBoost	0.86025	0.85134	0.86934	0.85866	0.70478
Red Neuronal	0.87670	0.87670	0.87670	0.87670	0.74471
Stacking	0.86764	0.84533	0.89116	0.86293	0.74626

Descripción Modelos

Los valores de parámetros colocados son un ejemplo de un modelo de los tantos entrenados.

- **Bayes Naive (Multinomial)**:

algoritmo utilizado comúnmente en clasificación de texto. Se basa en el teorema de Bayes y asume independencia entre las características (o palabras) en los datos. El Naive Bayes Multinomial calcula la probabilidad de que un documento pertenezca a una categoría específica basándose en la frecuencia con la que aparecen ciertas palabras en esa categoría.

hiper-parámetros utilizados:

- 'bn__alpha': uniform(0.5, 3.0): hiperparámetro de suavizado de Laplace.
- 'bn__fit_prior': [True, False]: aprender las probabilidades previas de la clase o no. Si es falso, se utilizará un previo uniforme.
- 'bn__class_prior': [None, [0.3, 0.4, 0.3]]: Probabilidades previas de las clases. Si se especifica, los antecedentes no se ajustan según los datos.

Parámetros del mejor modelo:

- bn__alpha': 0.120584'
- maximas caracteristicas del vectorizer: 15000 (Se utilizaron diferentes cantidades de características y se noto que a pesar que en las métricas daba valores no mayores de 0.87 en kaggle daba buenos resultados sin embargo cuando se utilizaban características mayores a 100.000 en las métricas daba valores mayores de 0.87 pero en kaggle los resultados no eran mejores que cuando se utilizaban menos características)
- n-gramas: (1, 2)

- **Random Forest:** Crea múltiples árboles de decisión y combina sus resultados para tener una predicción más precisa.

hiper-parámetros utilizados:

- ‘tfidfvectorizer__max_features': [1000, 5000, 9000, 11000]: Cantidad de palabras relevantes
- 'randomforestclassifier__n_estimators': [200, 300]: Número de árboles que se van a construir
- 'randomforestclassifier__min_samples_split': range(2,5): Cantidad Mínima de muestras para separar un nodo interno
- 'randomforestclassifier__min_samples_leaf': range(1,5): Controla el número mínimo de muestras requeridas en un nodo hoja
- 'randomforestclassifier__max_depth': range(25,40): Máxima profundidad que puede alcanzar los árboles
- 'randomforestclassifier__criterion': ['entropy', ‘gini]: Medir la impureza de la división
- ‘randomforestclassifier__ccp_alpha’: (0, 0.0001, 10): Parámetro de poda

Parámetros del mejor modelo:

- ‘tfidfvectorizer__max_features': 11000
- 'randomforestclassifier__n_estimators': 300

- 'randomforestclassifier__min_samples_split': 4
- 'randomforestclassifier__min_samples_leaf': 4
- 'randomforestclassifier__max_depth': 38
- 'randomforestclassifier__criterion': entropy
- ‘randomforestclassifier__ccp_alpha’: 2.2e-05

- **XGBoost:** Crea árboles de decisión en serie y en cada iteración corrige los errores de los árboles anteriores realizando una regulación para evitar un sobreajuste. Hiperparámeros utilizados:
  - 'n_estimators': range(50, 200, 10): número de árboles que se van a construir.
  - 'max_depth': range(3, 10): máxima profundidad que pueden alcanzar los árboles.
  - 'learning_rate': [0.01, 0.1, 0.2, 0.3, 0.5]: establece la tasa de aprendizaje del modelo.
  - 'subsample': [0.7, 0.8, 0.9, 1.0]: proporción de muestras que se utilizarán para entrenar cada árbol individual.
  - 'colsample_bytree': [0.6, 0.7, 0.8, 0.9, 1.0]: proporción de características (columnas) a considerar al construir cada árbol.
- **Red Neuronal:** Comenzamos con la arquitectura utilizada en el trabajo práctico anterior la cual tenía 7 capas, sin embargo, no obtuvo muy buenos resultados, así que modificamos la cantidad de capas, finalmente la que tuvo mejor rendimiento fue la que tenía 5 capas; para la cantidad de neuronas también se utilizó la que tenía la arquitectura base, aunque también se modificaron en un rango entre 40 y 100, además se probó poner una cantidad proporcional a la cantidad de palabras en el vocabulario devuelto por el TfidVectorizer, primero igual a la cantidad total de palabras pero impactó demasiado en el tiempo de entrenamiento sin mostrar mejoras en los resultados, entonces se concluyó en dejar 50 unidades por capa, ya que es la que mejor resultados dió; en cuanto a las capas ocultas, se encuentran: una primer capa oculta (“Dense”) con función de activación “linear”, una segunda capa oculta (“Dense”) con función de activación “relu” y regularización de kernel “l1_l2” (luego de haber probado además las regularizaciones “l1” y “l2”) y una capa Dropout, y finalmente una capa de salida (“Dense”) con una unidad y función de activación “sigmoide”. Entre los optimizadores que se probaron están: “AdamW”, “Adafactor”, “Adamax”, “Adadelta”, “Nadam” y por último el que mejores resultados dió: “Adam”. También se modificó la cantidad de “max_features”, utilizando valores entre 5000 y 11000, ya que era el rango que permitía utilizar sin que se consumiera la RAM de la notebook, el valor que mejor resultados dió es “max_features”=7000. Nota: el mejor modelo de Red se consiguió previo a la incorporación del filtro de críticas en inglés. Además se intentó utilizar Embedding, pero debido a no conseguir agilizar el tiempo de entrenamiento, se decidió quitar esta característica.
- **Stacking:** Ensamble que entrena modelos base para combinar predicciones y luego usar un modelo meta para la estimación final, en nuestro caso usamos como modelos base al Random Forest y al XGboost, y como modelo meta al MultinomialNB. Decidimos usar este ensamble ya que fue el que mejor nos dio en el trabajo práctico anterior. Nota: Para la predicción de Stacking que mejor nos dio en Kaggle se puso un max_features: 7000 (fue el máximo de características que se pudo utilizar sin que se consumiera la RAM de la notebook) y se utilizó el DataFrame previo a eliminar las

críticas en inglés. Y la división de datos utilizada fue de un “test_size”: 0.3 a diferencia de los demás modelos que fue de un “test_size”: 0.2.

Conclusiones generales

Para la parte de Análisis exploratorio, no hubo mucho análisis posible, ya que solo había una característica a analizar (críticas).
Las tareas de preprocesamiento resultaron útiles para poder agilizar y mejorar las predicciones, la que más impacto tuvo fue la de eliminar las críticas en inglés, ya que al hacerlo se consiguió el mejor modelo, el cual fue Bayes Naive, el más sencillo y rápido de entrenar, además de ser el que mejor desempeño tuvo en Kaggle.

Tareas realizadas

Se realizaron todas las tareas de análisis exploratorio y preprocesamiento de datos siempre en grupo por reuniones en meet por lo tanto no hubo una división clara de las mismas (al menos 1 reunión semanal de aproximadamente 2-3 horas). Al concluir el análisis, dividimos los modelos a entrenar en dónde cada integrante del equipo le dedicó cierta cantidad de horas extra semanalmente a realizar los entrenamientos y predicciones correspondientes. Para agilizar la ejecución de los algoritmos se utilizaron copias de la notebook para entrenar modelos en paralelo y luego importarlos desde el original.