Grupo 11
Integrantes:
Daniel León
Jorge Enrique
Juan Andrés Cabal
Juan Fernando Ortiz

Movie Genre Classification

Maestría en Inteligencia Analítica para la Toma de Decisiones

Modelos Avanzados para el Análisis de Datos I

1. Introducción

Como resultado de la expansión en el volumen de datos no estructurados generados en los últimos tiempos, ha surgido el interés de desarrollar herramientas que permitan identificar similitudes en los textos y predecir características a partir de su procesamiento. Con el propósito de avanzar en el conocimiento de estos instrumentos, este documento presenta todos los detalles de la solución, el análisis y las conclusiones de la aplicación de un algoritmo de procesamiento de lenguaje natural (NLP, por sus siglas en inglés) de una base de datos de 11.278 películas sobre las que se cuenta con información acerca de su sinopsis y los géneros en los que fueron clasificadas.

2. Lematización y TF-IDF Vectorizer

Con el propósito de que las palabras utilizadas en cada uno de los 7895 registros de la columna *plot* contenidos en la base de datos de entrenamiento (*dataTraining*) guarden equivalencia entre sí, se crea una función que tiene por objeto (i) convertir la totalidad de caracteres lingüísticos contenidos en la columna *plot* a minúsculas (***text.lower***), (ii) dividir el texto de la columna *plot* en palabras (***text.split***), y; (iii) utilizar el lematizador *wordnet* contenido en la librería NLTK para transformar las palabras, ya divididas, en lemas que guardan cierta correspondencia (***return [wordnet_lemmatizer.lemmatize(word) for word in words]***).

Después de creada la función descrita previamente, por medio del *Tfidfvectorizer* se procesa la base de datos con el propósito de encontrar las partes de texto más relevantes en cada documento (esto es, la columna plot ya lematizada). Bajo este contexto, el Term Frequency-Inverse Document Frequency otorga un mayor peso a los términos que son menos comunes en la totalidad de la base de datos pero que se repiten con mayor frecuencia dentro de uno de los registros (Zhao & Cen, 2014). Como parte de la aplicación de este algoritmo, se le imputan algunos parámetros que (i) eliminan los acentos que no se encuentran clasificados como 'unicode' (`strip_accents='unicode'`), (ii) permiten delimitar el universo de lo que es considerado como *token* al imputarse una expresión regular (`token_pattern=r'\w{2,}'`), (iii) reemplazan la expresión tf de la ecuación TF-IDF a 1+log(tf) (`sublinear_tf=True`), (iv) crean n-gramas de entre 4 y 8 términos, entre otros (ver Ilustración 1). En lo que respecta al parámetro *analyzer*, se le imputa la función de lematización diseñada previamente con el objetivo de calcular la relevancia de cada uno de ellos dentro de cada documento. Además, se eliminan las stop-words por medio del parámetro (`stop_words='english'`).

Ilustración 1. Lematizador y TF-IDF vectorizer

```
import nltk
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()

def split_into_lemmas(text):
    text = text.lower()
    words = text.split()
    return [wordnet_lemmatizer.lemmatize(word) for word in words]

vect = TfidfVectorizer(strip_accents='unicode', token_pattern=r'\w{2,}',sublinear_tf=True,
                       ngram_range=(4, 8),max_df=0.5,analyzer=split_into_lemmas,stop_words='english')
X_dtm = vect.fit_transform(dataTraining['plot'])
```

A modo de ejemplo, la Tabla 1 muestra la matriz TF-IDF para el primer y segundo registro de la base de datos de entrenamiento. De este resultado, es posible observar que el algoritmo extrajo 34611 cadenas de texto de la totalidad de la base de datos, de las cuales *bridge, drawbridge, addict, train* y *gear* son las más relevantes para el primer registro y *satifying, clerk, video, serial* y *teach* son las más relevantes para el segundo.

Tabla 1. Resultados del TF-IDF Vectorizer sobre los dos primeros registros

	tfidf
bridge	0.267898
drawbridge	0.254600
addict	0.221297
train	0.202850
gear	0.201219
...	...
fleury	0.000000
fletcher	0.000000
fletch	0.000000
flesheating	0.000000
ö	0.000000

34611 rows × 1 columns

	tfidf
satisfying	0.485479
clerk	0.358662
video	0.335213
serial	0.329058
teach	0.314150
...	...
fledgling	0.000000
fledged	0.000000
fled	0.000000
fleabag	0.000000
ö	0.000000

34611 rows × 1 columns

3. Estandarización de las bases de datos para la aplicación del modelo

Después de obtenidos los resultados de correr el TFidfvectorizer para la totalidad de la base de datos, y debido a que el ejercicio desarrollado es un problema de multiclasificación, resulta conveniente convertir la variable a predecir en arreglos independientes, por registro, que identifiquen los géneros en los que se clasifica cada película. Después de separada la variable a predecir de las variables predictoras, se utiliza el MultiLabelBinarizer que se encuentra en la librería ScikitLearn (ver Ilustración 2). Esta función arroja un arreglo de ceros y unos de tamaño equivalente

al número de géneros únicos encontrados en la base de datos, que en este caso es 24 (ver Ilustración 3). En este sentido, cada posición que tome el valor de 1 en el arreglo generado por registro representará la pertenencia de la película a cada género. Para efectos de la predicción, se generan la base de datos de prueba con el 33% de los 7.895 registros, esto es, 2.605.

Ilustración 2. MultiLabelBinarizer para la variable a predecir

```
dataTraining['genres'] = dataTraining['genres'].map(lambda x: eval(x))
le = MultiLabelBinarizer()
y_genres = le.fit_transform(dataTraining['genres'])
X_train, X_test, y_train_genres, y_test_genres = train_test_split(X_dtm, y_genres,
                                                                  test_size=0.33, random_state=42)
```

Ilustración 3. Arreglo generado por MultilabelBinarizer

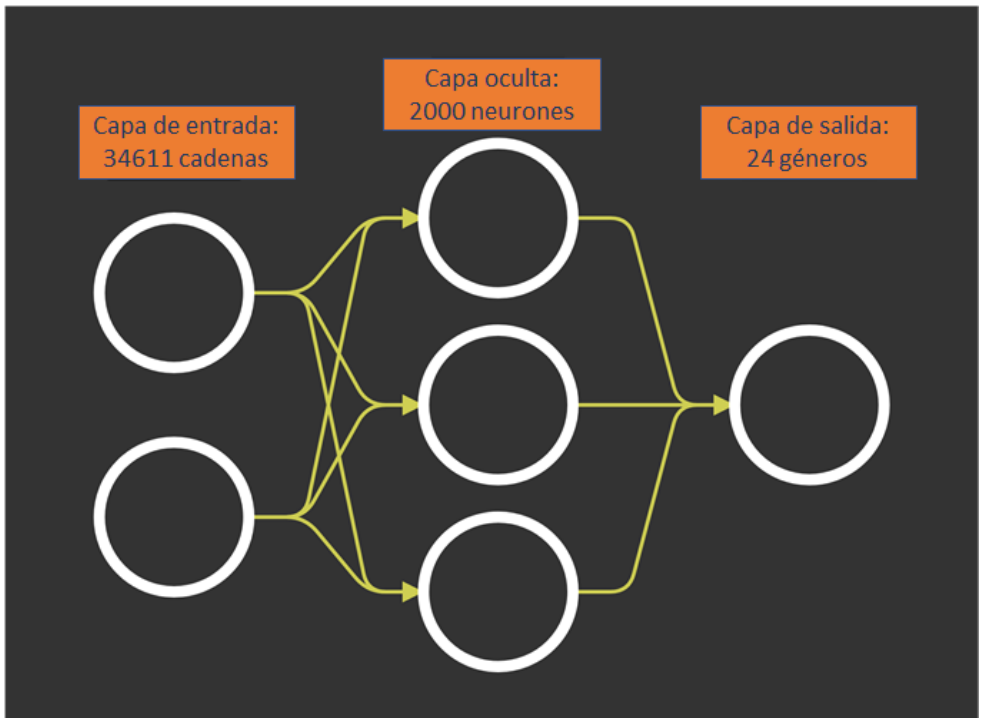
```
[12] y_genres

array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 1, 0, 0],
       ...,
       [0, 1, 0, ..., 0, 0, 0],
       [0, 1, 1, ..., 0, 0, 0],
       [0, 1, 1, ..., 0, 0, 0]])
```

4. Entrenamiento del Modelo

Después de procesar tanto las variables predictoras como la variable a predecir, se crea un modelo secuencial con TensorFlow – Keras que contiene una capa oculta de 2000 neurones. Respecto al parámetro *input_shape* de la primera capa declarada, cabe anotar que corresponde al TF-IDF calculado para cada una de las 34.611 cadenas de texto contenidas en la matriz que se mencionó con anterioridad (ver Tabla 1). Además, la capa de salida definida tiene un tamaño equivalente los 24 géneros contenidos en la base de datos de películas (Ilustración 4).

Ilustración 4. Visualización de la red neuronal a entrenar



En cuanto a la red neuronal descrita previamente, es necesario imputarle una función de activación que permita que los resultados de la capa oculta y de la capa de salida guarden correspondencia con la escala de los valores contenidos en la capa de entrada, que están entre 0 y 1. En función de lo anterior, se utiliza una función de activación exponencial en la capa oculta y una función de activación sigmoide en la capa de salida, lo que permitirá normalizar los resultados obtenidos de aquellas capas a los encontrados en la capa de entrada.

Después de haber diseñado la red neuronal, se configura su proceso de aprendizaje por medio del comando *compile*. Esta rutina tendrá por objeto (i) definir un optimizador, esto es, la función por medio de la que el algoritmo aprenderá, que para este caso es RMSProp, (ii) la función de pérdida, que será la función objetivo que el modelo intentará minimizar y para este caso se utilizará *binary_crossentropy*, y; (iii) una métrica de precisión que, para este caso, será *binary_accuracy* (ver Ilustración 5).

Ilustración 5. Formulación y resultados del modelo

```
#Número de columnas de X_train (34611)
dims = X_train.shape[1]
#Número de columnas en y_train (géneros = 24)
output_var = y_train_genres.shape[1]
K.clear_session()
print("Building model...")
#Iniciación del modelo secuencial
model = Sequential()
#Capa oculta de 2000 neurones, cuya capa de entrada es equivalente a DIMS
model.add(Dense(2000, input_shape=(dims,),activation='exponential'))
#Capa de salida n= 24
model.add(Dense(output_var))
#Función de activación para la capa de salida
model.add(Activation('sigmoid'))
#Entrenamiento del modelo
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['binary_accuracy'])
model.summary()
```

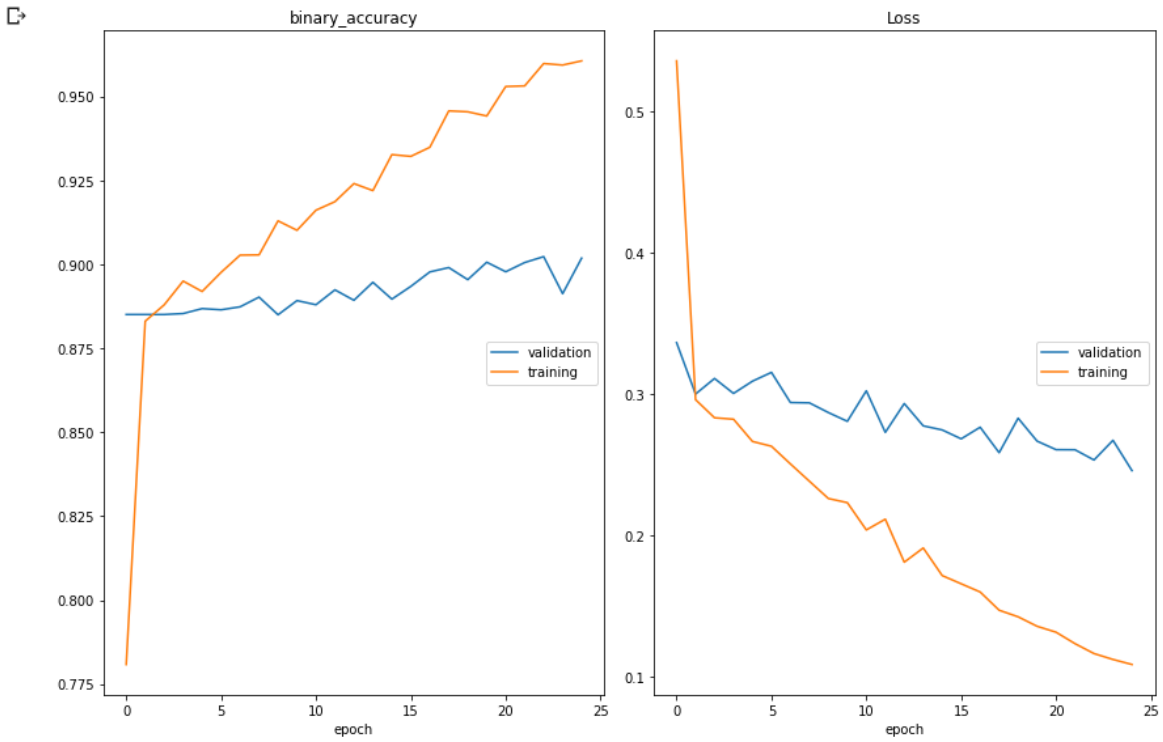
```
Building model...
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 2000)              69224000
_________________________________________________________________
dense_2 (Dense)              (None, 24)                48024
_________________________________________________________________
activation_1 (Activation)    (None, 24)                0
=================================================================
Total params: 69,272,024
Trainable params: 69,272,024
Non-trainable params: 0
_________________________________________________________________
```

Como se observa de los resultados del modelo expuestos en la ilustración anterior, el total de parámetros a estimar será de 69,3 millones, producto de la sumatoria de (i) la multiplicación del número de cadenas de texto de la capa de entrada (34612) y el número de unidades por medio de las que se conecta la capa de entrada (2000), y; (ii) la multiplicación entre el número de unidades en la capa oculta y los parámetros de la capa de salida (24).

Para el entrenamiento del modelo, la variable *epochs* se establece en 25, esto es, el total de veces que el modelo pasará por la base de datos. Para este caso en particular, la base de datos de entrenamiento cuenta con 7.895 registros, lo que implica que el modelo descrito con anterioridad procesará esa cantidad de muestras durante 25 ciclos o iteraciones. De otra parte, se define un *batch_size* de 1000, lo que significa que entrenará el modelo con lotes de datos igual a 1000 unidades. Además, para conocer la manera en que evoluciona la precisión y el resultado de la función de pérdida de los datos de entrenamiento respecto a los datos de prueba, se aplica el objeto *PlotLossesKeras* a la función *callbacks*, que arrojará un gráfico que mostrará aquellos resultados en función de el número de épocas calculadas.

```
model.fit(X_train, y_train_genres, validation_data=[X_test, y_test_genres],
          batch_size=1000, epochs=25, verbose=1,
          callbacks=[PlotLossesKeras()])
```

Al compilar el modelo, se observa cierta convergencia entre los resultados de la función de pérdida de los datos de entrenamiento y los de validación para cada época. De igual manera, a medida que aumentan las épocas se disminuye la pérdida de los datos de validación, hasta llegar a un valor de 0.238. Paralelamente, al evaluar la precisión del modelo se observa que, para los datos de validación, se obtiene una precisión máxima de 0.902, lo que permite concluir que de cada mil películas en los datos de validación cerca de 902 serán pronosticadas correctamente en función de su sinopsis. Además, es posible concluir que la red neuronal aprende de los datos de entrenamiento progresivamente hasta llegar al 96,1%, mientras que la función de pérdida de los datos de entrenamiento tiene un mínimo de 0.109.




```
binary_accuracy
        validation              (min:    0.885, max:    0.902, cur:    0.902)
        training                (min:    0.781, max:    0.961, cur:    0.961)
Loss
        validation              (min:    0.246, max:    0.336, cur:    0.246)
        training                (min:    0.109, max:    0.536, cur:    0.109)
<keras.callbacks.callbacks.History at 0x7fd8f39a1f98>
```

5. Pruebas y Conclusiones

Con el propósito de evaluar la capacidad del modelo para distinguir entre las 24 clases de salida, se calcula el área bajo la curva ROC que arroja un resultado de 0.8942 para los datos de validación. Esto significa que existe ese nivel de probabilidad de que el modelo distinga adecuadamente entre los 24 géneros.

Después de desarrollar el modelo de multiclasificación y de montar la API en Amazon Web Services, se ejecutan pruebas para determinar si el diseño de la red neuronal y sus resultados guardan correspondencia con los géneros de algunas películas escogidas aleatoriamente. A modo de ejemplo, a continuación se presenta el resultado de ejecutar la API con la sinopsis de la película *The Shawshank Redemption* (Ilustración 6)**.** Como se observa, el modelo determina que los géneros a los que pertenecerá con mayor probabilidad esa película son (i) drama, (ii) acción, y; (iii) aventura. Estos resultados son acordes a la sinopsis de la película y a su clasificación en IMDB.

Ilustración 6. Sinopsis y resultado

Two imprisoned men bond over a number of years, finding solace and eventual redemption through acts of common decency.	{
	"result": "{'p_Action': 0.5181384, 'p_Adventure': 0.5181384, 'p_Animation': 0.0009806275, 'p_Biography': 1.4759152e-05, 'p_Comedy': 0.0680561, 'p_Crime': 0.00022548513, 'p_Documentary': 0.004412467, 'p_Drama': 0.4337356, 'p_Family': 0.0014570222, 'p_Fantasy': 4.376687e-05, 'p_Film-Noir': 6.861518e-07, 'p_History': 0.0033839874, 'p_Horror': 2.8710826e-06, 'p_Music': 0.0005178677, 'p_Musical': 0.002762162, 'p_Mystery': 0.0039207614, 'p_News': 5.0060447e-05, 'p_Romance': 0.0071524675, 'p_Sci-Fi': 0.034847185, 'p_Short': 0.0013501149, 'p_Sport': 7.902432e-05, 'p_Thriller': 0.0031070407, 'p_War': 0.0015831612, 'p_Western': 0.0014242167}"
	}

Bibliografía

Gupta, K. (11 de 06 de 2019). *Predicting Movie Genres Based on Plot Summaries*. Obtenido de https://medium.com/@kunalgupta4595/predicting-movie-genres-based-on-plot-summaries-bae646e70e04

JOSHI, P. (s.f.). *Predicting Movie Genres using NLP*. Obtenido de https://www.analyticsvidhya.com/blog/2019/04/predicting-movie-genres-nlp-multi-label-classification/

Keras. (s.f.). *Layer activation functions*. Obtenido de Usage of activations: https://keras.io/api/layers/activations/

Keras. (s.f.). *Optimizers*. Obtenido de Usage with compile() & fit(): https://keras.io/api/optimizers/

Zhao, Y., & Cen, Y. (2014). *Data Mining Applications with R.*