Movie genre prediction

Krystian Wrotniak

July 2023

1 Introduction

The goal of this project was to construct a NN model able to predict a movie genre based on a short description of the movie.

2 Code running

The code consists of two part, each one being a IPYNB file. The "data prep" one provides a proper dataframe to work with, while the "Final Project" is the main one, where the model was born.

To run the code, one needs to use a VS Code application, Google Collab, Jupyter Notebook or other environment capable of opening and running the IPYNB files.

3 Data exploration

Data was provided for me from the Kaggle website. The dataset is without a doubt too big for the capacity of my hardware, even with a help of a GPU from Google Collab. It consists of over 40k entries with multiple columns. This is why some data exploration was at first in order, so that I could limit myself only to the "movie description" and "genre" columns.

After a lot of trying, however, I simply could not get my model to predict all of 20 genres properly. The accuracy kept on reaching 40 percent at best. I then began to narrow down the range of genres to predict (going down to 5-7 genres), reaching up to 55 percent. The genres I chose were the most popular ones.

At some point, while trying to get my model to perform better, I came to an important observation. Even a human person would have great difficulties in distinguishing the great deal of the overviews. For example, how am I to decide whether a movie is of "Comedy" or of "Romance, Comedy" genre? Or is the "Action" genre suppose to be included or not. After all, aren't all movies an "action" ones?

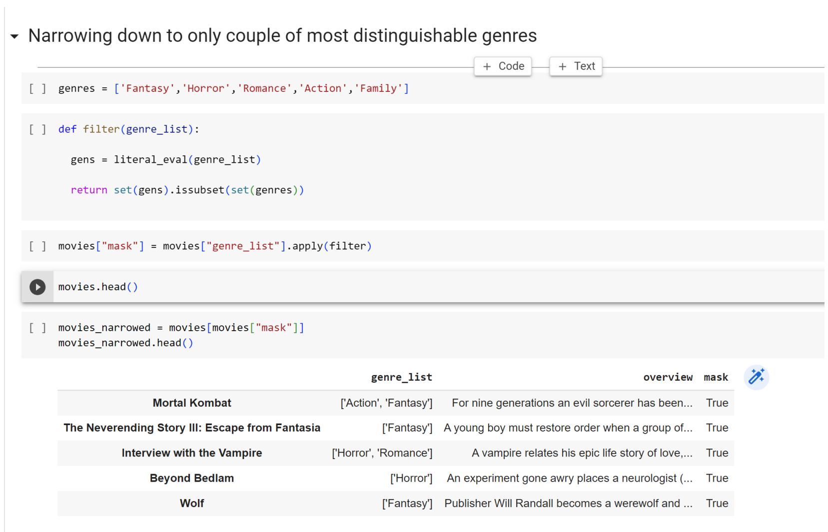

Figure 1: A quick look at the data exploration process

That's why I finally settled for five genres I found the most distinguishable, namely: "Fantasy, Horror, Romance, Action, Family". The choice obviously isn't ideal, but I don't think that any other is.

4 Model building

In building the model I based my hopes on trying the CountVectorizer with various n-grams parameter. This, however, yielded results not at all satisfactory. I kept on trying.

Next, I went to tokenize the description texts and attempted to build an RNN model. I tried with both LSTM and GRU but the results were just embarrassing-I suppose I had to do something wrong.

Finally, I settled for rather a simple model, with a GlobalAveragePooling1D. This gave me the most promising results, so I kept on experimenting with it.

In terms of labeling, I had to settle on the multilabel classification. For that, I searched a couple of websites and discovered MultiLabelBinarizer() as my best suit.

5 Model evaluation

Figure 3. shows the final results of my model. Despite it still not being much close to optimal I honestly am quite happy with the final accuracy. The reason for that I actually already stated before: The task of deciding which SPECIFIC set of genres applies to a movie, especially when based only on its short de-

```
[ ] for i in movies["genre_list"].values:

      labels.append(literal_eval(i))

    # Transform labels

    mlb = MultiLabelBinarizer()
    Y = mlb.fit_transform(labels)

    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)
```

```
[ ] # Tokenizing the Xs

    from tensorflow.keras.preprocessing.text import Tokenizer
    from tensorflow.keras.preprocessing.sequence import pad_sequences

    tokenizer = Tokenizer(oov_token="<OOV>")

    tokenizer.fit_on_texts(X_train)

    word_index = tokenizer.word_index
    vocab_size = len(tokenizer.word_index) + 1

    X_train = tokenizer.texts_to_sequences(X_train)
    X_train = pad_sequences(X_train, padding="post")

    X_test = tokenizer.texts_to_sequences(X_test)
    X_test = pad_sequences(X_test, padding="post")
```

```
[ ] X_train

    array([[ 1068,     5,  3132, ...,     0,     0,     0],
           [ 4893,  4894,  4895, ...,     0,     0,     0],
           [  204,    14,     2, ...,     0,     0,     0],
           ...,
           [  230,   416, 10674, ...,     0,     0,     0],
           [ 4884,     8,   917, ...,     0,     0,     0],
           [    3,   298, 10678, ...,     0,     0,     0]], dtype=int32)
```

Figure 2: Text tokenization and labels encoding

scription, is a surprisingly difficult one. It might probably be due to the actual subjectivity in the process of assigning the genres in the first place.

However, what triggers me, is the behavior of the validation plot. It might suggest a strong overfitting, which I simply cannot get rid of, despite numerous attempts.

I also provide eight prediction tests (Figure 4).

6 Conclusions

All my conclusions I have already mentioned in the earlier sections. The model is not good, definitely, but as far as the difficulty of the very task goes, along with the length of time I spent on trying to make it work, I am frankly quite satisfied with the performance. It surely does not deserve an applause, but I could consider rewarding it with a slight pat on the back.

Training

```
# BUILD AND FIT A MODEL
model = build_model(embed_dim=256, perceptrons=256, hidden_layers=1, LR=0.01)
loss_accuracy = fit_model(model, X_train, Y_train, epochs=10, validation_split=0.2)[0]
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 178, 256)          2738688

 global_average_pooling1d_1   (None, 256)              0
 (GlobalAveragePooling1D)

 dense_2 (Dense)             (None, 256)               65792

 dense_3 (Dense)             (None, 5)                 1285

=================================================================
Total params: 2,805,765
Trainable params: 2,805,765
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
30/30 - 8s - loss: 0.4495 - accuracy: 0.5735 - val_loss: 0.4537 - val_accuracy: 0.5333 - 8s/epoch - 250ms/step
Epoch 2/10
30/30 - 7s - loss: 0.3564 - accuracy: 0.6142 - val_loss: 0.3580 - val_accuracy: 0.5958 - 7s/epoch - 219ms/step
Epoch 3/10
30/30 - 5s - loss: 0.2134 - accuracy: 0.7821 - val_loss: 0.3424 - val_accuracy: 0.6958 - 5s/epoch - 173ms/step
Epoch 4/10
30/30 - 5s - loss: 0.1151 - accuracy: 0.8738 - val_loss: 0.3738 - val_accuracy: 0.6875 - 5s/epoch - 168ms/step
Epoch 5/10
30/30 - 5s - loss: 0.0538 - accuracy: 0.9062 - val_loss: 0.3906 - val_accuracy: 0.7208 - 5s/epoch - 160ms/step
Epoch 6/10
30/30 - 3s - loss: 0.0234 - accuracy: 0.9187 - val_loss: 0.5162 - val_accuracy: 0.6667 - 3s/epoch - 92ms/step
Epoch 7/10
```

Figure 3: The training process.

Plotting and testing

```
# PLOT TRAINING HISTORY
plot_loss_accuracy(loss_accuracy.history, validation=True)

# TEST THE MODEL
loss, accuracy = model.evaluate(X_test, Y_test, verbose=0)
print('Test loss:', loss)
print('Test accuracy:', accuracy)
```

```
Test loss: 0.509013831615448
Test accuracy: 0.7198443412780762
```

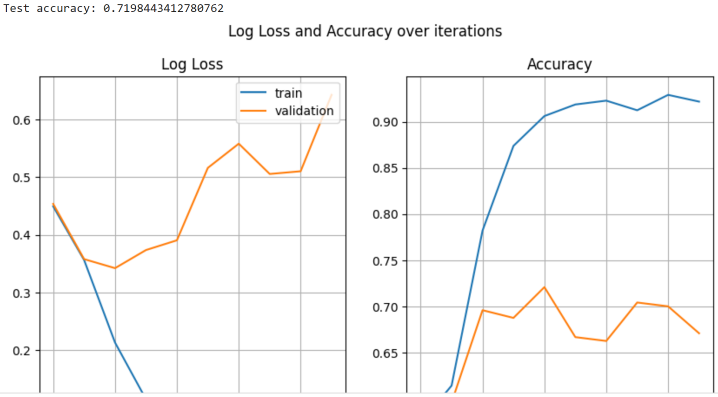

Figure 4: The result of the training. Not the greatest, but could be worse, speaking from experience

```
# Test 2

i = 561

overview = movies["overview"].values[i]

overview = tokenizer.texts_to_sequences(overview)
overview = pad_sequences(overview,padding="post")

prediction = model.predict(overview)

print("Prediction: ", mlb.inverse_transform(np.round(prediction))[0])
print("True value: ", movies["genre_list"].values[i])
```

```
16/16 [==============================] - 0s 2ms/step
Prediction:  ('Fantasy', 'Horror')
True value:  ['Horror']
```

```
[ ] # Test 3

i = 676

overview = movies["overview"].values[i]

overview = tokenizer.texts_to_sequences(overview)
overview = pad_sequences(overview,padding="post")

prediction = model.predict(overview)

print("Prediction: ", mlb.inverse_transform(np.round(prediction))[0])
print("True value: ", movies["genre_list"].values[i])
```

```
4/4 [==============================] - 0s 3ms/step
Prediction:  ('Horror',)
True value:  ['Horror']
```

Figure 5: Couple of prediction tests.