Computer Vision 2

Neural Network with Multi-Label Classification

Created by: **Abhishek Narvekar**

Student Id**: 649744**

Email**: 649744@student.inholland.nl**

Lecturer: **Ms. Marya Brutt**

University name: **Inholland University Of Applied Sciences**

Date: 29/06/2023

Contents

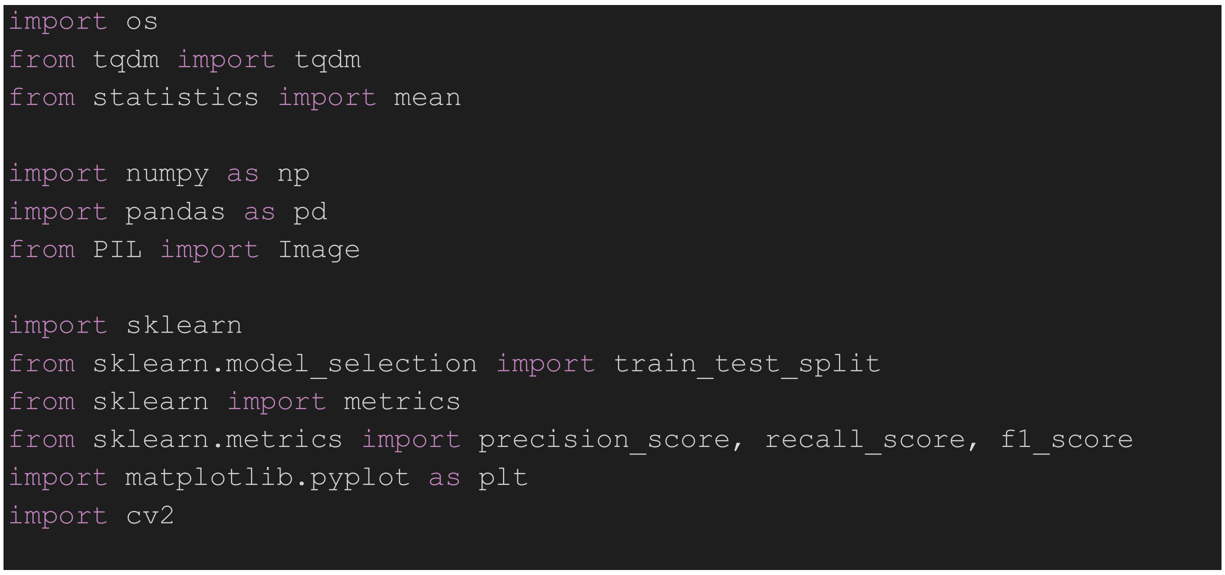
Section 1: Introduction to the dataset

For performing Multi-Label classification, the dataset is taken from the GitHub repository: https://github.com/laxmimerit/Movies-Poster_Dataset which is based on a recommendation from my computer vision teacher. This repository contains one image folder. This folder contains the images of movie posters, in .jpg format. It also contains a train.csv file. This file contains the information of the genres of a movie based of its poster image. The dataset contains 7254 rows and 27 columns in total. This information was obtained by executing the command: 'movieDF.shape', where 'movieDF' stands for the data frame, where we load the file in Python using the panda's library and the .shape method gives out the information on the number of rows and columns present inside the dataset.

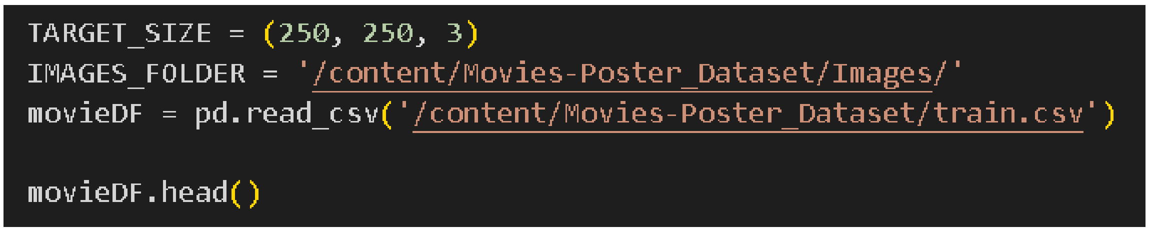
The 'train.csv' file contains the 'Id' column, which corresponds to the image names found within the images folder in the dataset. The 'Genre' column in the file enumerates the various genres encompassed by each movie poster images. The other remaining columns represent all the distinct genre names indicating the presence or absence of each respective genre within a specific movie poster image. A value of one is assigned underneath that particular genre column if the movie contains that genre, otherwise, the value assigned is zero indicating that the movie does not contain that particular genre.

The data type of the 'Id' and the 'Genre' column is of type 'object', and the other columns which are the individual genres are of datatype 'int64' since we either have a zero or one on that particular column. The following information was obtained using the 'movieDF.info()' command mentioned in the screenshots below. The total size of the dataset is 519.658.125 bytes which is about 495MB. These values were obtained by first cloning the repository locally and then right-clicking on the folder name to get the appropriate size of the dataset.

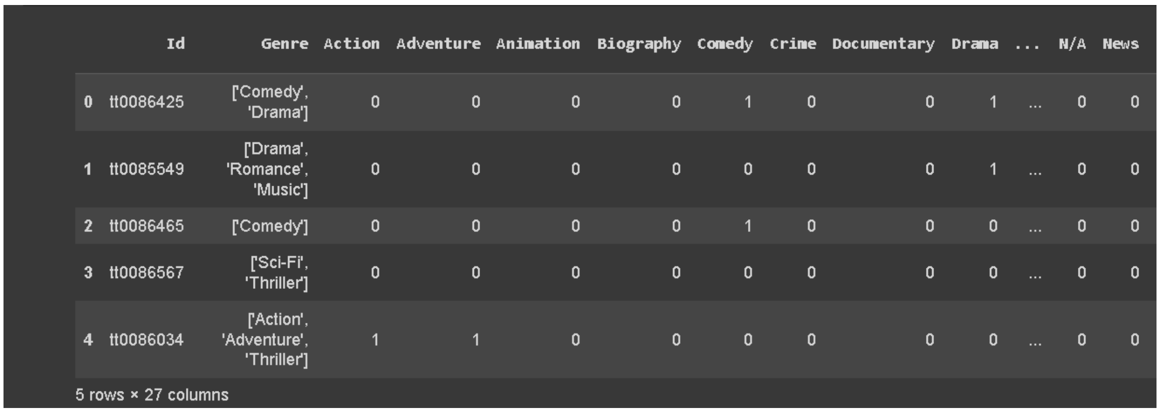
I used the libraries os, tqdm, numpy, pandas sklearn, keras, keras tuner and tensorflow. All the appropriate classes required for performing he multi-label classification in mentioned in the screenshot below.


```
import os
from tqdm import tqdm
from statistics import mean

import numpy as np
import pandas as pd
from PIL import Image

import sklearn
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import precision_score, recall_score, f1_score
import matplotlib.pyplot as plt
import cv2
import keras
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.preprocessing import image
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras import models
from keras import layers
from keras import optimizers
from keras.preprocessing.image import ImageDataGenerator

import tensorflow as tf
from tensorflow.keras.applications import VGG16

import keras_tuner
from keras_tuner.tuners import BayesianOptimization
from keras_tuner.engine import objective as kt_objective
```

The screenshot of the code and the outputs for the information of the dataset are mentioned below:

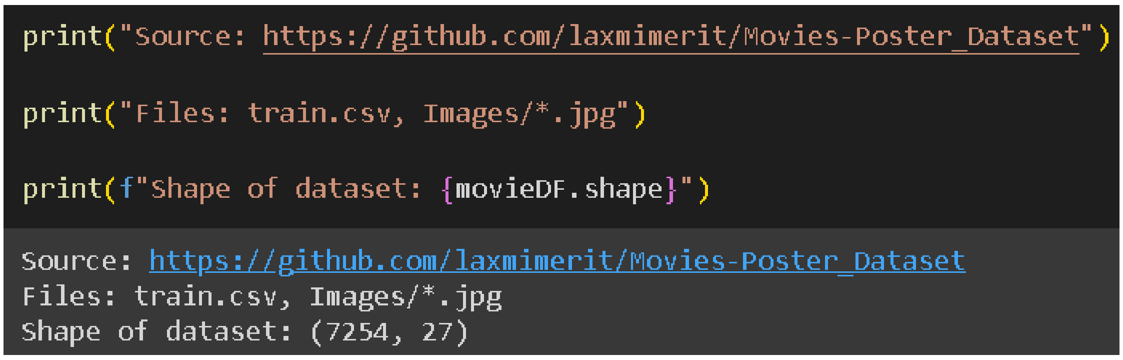


```
TARGET_SIZE = (250, 250, 3)
IMAGES_FOLDER = '/content/Movies-Poster_Dataset/Images/'
movieDF = pd.read_csv('/content/Movies-Poster_Dataset/train.csv')

movieDF.head()
```



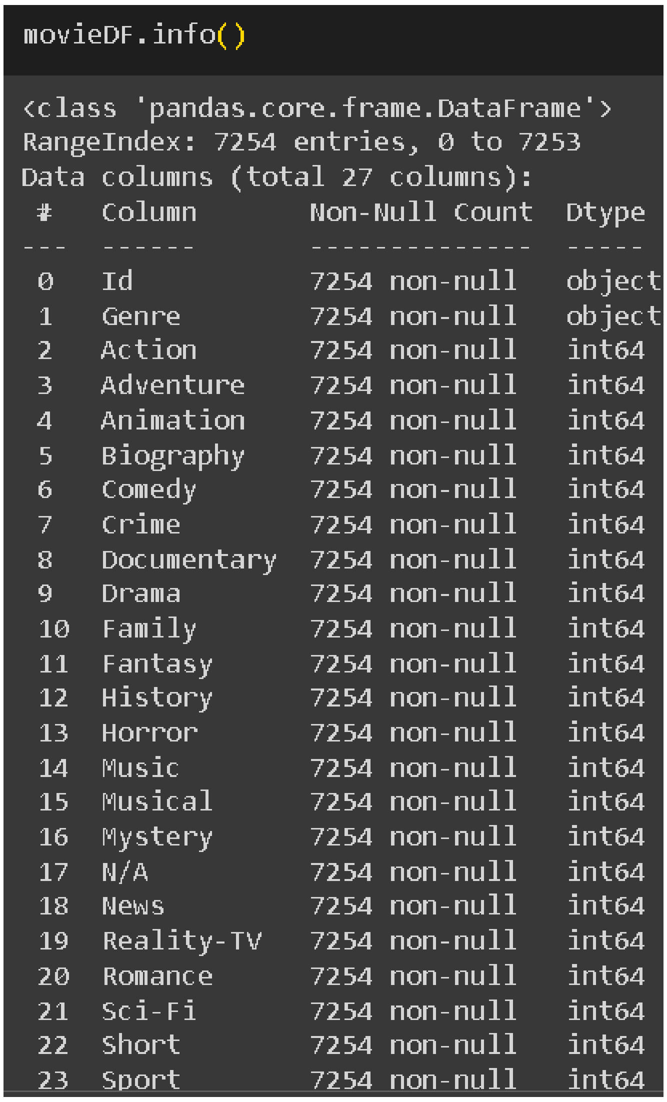
	Id	Genre	Action	Adventure	Animation	Biography	Comedy	Crime	Documentary	Drama	...	N/A	News
0	tt0086425	['Comedy', 'Drama']	0	0	0	0	1	0	0	1	...	0	0
1	tt0085549	['Drama', 'Romance', 'Music']	0	0	0	0	0	0	0	1	...	0	0
2	tt0086465	['Comedy']	0	0	0	0	1	0	0	0	...	0	0
3	tt0086567	['Sci-Fi', 'Thriller']	0	0	0	0	0	0	0	0	...	0	0
4	tt0086034	['Action', 'Adventure', 'Thriller']	1	1	0	0	0	0	0	0	...	0	0

5 rows × 27 columns



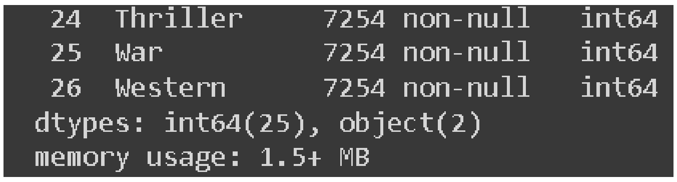
```
print("Source: https://github.com/laxmimerit/Movies-Poster_Dataset")

print("Files: train.csv, Images/*.jpg")

print(f"Shape of dataset: {movieDF.shape}")
```

```
Source: https://github.com/laxmimerit/Movies-Poster_Dataset
Files: train.csv, Images/*.jpg
Shape of dataset: (7254, 27)
```



```python
movieDF.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7254 entries, 0 to 7253
Data columns (total 27 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Id           7254 non-null   object
 1   Genre        7254 non-null   object
 2   Action       7254 non-null   int64
 3   Adventure    7254 non-null   int64
 4   Animation    7254 non-null   int64
 5   Biography    7254 non-null   int64
 6   Comedy       7254 non-null   int64
 7   Crime        7254 non-null   int64
 8   Documentary  7254 non-null   int64
 9   Drama        7254 non-null   int64
 10  Family       7254 non-null   int64
 11  Fantasy      7254 non-null   int64
 12  History      7254 non-null   int64
 13  Horror       7254 non-null   int64
 14  Music        7254 non-null   int64
 15  Musical      7254 non-null   int64
 16  Mystery      7254 non-null   int64
 17  N/A          7254 non-null   int64
 18  News         7254 non-null   int64
 19  Reality-TV   7254 non-null   int64
 20  Romance      7254 non-null   int64
 21  Sci-Fi       7254 non-null   int64
 22  Short        7254 non-null   int64
 23  Sport        7254 non-null   int64
```



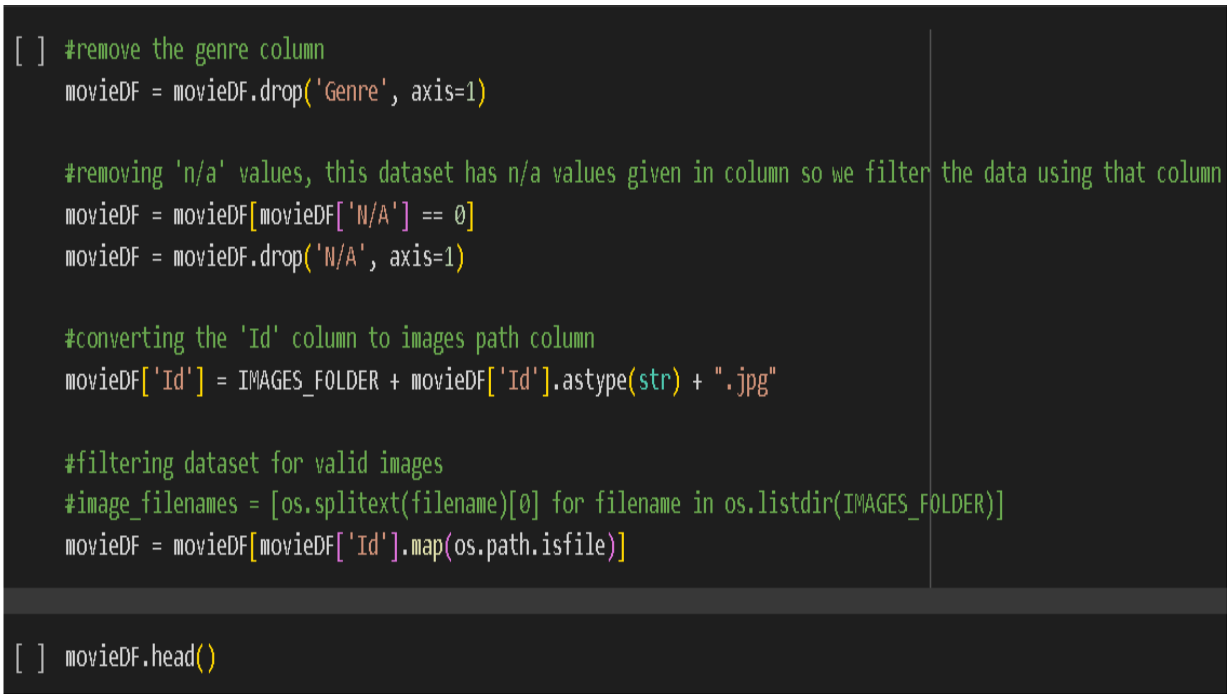
```
 24  Thriller     7254 non-null   int64
 25  War          7254 non-null   int64
 26  Western      7254 non-null   int64
dtypes: int64(25), object(2)
memory usage: 1.5+ MB
```

Section 2: Preprocessing and Setting up the Dataset

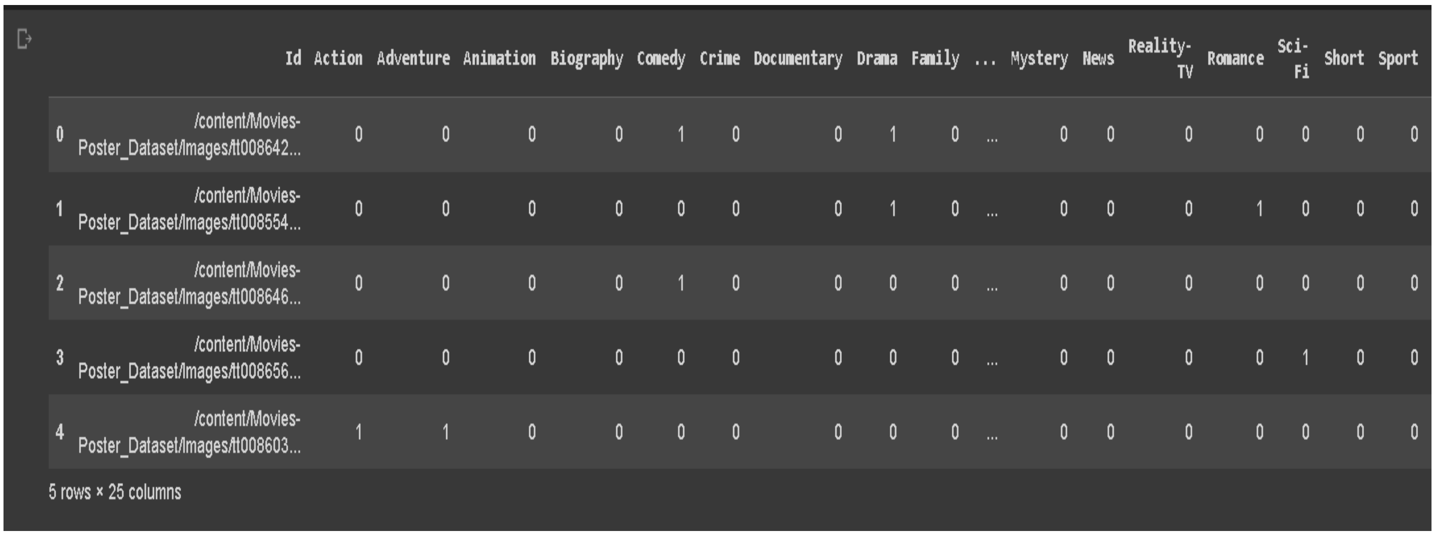
I used a dataset from the GitHub repository since I faced some issues in binding the correct metadata for each movie poster. The dataset itself did not particularly required a lot of preprocessing to be done, however there were some things inside the dataset that weren't particularly important to perform mulit-label classification, so I had to remove them. There were also some things that were needed to be added.

At first, I removed the 'Genre' and the 'N/A' column from the dataset since this was not particularly required to perform multi-label classification. Then I converted the 'Id' column in the dataset to the image path such that each image path inside the image folder is bound to the dataset based on its unique Image Id. After this I resized the images into 250 by 250 to make the images of the same size. Then I split my dataset into training testing and validation files. I used 70% of the data from the github repository for training the model on this data, 15% of the data for testing the model on new data, and the rest 15% of data for validation.

The screenshot of the code and the output are mentioned below:

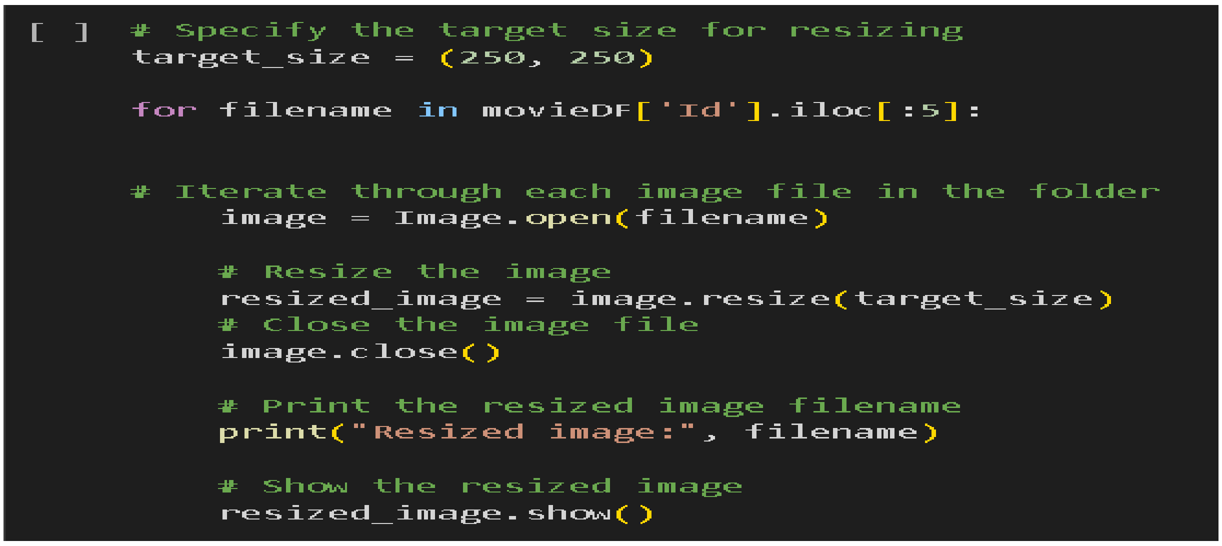


```
[ ] #remove the genre column
    movieDF = movieDF.drop('Genre', axis=1)

    #removing 'n/a' values, this dataset has n/a values given in column so we filter the data using that column
    movieDF = movieDF[movieDF['N/A'] == 0]
    movieDF = movieDF.drop('N/A', axis=1)

    #converting the 'Id' column to images path column
    movieDF['Id'] = IMAGES_FOLDER + movieDF['Id'].astype(str) + ".jpg"

    #filtering dataset for valid images
    #image_filenames = [os.splitext(filename)[0] for filename in os.listdir(IMAGES_FOLDER)]
    movieDF = movieDF[movieDF['Id'].map(os.path.isfile)]

[ ] movieDF.head()
```

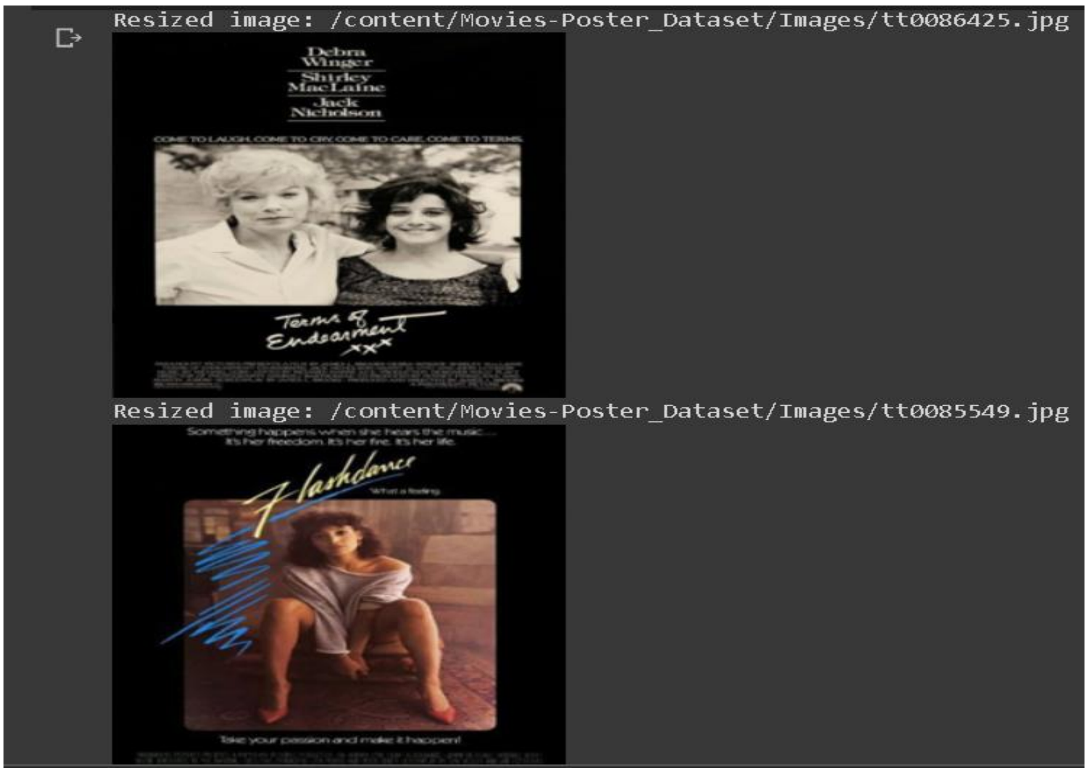


	Id	Action	Adventure	Animation	Biography	Comedy	Crime	Documentary	Drama	Family	...	Mystery	News	Reality-TV	Romance	Sci-Fi	Short	Sport
0	/content/Movies-Poster_Dataset/Images/tt008642...	0	0	0	0	1	0	0	1	0	...	0	0	0	0	0	0	0
1	/content/Movies-Poster_Dataset/Images/tt008554...	0	0	0	0	0	0	0	1	0	...	0	0	0	1	0	0	0
2	/content/Movies-Poster_Dataset/Images/tt008646...	0	0	0	0	1	0	0	0	0	...	0	0	0	0	0	0	0
3	/content/Movies-Poster_Dataset/Images/tt008656...	0	0	0	0	0	0	0	0	0	...	0	0	0	0	1	0	0
4	/content/Movies-Poster_Dataset/Images/tt008603...	1	1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0

5 rows × 25 columns



```
[ ]  # Specify the target size for resizing
     target_size = (250, 250)

     for filename in movieDF['Id'].iloc[:5]:


     # Iterate through each image file in the folder
         image = Image.open(filename)

         # Resize the image
         resized_image = image.resize(target_size)
         # Close the image file
         image.close()

         # Print the resized image filename
         print("Resized image:", filename)

         # Show the resized image
         resized_image.show()
```

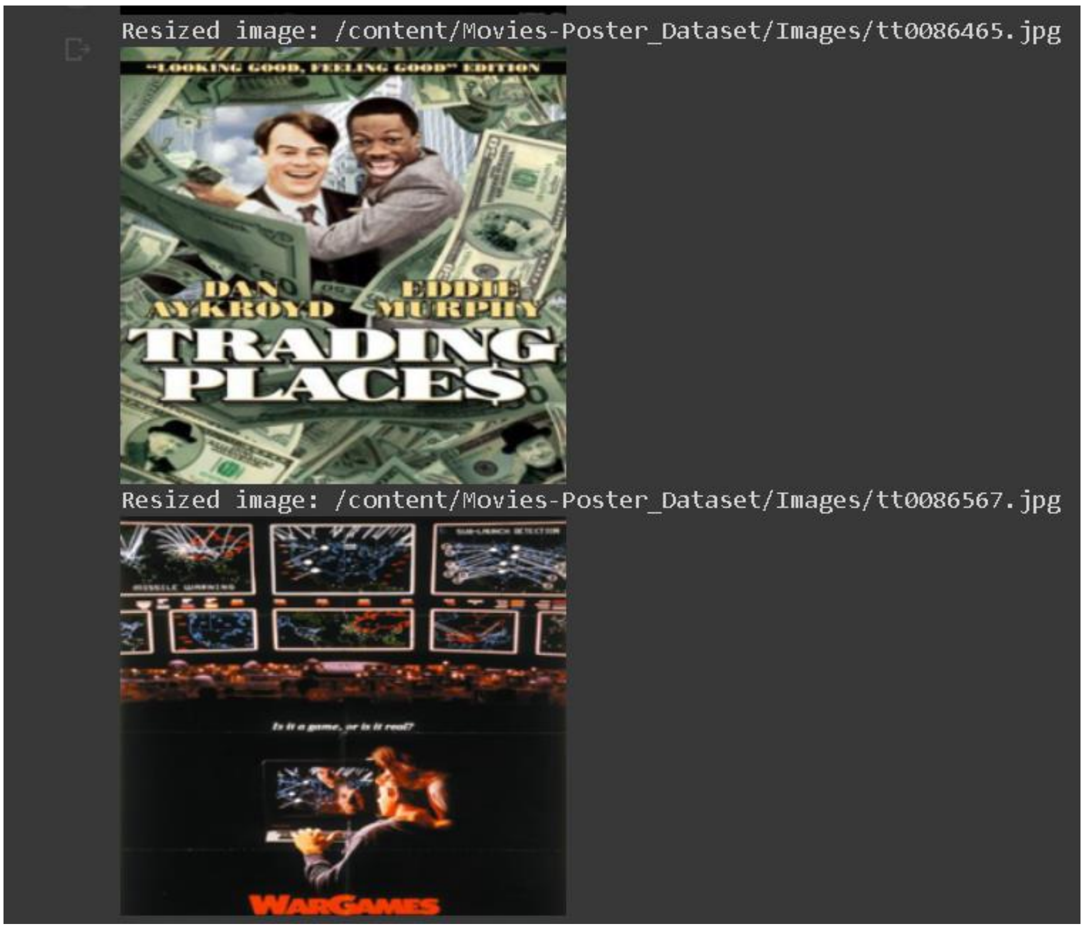


```
Resized image: /content/Movies-Poster_Dataset/Images/tt0086425.jpg
```

Debra Winger
Shirley MacLaine
Jack Nicholson
COME TO LAUGH, COME TO CRY, COME TO CARE, COME TO TERMS.
Terms of Endearment

```
Resized image: /content/Movies-Poster_Dataset/Images/tt0085549.jpg
```

Something happens when she hears the music... It's her freedom. It's her fire. It's her life.
Flashdance
What a feeling.
Take your passion and make it happen!

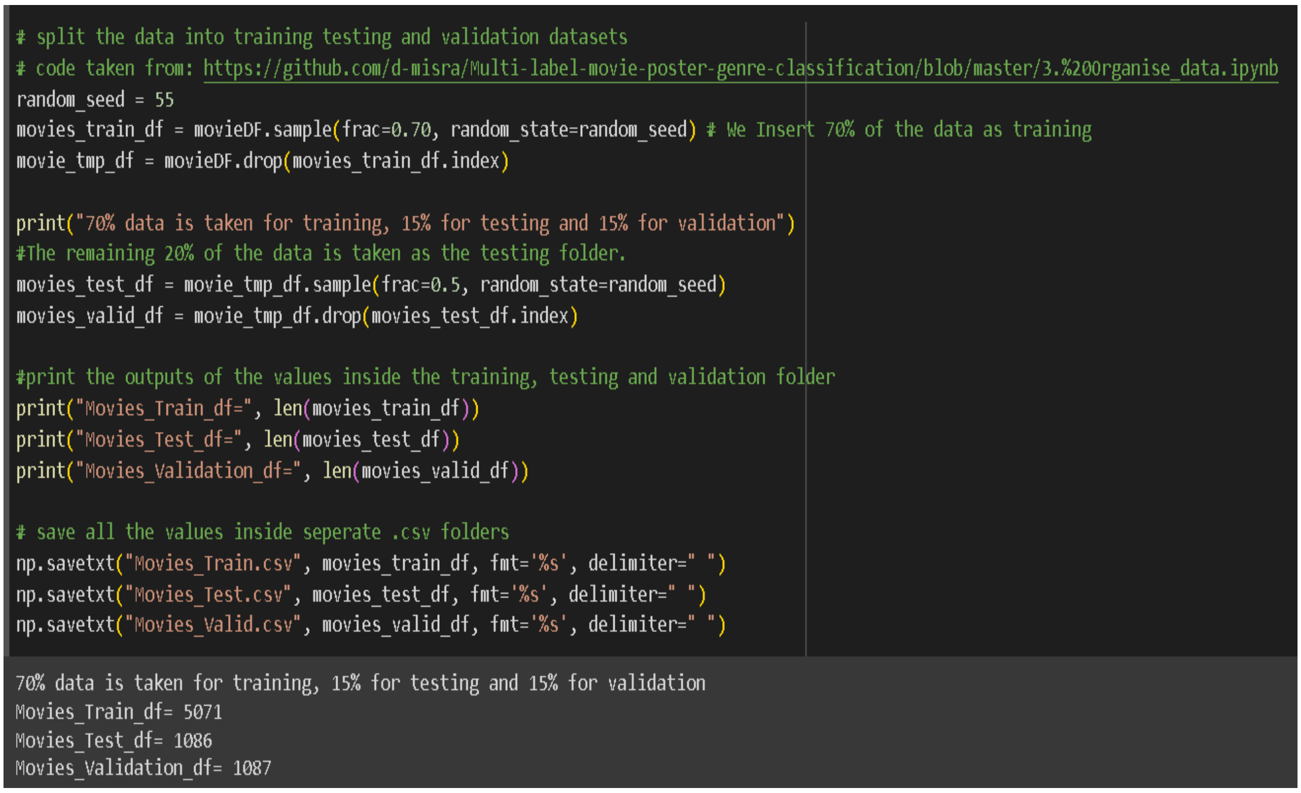


```
Resized image: /content/Movies-Poster_Dataset/Images/tt0086465.jpg
```

"LOOKING GOOD, FEELING GOOD" EDITION
DAN AYKROYD
EDDIE MURPHY
TRADING PLACES

```
Resized image: /content/Movies-Poster_Dataset/Images/tt0086567.jpg
```

MISSILE WARNING
Is it a game, or is it real?
WARGAMES



```
# split the data into training testing and validation datasets
# code taken from: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/3.%20Organise_data.ipynb
random_seed = 55
movies_train_df = movieDF.sample(frac=0.70, random_state=random_seed) # We Insert 70% of the data as training
movie_tmp_df = movieDF.drop(movies_train_df.index)

print("70% data is taken for training, 15% for testing and 15% for validation")
#The remaining 20% of the data is taken as the testing folder.
movies_test_df = movie_tmp_df.sample(frac=0.5, random_state=random_seed)
movies_valid_df = movie_tmp_df.drop(movies_test_df.index)

#print the outputs of the values inside the training, testing and validation folder
print("Movies_Train_df=", len(movies_train_df))
print("Movies_Test_df=", len(movies_test_df))
print("Movies_Validation_df=", len(movies_valid_df))

# save all the values inside seperate .csv folders
np.savetxt("Movies_Train.csv", movies_train_df, fmt='%s', delimiter=" ")
np.savetxt("Movies_Test.csv", movies_test_df, fmt='%s', delimiter=" ")
np.savetxt("Movies_Valid.csv", movies_valid_df, fmt='%s', delimiter=" ")
```

```
70% data is taken for training, 15% for testing and 15% for validation
Movies_Train_df= 5071
Movies_Test_df= 1086
Movies_Validation_df= 1087
```

Section 3: Designing the sequential model

In this section, I designed two models the first one is the custom-created model using sequential class in Keras library and the other model is the pre-trained model VGG-16. I added the pre-trained model as the base model in the sequential class. The choices of optimizers, number of filters, window size, activation functions etc.. are discussed in the sections below.

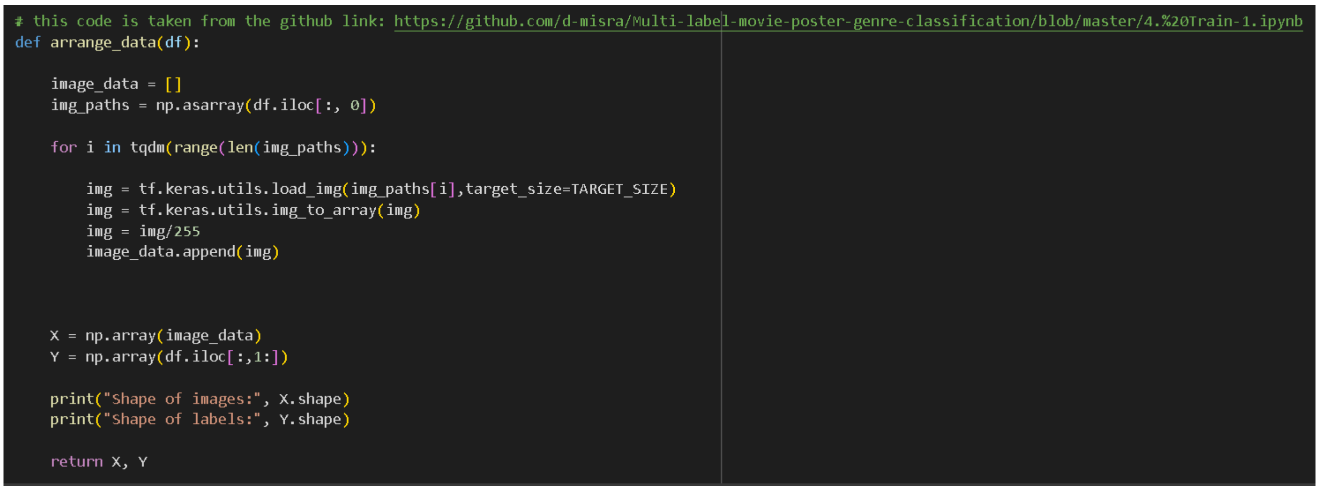
Custom created Model

In this model, we are using Convolution2D, MaxPooling followed by Dense Neural Layers. When designing our model we are using the Convolution2D and max-pooling 2d layers, four times within our model. I also added six dropout layers and three dense layers. The convolution2D layer applies filters to our data to highlight the features inside the data. The max-pooling2d layer helps in capturing the most prominent features within our poster images, therefore, reducing computational complexity. I used 'ReLU' as an activation function in the convolution2D network because it introduces non-linearities to the network, which helps our model to efficiently learn complex relationships. The kernel size used is 5x5 which is small because I am only looking for overall contrast within a movie poster image to predict its genre and not particularly the explicit features inside the image of a movie poster.

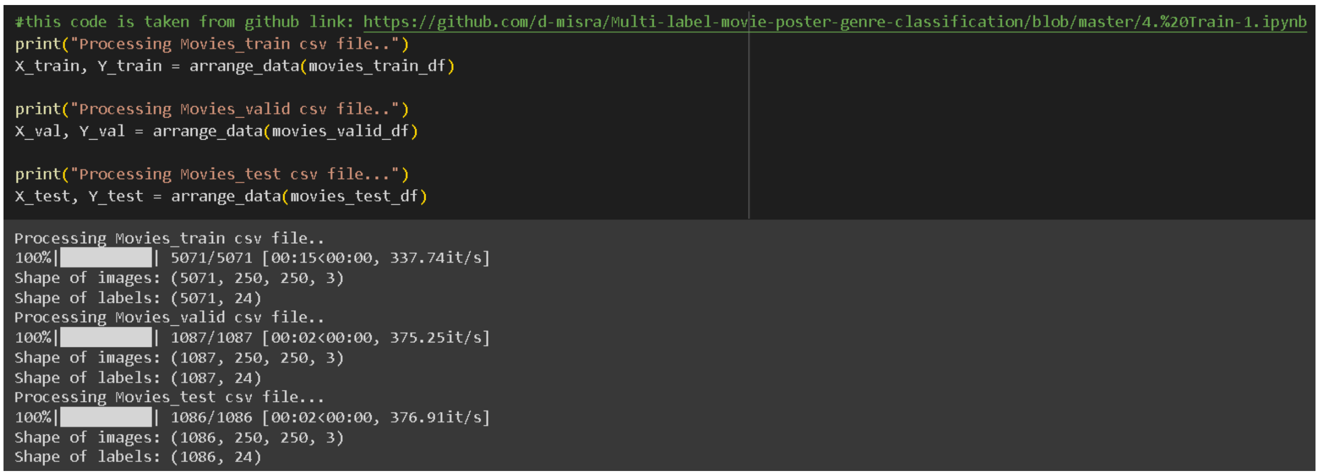
The dense layer is used for extracting and transforming relevant features from the movie posters, capturing the non-relationships, and make predictions on the genre of a movie based on relevant features within the image. The activation function used in dense layer is 'sigmoid', because this function is able to obtain the independent probabilities of each unique genre for a particular movie poster. The dropout layers are added to elevate the performance of the model on new and unseen data, and avoid overfitting of our model. In max-pooling 2d layer I decided to use the pool size two by two because this helps in down sampling some feature maps, which helps in capturing important patterns within our model.

In compiling our model I decided to use the ‘adam’ optimizer because it combines the benefits of AdaGrad and RMSProp and provides an efficient convergence and generalization performance. As a loss function, I used binary cross entropy since this function is efficient in handling problems related to multi-label classification and motivates the model to predict the correct appropriate classes. For metrics, I decided to use the precision metric over accuracy because precision allows us to measure the model performance on correctly predicting the genres of a movie based on its poster, irrespective of whether the other genres were predicted correctly or not. Precision also helps in measuring the genres of a movie that could a bit difficult to predict based on the given independent variables.

The full layer architecture of the custom-created sequential model with the code and the output is mentioned below:

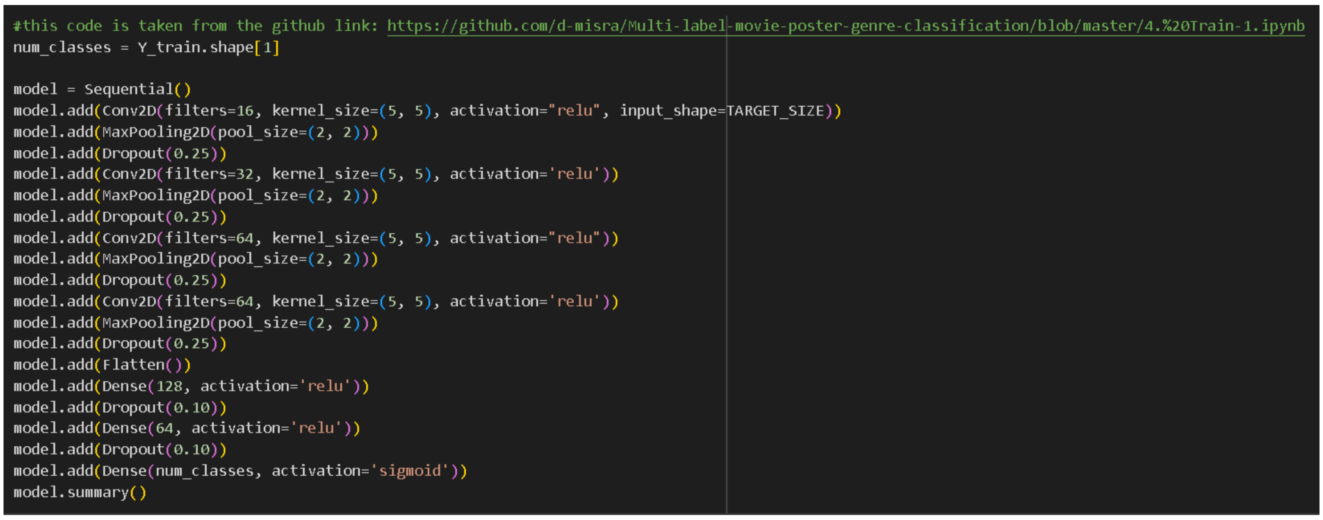


```
# this code is taken from the github link: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/4.%20Train-1.ipynb
def arrange_data(df):

    image_data = []
    img_paths = np.asarray(df.iloc[:, 0])

    for i in tqdm(range(len(img_paths))):

        img = tf.keras.utils.load_img(img_paths[i],target_size=TARGET_SIZE)
        img = tf.keras.utils.img_to_array(img)
        img = img/255
        image_data.append(img)



    X = np.array(image_data)
    Y = np.array(df.iloc[:,1:])

    print("Shape of images:", X.shape)
    print("Shape of labels:", Y.shape)

    return X, Y
```

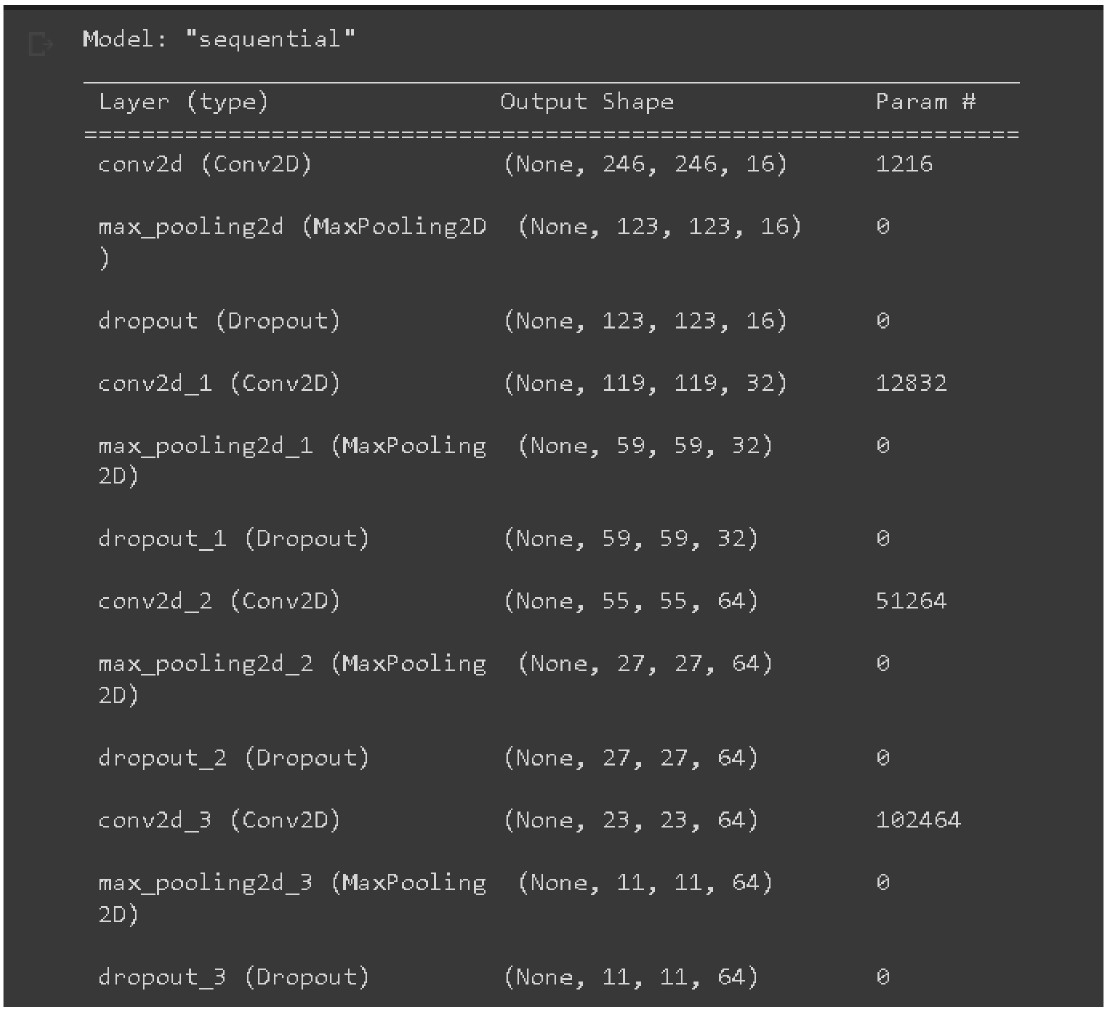


```
#this code is taken from github link: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/4.%20Train-1.ipynb
print("Processing Movies_train csv file..")
X_train, Y_train = arrange_data(movies_train_df)

print("Processing Movies_valid csv file..")
X_val, Y_val = arrange_data(movies_valid_df)

print("Processing Movies_test csv file...")
X_test, Y_test = arrange_data(movies_test_df)
```

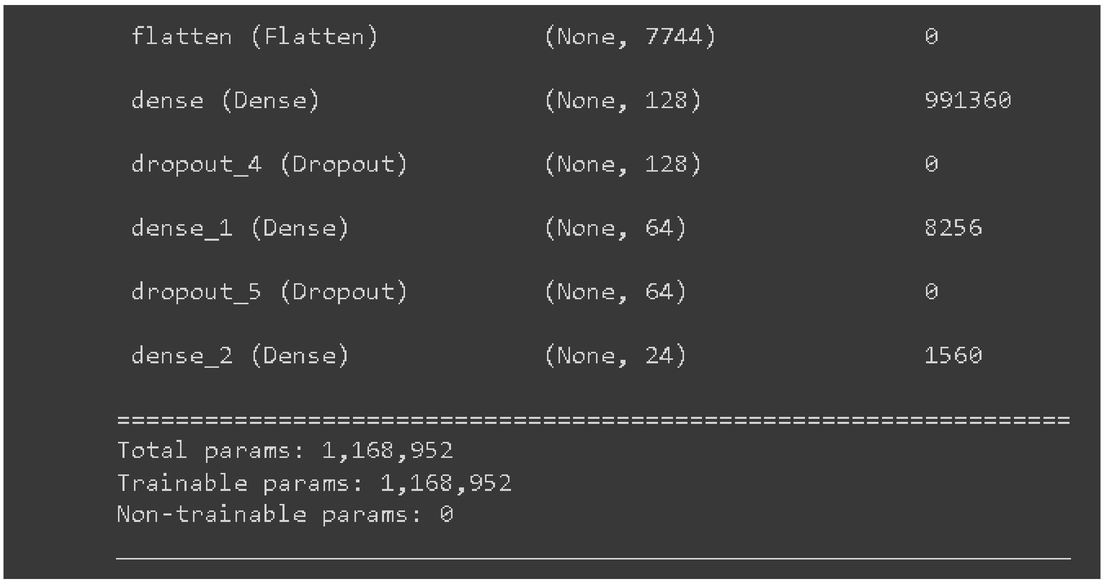
```
Processing Movies_train csv file..
100%|██████████| 5071/5071 [00:15<00:00, 337.74it/s]
Shape of images: (5071, 250, 250, 3)
Shape of labels: (5071, 24)
Processing Movies_valid csv file..
100%|██████████| 1087/1087 [00:02<00:00, 375.25it/s]
Shape of images: (1087, 250, 250, 3)
Shape of labels: (1087, 24)
Processing Movies_test csv file...
100%|██████████| 1086/1086 [00:02<00:00, 376.91it/s]
Shape of images: (1086, 250, 250, 3)
Shape of labels: (1086, 24)
```



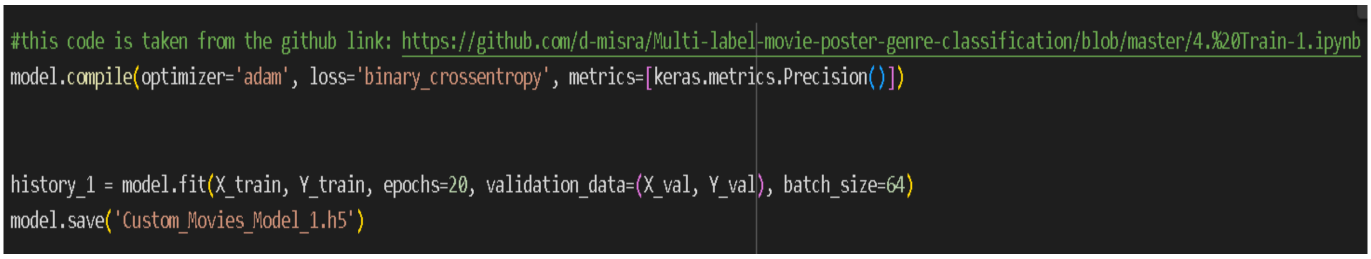
```
#this code is taken from the github link: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/4.%20Train-1.ipynb
num_classes = Y_train.shape[1]

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=(5, 5), activation="relu", input_shape=TARGET_SIZE))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(filters=32, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(filters=64, kernel_size=(5, 5), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(filters=64, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.10))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.10))
model.add(Dense(num_classes, activation='sigmoid'))
model.summary()
```



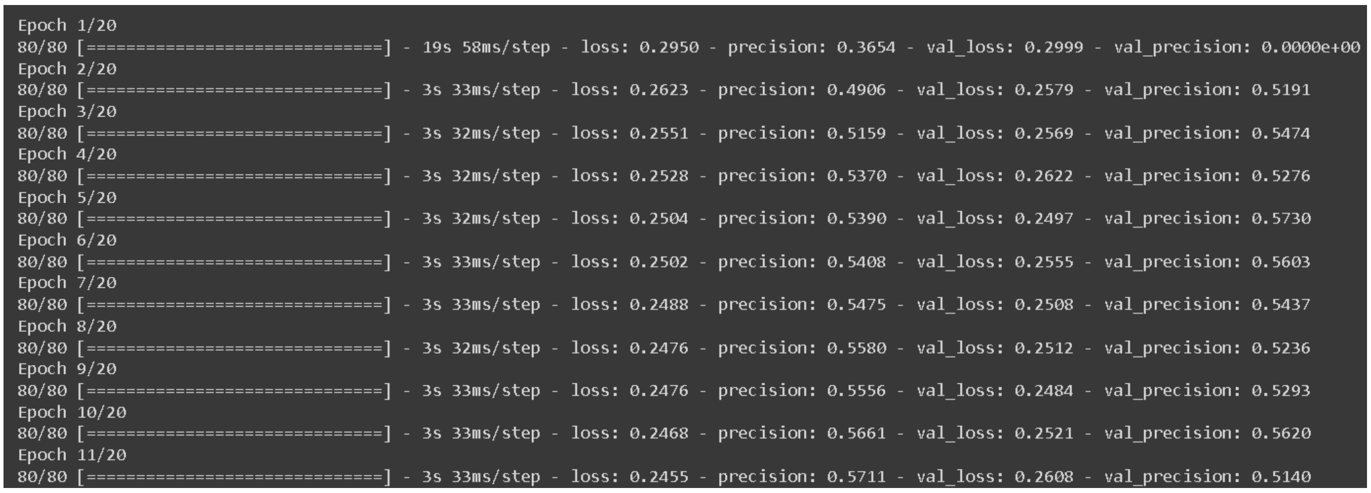
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 246, 246, 16)      1216

 max_pooling2d (MaxPooling2D  (None, 123, 123, 16)     0
 )

 dropout (Dropout)           (None, 123, 123, 16)      0

 conv2d_1 (Conv2D)           (None, 119, 119, 32)      12832

 max_pooling2d_1 (MaxPooling  (None, 59, 59, 32)       0
 2D)

 dropout_1 (Dropout)         (None, 59, 59, 32)        0

 conv2d_2 (Conv2D)           (None, 55, 55, 64)        51264

 max_pooling2d_2 (MaxPooling  (None, 27, 27, 64)       0
 2D)

 dropout_2 (Dropout)         (None, 27, 27, 64)        0

 conv2d_3 (Conv2D)           (None, 23, 23, 64)        102464

 max_pooling2d_3 (MaxPooling  (None, 11, 11, 64)       0
 2D)

 dropout_3 (Dropout)         (None, 11, 11, 64)        0
```



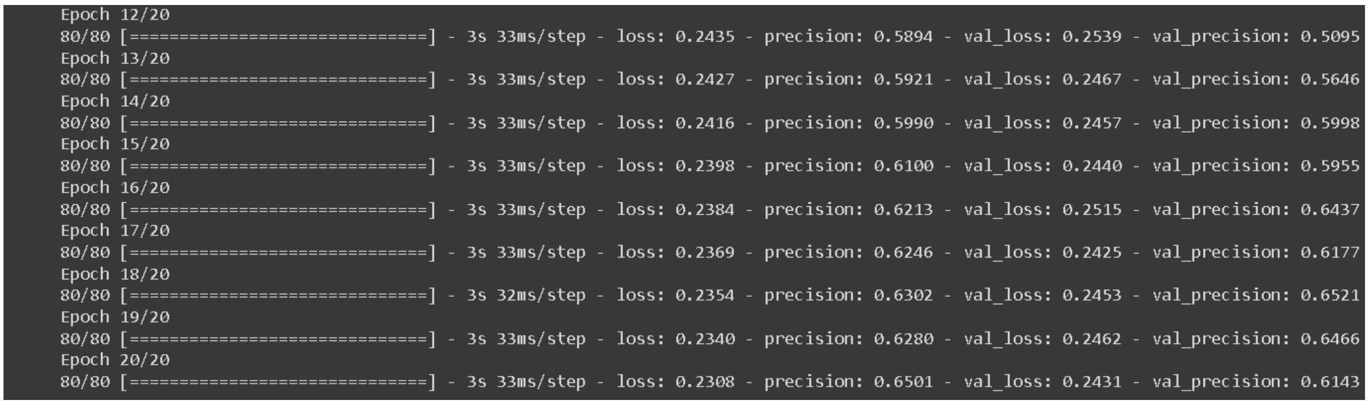
```
 flatten (Flatten)           (None, 7744)              0

 dense (Dense)               (None, 128)               991360

 dropout_4 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 64)                8256

 dropout_5 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 24)                1560

=================================================================
Total params: 1,168,952
Trainable params: 1,168,952
Non-trainable params: 0
_________________________________________________________________
```



```
#this code is taken from the github link: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/4.%20Train-1.ipynb
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[keras.metrics.Precision()])


history_1 = model.fit(X_train, Y_train, epochs=20, validation_data=(X_val, Y_val), batch_size=64)
model.save('Custom_Movies_Model_1.h5')
```



```
Epoch 1/20
80/80 [==============================] - 19s 58ms/step - loss: 0.2950 - precision: 0.3654 - val_loss: 0.2999 - val_precision: 0.0000e+00
Epoch 2/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2623 - precision: 0.4906 - val_loss: 0.2579 - val_precision: 0.5191
Epoch 3/20
80/80 [==============================] - 3s 32ms/step - loss: 0.2551 - precision: 0.5159 - val_loss: 0.2569 - val_precision: 0.5474
Epoch 4/20
80/80 [==============================] - 3s 32ms/step - loss: 0.2528 - precision: 0.5370 - val_loss: 0.2622 - val_precision: 0.5276
Epoch 5/20
80/80 [==============================] - 3s 32ms/step - loss: 0.2504 - precision: 0.5390 - val_loss: 0.2497 - val_precision: 0.5730
Epoch 6/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2502 - precision: 0.5408 - val_loss: 0.2555 - val_precision: 0.5603
Epoch 7/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2488 - precision: 0.5475 - val_loss: 0.2508 - val_precision: 0.5437
Epoch 8/20
80/80 [==============================] - 3s 32ms/step - loss: 0.2476 - precision: 0.5580 - val_loss: 0.2512 - val_precision: 0.5236
Epoch 9/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2476 - precision: 0.5556 - val_loss: 0.2484 - val_precision: 0.5293
Epoch 10/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2468 - precision: 0.5661 - val_loss: 0.2521 - val_precision: 0.5620
Epoch 11/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2455 - precision: 0.5711 - val_loss: 0.2608 - val_precision: 0.5140
```



```
Epoch 12/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2435 - precision: 0.5894 - val_loss: 0.2539 - val_precision: 0.5095
Epoch 13/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2427 - precision: 0.5921 - val_loss: 0.2467 - val_precision: 0.5646
Epoch 14/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2416 - precision: 0.5990 - val_loss: 0.2457 - val_precision: 0.5998
Epoch 15/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2398 - precision: 0.6100 - val_loss: 0.2440 - val_precision: 0.5955
Epoch 16/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2384 - precision: 0.6213 - val_loss: 0.2515 - val_precision: 0.6437
Epoch 17/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2369 - precision: 0.6246 - val_loss: 0.2425 - val_precision: 0.6177
Epoch 18/20
80/80 [==============================] - 3s 32ms/step - loss: 0.2354 - precision: 0.6302 - val_loss: 0.2453 - val_precision: 0.6521
Epoch 19/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2340 - precision: 0.6280 - val_loss: 0.2462 - val_precision: 0.6466
Epoch 20/20
80/80 [==============================] - 3s 33ms/step - loss: 0.2308 - precision: 0.6501 - val_loss: 0.2431 - val_precision: 0.6143
```

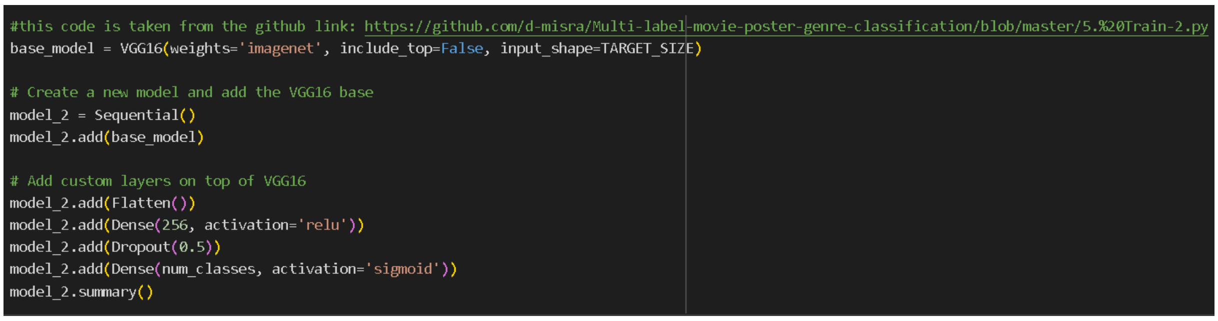
VGG16

After conducting some research I decided to use VGG16 as a pre-trained model. VGG16 is a convolutional neural network known as ConvNet. This neural network has input and output layers followed by various hidden layers. It is considered to be one of the best models since It is able to classify 1000 different images of 1000 different categories, having an accuracy of 92.7% (Rohini G, 2021). It is also one of the popular algorithms since it is very easy to use with transfer learning. This model is a good choice for Mulit-Label classification because of its ability to capture all the hierarchical and abstract features from images.

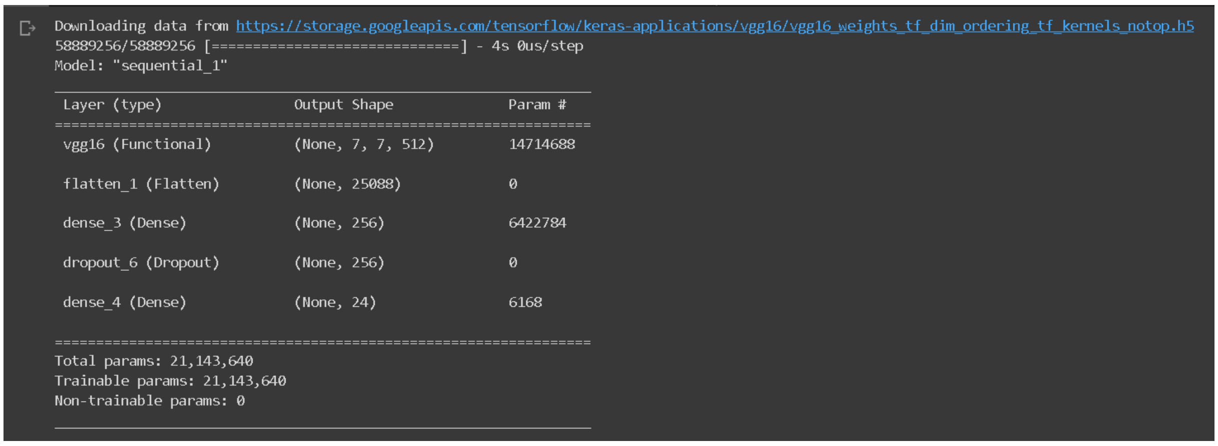
VGG16 has already been pre-trained on a large scale ImageNet dataset. This dataset contains millions of Images which are labelled in various categories. The pre-training of VGG16 will enable our model to learn features like: edges, shapes etc.. efficiently, thus helping us in performing accurate multi-label classification of movie genres. We can use transfer learning with VGG16 by leveraging the learned features in the ImageNet dataset and then transfer them for movie genre classification. This saves time and computational resources. VGG16 can be used for feature extraction and fine-tuning where we can freeze and unfreeze some layers and only train some particular layers.

In our code, the VGG16 model is being used as a layer inside a sequential model after which we are applying the dense layers to improve the neural net capabilities. Since VGG16 is trained on a relatively big imagenet dataset, there will be a good chance of overfitting so I decided to add a dropout layer within our model. The dense layers are then used for increasing the capacity of the model and enable it to learn complex patterns inside the input data.

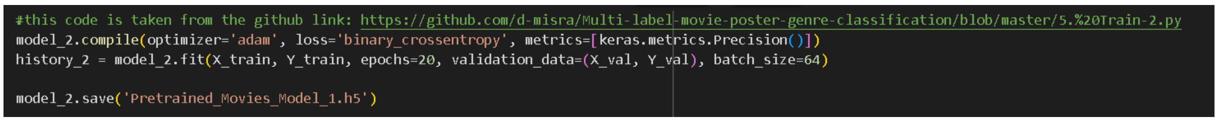
The screenshots of the full-layer design architecture of the pre-trained model are shown below:



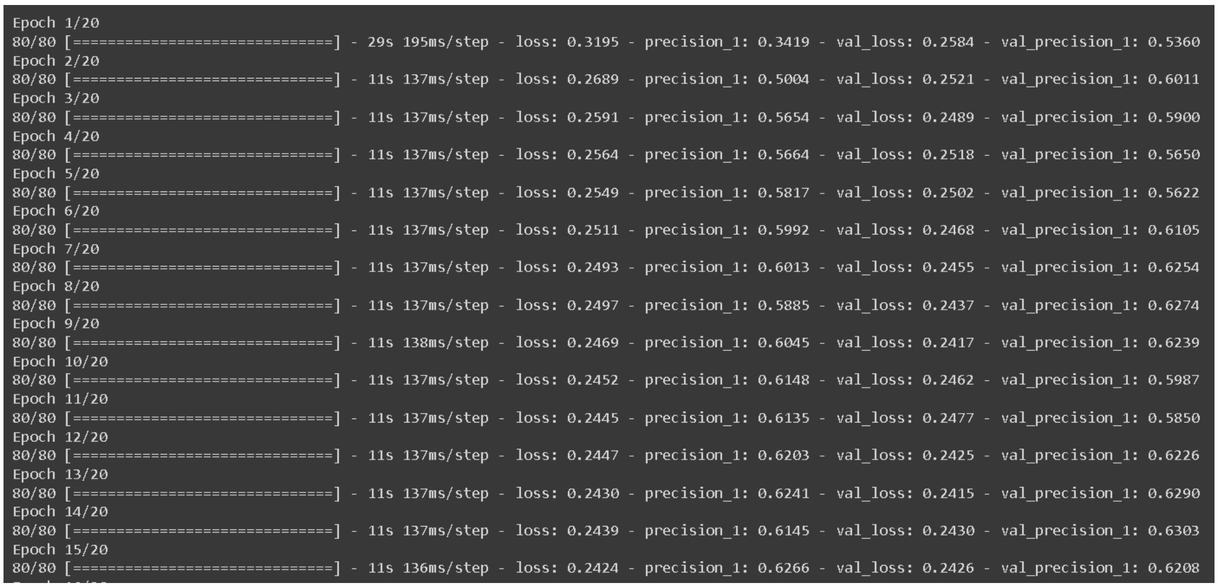
```
#this code is taken from the github link: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/5.%20Train-2.py
base_model = VGG16(weights='imagenet', include_top=False, input_shape=TARGET_SIZE)

# Create a new model and add the VGG16 base
model_2 = Sequential()
model_2.add(base_model)

# Add custom layers on top of VGG16
model_2.add(Flatten())
model_2.add(Dense(256, activation='relu'))
model_2.add(Dropout(0.5))
model_2.add(Dense(num_classes, activation='sigmoid'))
model_2.summary()
```



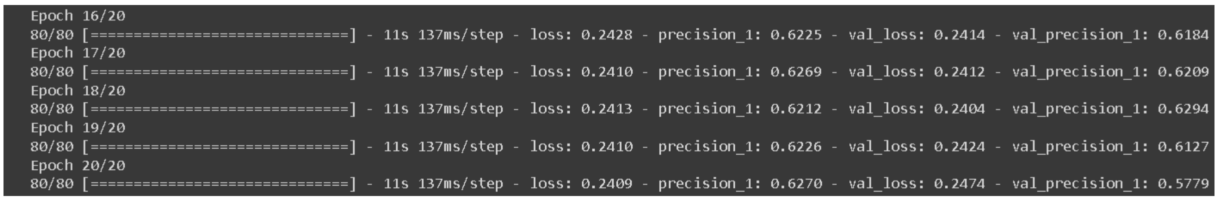
```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 4s 0us/step
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten_1 (Flatten)         (None, 25088)             0

 dense_3 (Dense)             (None, 256)               6422784

 dropout_6 (Dropout)         (None, 256)               0

 dense_4 (Dense)             (None, 24)                6168

=================================================================
Total params: 21,143,640
Trainable params: 21,143,640
Non-trainable params: 0
_________________________________________________________________
```



```
#this code is taken from the github link: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/5.%20Train-2.py
model_2.compile(optimizer='adam', loss='binary_crossentropy', metrics=[keras.metrics.Precision()])
history_2 = model_2.fit(X_train, Y_train, epochs=20, validation_data=(X_val, Y_val), batch_size=64)

model_2.save('Pretrained_Movies_Model_1.h5')
```



```
Epoch 1/20
80/80 [==============================] - 29s 195ms/step - loss: 0.3195 - precision_1: 0.3419 - val_loss: 0.2584 - val_precision_1: 0.5360
Epoch 2/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2689 - precision_1: 0.5004 - val_loss: 0.2521 - val_precision_1: 0.6011
Epoch 3/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2591 - precision_1: 0.5654 - val_loss: 0.2489 - val_precision_1: 0.5900
Epoch 4/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2564 - precision_1: 0.5664 - val_loss: 0.2518 - val_precision_1: 0.5650
Epoch 5/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2549 - precision_1: 0.5817 - val_loss: 0.2502 - val_precision_1: 0.5622
Epoch 6/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2511 - precision_1: 0.5992 - val_loss: 0.2468 - val_precision_1: 0.6105
Epoch 7/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2493 - precision_1: 0.6013 - val_loss: 0.2455 - val_precision_1: 0.6254
Epoch 8/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2497 - precision_1: 0.5885 - val_loss: 0.2437 - val_precision_1: 0.6274
Epoch 9/20
80/80 [==============================] - 11s 138ms/step - loss: 0.2469 - precision_1: 0.6045 - val_loss: 0.2417 - val_precision_1: 0.6239
Epoch 10/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2452 - precision_1: 0.6148 - val_loss: 0.2462 - val_precision_1: 0.5987
Epoch 11/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2445 - precision_1: 0.6135 - val_loss: 0.2477 - val_precision_1: 0.5850
Epoch 12/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2447 - precision_1: 0.6203 - val_loss: 0.2425 - val_precision_1: 0.6226
Epoch 13/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2430 - precision_1: 0.6241 - val_loss: 0.2415 - val_precision_1: 0.6290
Epoch 14/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2439 - precision_1: 0.6145 - val_loss: 0.2430 - val_precision_1: 0.6303
Epoch 15/20
80/80 [==============================] - 11s 136ms/step - loss: 0.2424 - precision_1: 0.6266 - val_loss: 0.2426 - val_precision_1: 0.6208
```



```
Epoch 16/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2428 - precision_1: 0.6225 - val_loss: 0.2414 - val_precision_1: 0.6184
Epoch 17/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2410 - precision_1: 0.6269 - val_loss: 0.2412 - val_precision_1: 0.6209
Epoch 18/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2413 - precision_1: 0.6212 - val_loss: 0.2404 - val_precision_1: 0.6294
Epoch 19/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2410 - precision_1: 0.6226 - val_loss: 0.2424 - val_precision_1: 0.6127
Epoch 20/20
80/80 [==============================] - 11s 137ms/step - loss: 0.2409 - precision_1: 0.6270 - val_loss: 0.2474 - val_precision_1: 0.5779
```

Section 4: Hyperparameter tuning

4.1 Hyperparameter Discussion

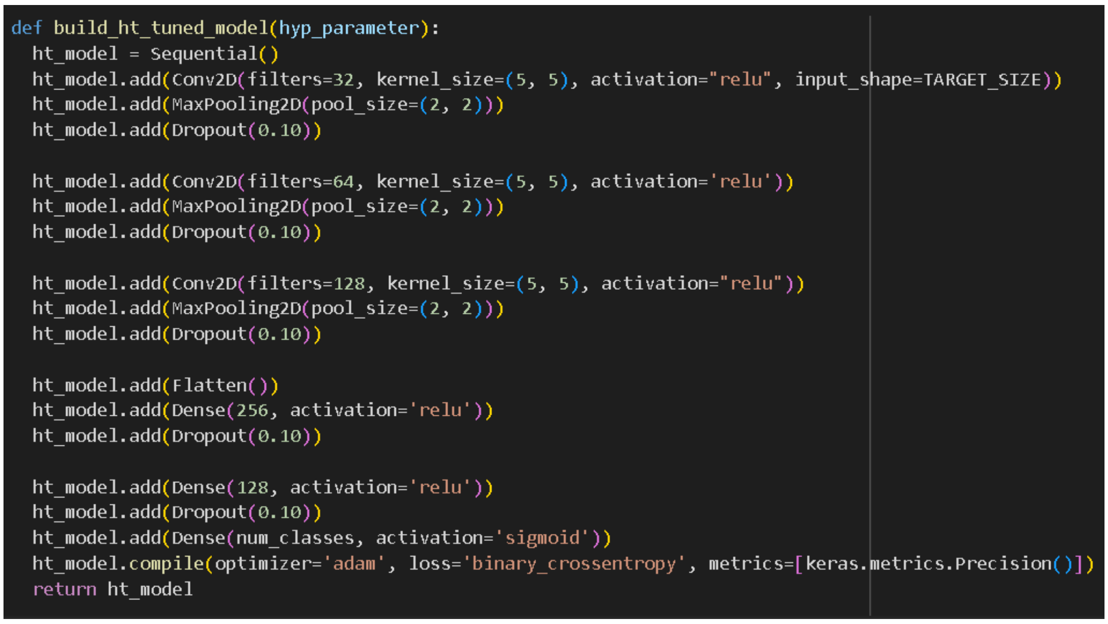
In order to find out the best hyperparameter tuner for my model I explored all the tuners from the keras library. There are various tuners available such as random search, hyperband and Bayesian optimization tuners in Keras (Wadekar.S., 2021). Random search tries to find the optimal parameters and tries every possible combination out of all the available parameters, so trying these combinations one after the other will take a lot of time to explore. Hyperband technique is better than random search, since random search sometimes picks irrelevant values and does a complete training which could be of waste (Wadekar.S., 2021). Hyperband samples all the combinations at random, and instead of full training and evaluation it trains the model for a few epochs and then selects the best values that were obtained from these epochs.

Although the random search and hyperband tuners give out optimal results, both of them have a problem, which is, hyperparameters are chosen randomly, which does not give out promising results. Bayesian Optimization could be there to solve this problem (Wadekar.S., 2021). Instead of taking all the possible combinations at random, it chooses the values randomly and determines the performance of these hyperparameters (Wadekar.S., 2021). The tuner then chooses the next best possible hyperparameter based on the performance on the previous values, so it considers all the hyperparameters we tried before previously. Therefore, considering all this information, I decided to use the Bayesian optimization tuner defined in the Keras library to perform hyperparameter tuning.

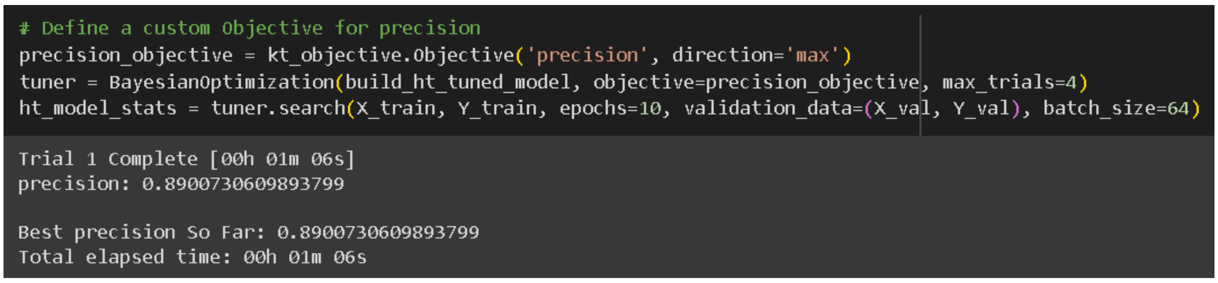
4.2 Custom model fitting using the Bayesian Optimization tuner.

For tuning our custom model, I decided to use precision metric, because it takes into account the number of true and false positives for each genre of a movie poster individually which enables us to evaluate the performance of our model more accurately for each genre of a movie poster. I decided keep the 'adam' optimizer, because I tried other optimizers like RMSProp, Adadelta etc, but I wasn't getting the precision value 60% atleast. In order to search for best optimal hyperparameters I used 10 epoches for determining the best precision with Bayesian Optimization tuner. After that I get the best optimial hyperparameters using 'get_best_hyperparameters()' function from the tuner variable, then build and fit the hyperparameter-tuned model. I trained the model with 9 epochs because I was able to achieve the precision of atleast 60% and having greater number of epochs lead to overfitting of my model, which means it was performing well on training data but not so great on the validation data.

The full-layer architecture(code and output) for hyperparameter tuning with a custom sequential model is mentioned below.

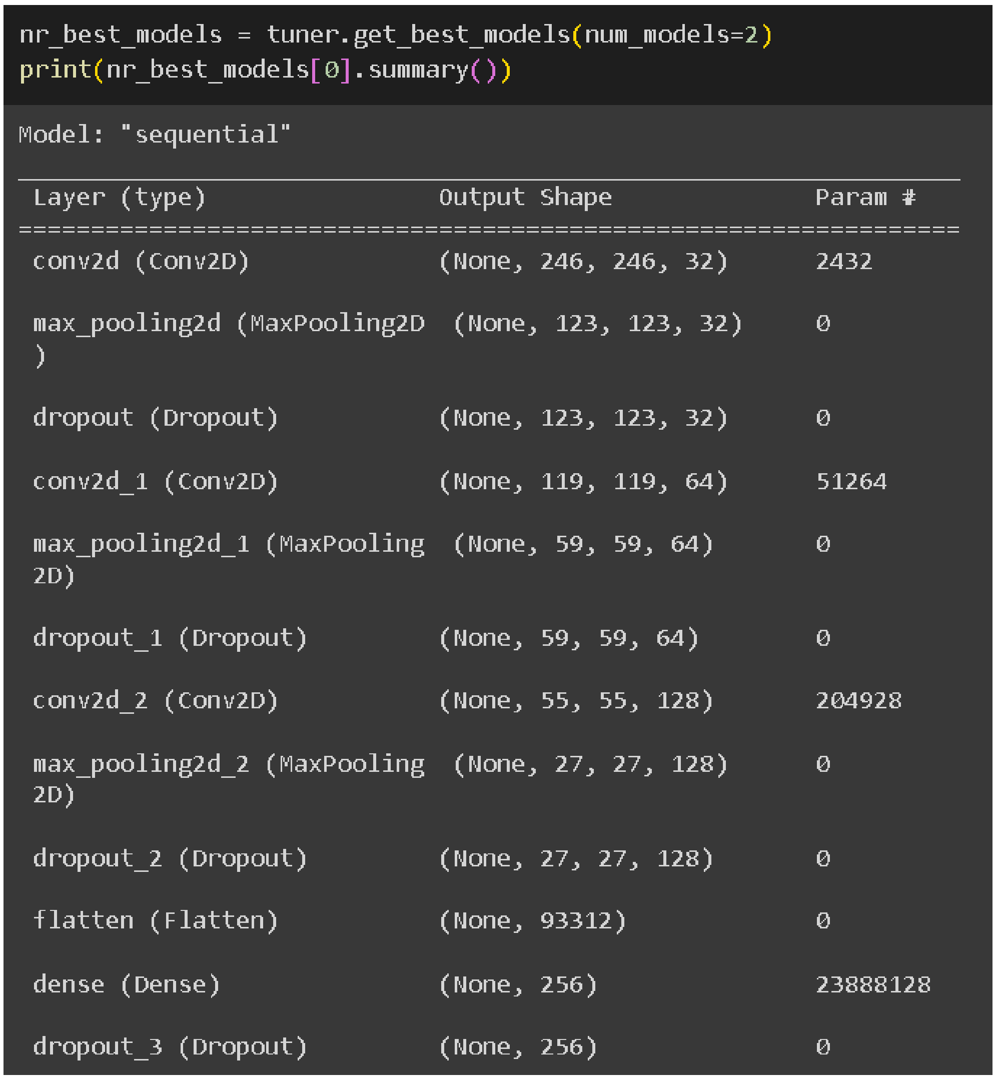


```
def build_ht_tuned_model(hyp_parameter):
  ht_model = Sequential()
  ht_model.add(Conv2D(filters=32, kernel_size=(5, 5), activation="relu", input_shape=TARGET_SIZE))
  ht_model.add(MaxPooling2D(pool_size=(2, 2)))
  ht_model.add(Dropout(0.10))

  ht_model.add(Conv2D(filters=64, kernel_size=(5, 5), activation='relu'))
  ht_model.add(MaxPooling2D(pool_size=(2, 2)))
  ht_model.add(Dropout(0.10))

  ht_model.add(Conv2D(filters=128, kernel_size=(5, 5), activation="relu"))
  ht_model.add(MaxPooling2D(pool_size=(2, 2)))
  ht_model.add(Dropout(0.10))

  ht_model.add(Flatten())
  ht_model.add(Dense(256, activation='relu'))
  ht_model.add(Dropout(0.10))

  ht_model.add(Dense(128, activation='relu'))
  ht_model.add(Dropout(0.10))
  ht_model.add(Dense(num_classes, activation='sigmoid'))
  ht_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[keras.metrics.Precision()])
  return ht_model
```

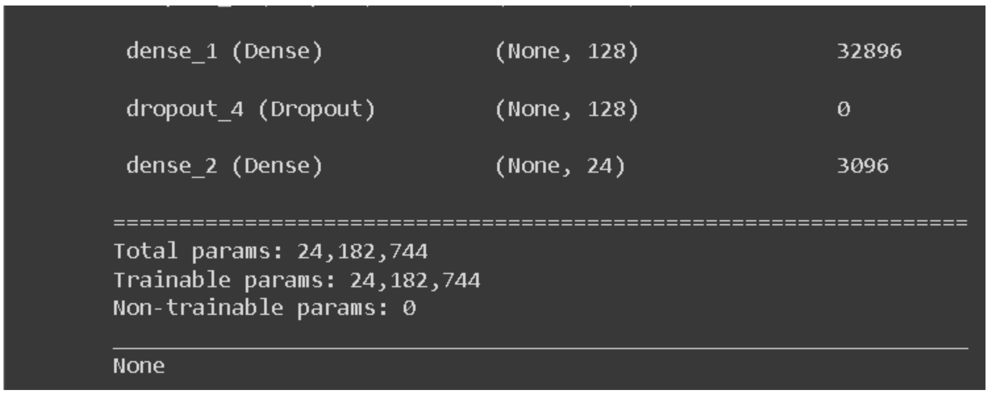


```
# Define a custom Objective for precision
precision_objective = kt_objective.Objective('precision', direction='max')
tuner = BayesianOptimization(build_ht_tuned_model, objective=precision_objective, max_trials=4)
ht_model_stats = tuner.search(X_train, Y_train, epochs=10, validation_data=(X_val, Y_val), batch_size=64)
```

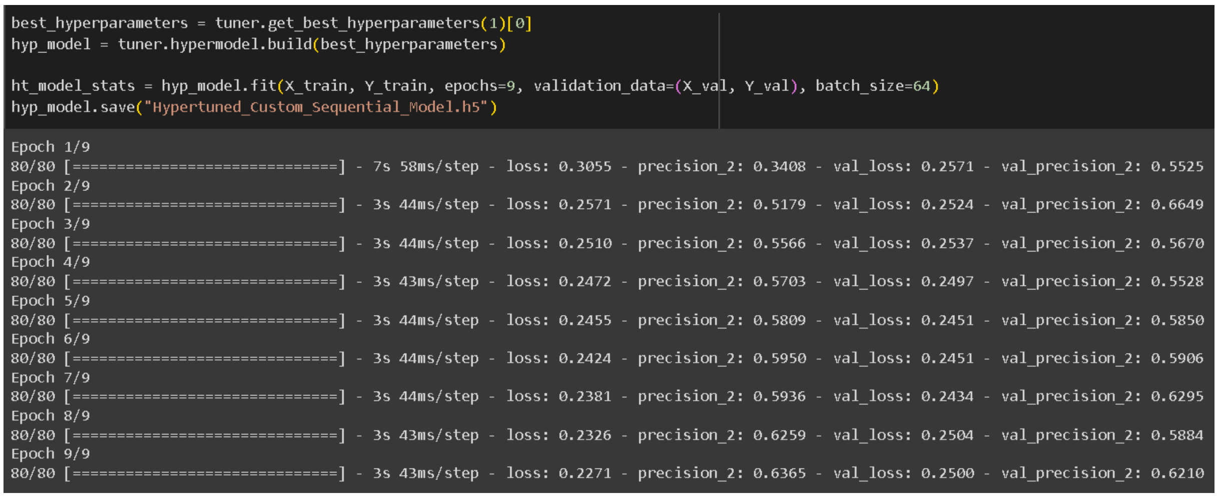
```
Trial 1 Complete [00h 01m 06s]
precision: 0.8900730609893799

Best precision So Far: 0.8900730609893799
Total elapsed time: 00h 01m 06s
```



```
nr_best_models = tuner.get_best_models(num_models=2)
print(nr_best_models[0].summary())
```



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 246, 246, 32)      2432

 max_pooling2d (MaxPooling2D  (None, 123, 123, 32)     0
 )

 dropout (Dropout)           (None, 123, 123, 32)      0

 conv2d_1 (Conv2D)           (None, 119, 119, 64)      51264

 max_pooling2d_1 (MaxPooling  (None, 59, 59, 64)       0
 2D)

 dropout_1 (Dropout)         (None, 59, 59, 64)        0

 conv2d_2 (Conv2D)           (None, 55, 55, 128)       204928

 max_pooling2d_2 (MaxPooling  (None, 27, 27, 128)      0
 2D)

 dropout_2 (Dropout)         (None, 27, 27, 128)       0

 flatten (Flatten)           (None, 93312)             0

 dense (Dense)               (None, 256)               23888128

 dropout_3 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 128)               32896

 dropout_4 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 24)                3096

=================================================================
Total params: 24,182,744
Trainable params: 24,182,744
Non-trainable params: 0
_________________________________________________________________
None
```



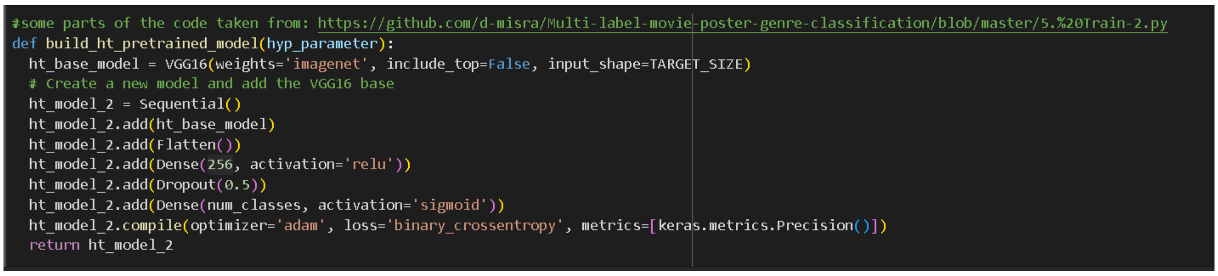
```
best_hyperparameters = tuner.get_best_hyperparameters(1)[0]
hyp_model = tuner.hypermodel.build(best_hyperparameters)

ht_model_stats = hyp_model.fit(X_train, Y_train, epochs=9, validation_data=(X_val, Y_val), batch_size=64)
hyp_model.save("Hypertuned_Custom_Sequential_Model.h5")
```

```
Epoch 1/9
80/80 [==============================] - 7s 58ms/step - loss: 0.3055 - precision_2: 0.3408 - val_loss: 0.2571 - val_precision_2: 0.5525
Epoch 2/9
80/80 [==============================] - 3s 44ms/step - loss: 0.2571 - precision_2: 0.5179 - val_loss: 0.2524 - val_precision_2: 0.6649
Epoch 3/9
80/80 [==============================] - 3s 44ms/step - loss: 0.2510 - precision_2: 0.5566 - val_loss: 0.2537 - val_precision_2: 0.5670
Epoch 4/9
80/80 [==============================] - 3s 43ms/step - loss: 0.2472 - precision_2: 0.5703 - val_loss: 0.2497 - val_precision_2: 0.5528
Epoch 5/9
80/80 [==============================] - 3s 44ms/step - loss: 0.2455 - precision_2: 0.5809 - val_loss: 0.2451 - val_precision_2: 0.5850
Epoch 6/9
80/80 [==============================] - 3s 44ms/step - loss: 0.2424 - precision_2: 0.5950 - val_loss: 0.2451 - val_precision_2: 0.5906
Epoch 7/9
80/80 [==============================] - 3s 44ms/step - loss: 0.2381 - precision_2: 0.5936 - val_loss: 0.2434 - val_precision_2: 0.6295
Epoch 8/9
80/80 [==============================] - 3s 43ms/step - loss: 0.2326 - precision_2: 0.6259 - val_loss: 0.2504 - val_precision_2: 0.5884
Epoch 9/9
80/80 [==============================] - 3s 43ms/step - loss: 0.2271 - precision_2: 0.6365 - val_loss: 0.2500 - val_precision_2: 0.6210
```

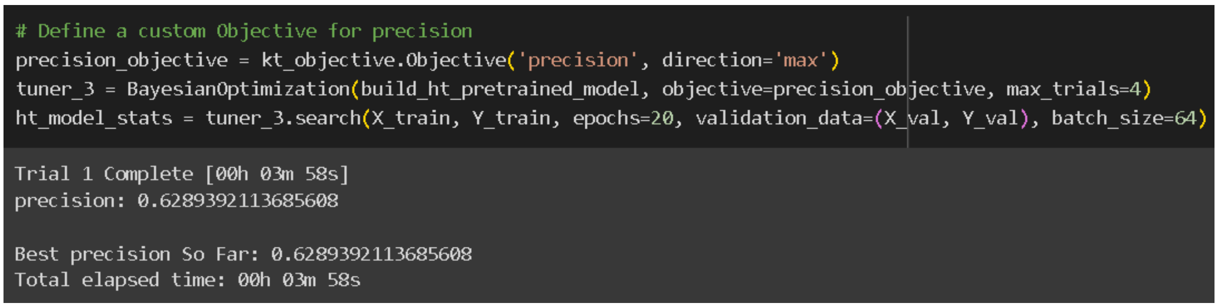
4.3 Pre-trained model fitting using Bayesian Optimization tuner

For the pre-trained model, I decided to use the precision metric over accuracy since the accuracy based of the same previous explanation. I also decided to add the dropout layer since VGG16 is trained on a large dataset, and training the model without adding the dropout layer could lead to overfitting of our model. For training the pre-trained model using the optimizer. I decided to increase the number of epochs from 10 to 20 epochs in order to achieve a precision of at least 60%. The reason for using the 'adam' optimizer and loss function as 'binary cross entropy' were the same as mentioned before. For fitting the hyper-tuned pre-trained model I first extracted the best hyperparameters from the tuner, then build and fit the model and trained on 20 epochs to achieve the metrics value of 60%. I also tried to train the model with epochs less then 20 and more then twenty, but then this process was taking a lot of time and the accuracy metrics was less than 60% so I trained the model with 20 epoches.

The full-layer architecture(code and output) for hyper parameter tuning with VGG16 is shown below.

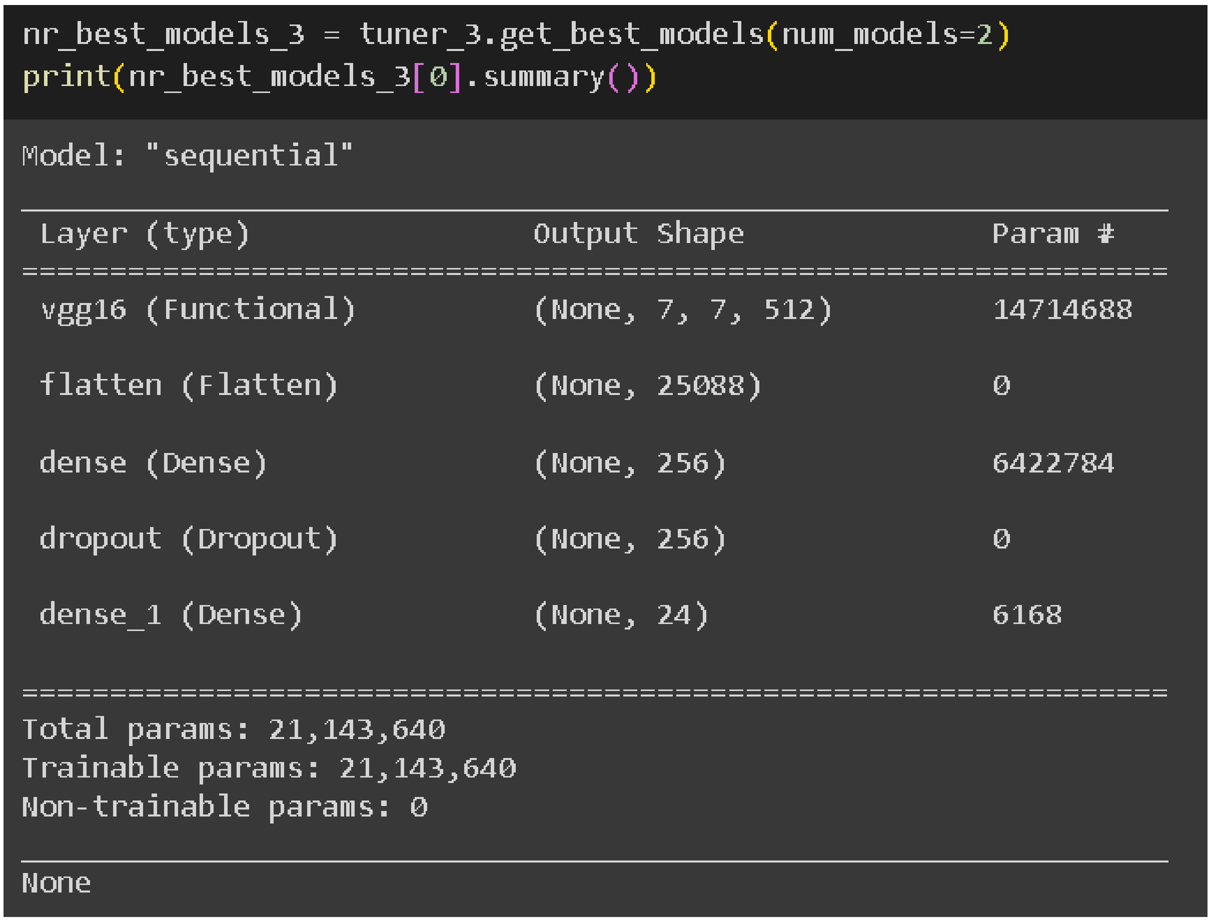


```
#some parts of the code taken from: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/5.%20Train-2.py
def build_ht_pretrained_model(hyp_parameter):
  ht_base_model = VGG16(weights='imagenet', include_top=False, input_shape=TARGET_SIZE)
  # Create a new model and add the VGG16 base
  ht_model_2 = Sequential()
  ht_model_2.add(ht_base_model)
  ht_model_2.add(Flatten())
  ht_model_2.add(Dense(256, activation='relu'))
  ht_model_2.add(Dropout(0.5))
  ht_model_2.add(Dense(num_classes, activation='sigmoid'))
  ht_model_2.compile(optimizer='adam', loss='binary_crossentropy', metrics=[keras.metrics.Precision()])
  return ht_model_2
```



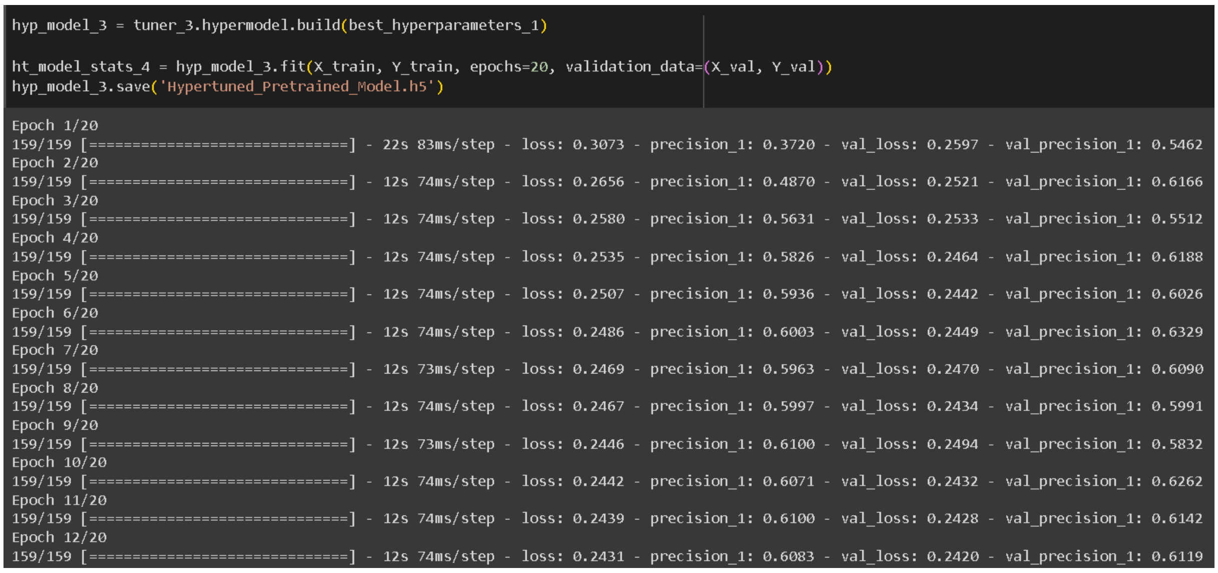
```
# Define a custom Objective for precision
precision_objective = kt_objective.Objective('precision', direction='max')
tuner_3 = BayesianOptimization(build_ht_pretrained_model, objective=precision_objective, max_trials=4)
ht_model_stats = tuner_3.search(X_train, Y_train, epochs=20, validation_data=(X_val, Y_val), batch_size=64)
```

```
Trial 1 Complete [00h 03m 58s]
precision: 0.6289392113685608

Best precision So Far: 0.6289392113685608
Total elapsed time: 00h 03m 58s
```



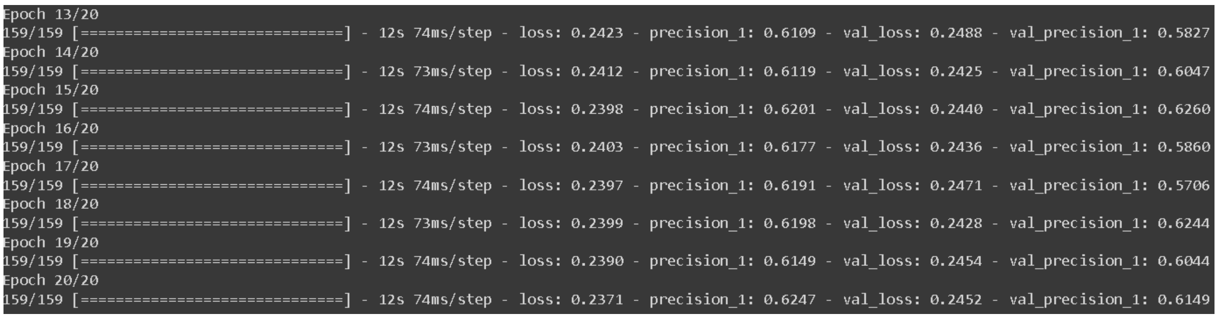
```
nr_best_models_3 = tuner_3.get_best_models(num_models=2)
print(nr_best_models_3[0].summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 256)               6422784

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 24)                6168

=================================================================
Total params: 21,143,640
Trainable params: 21,143,640
Non-trainable params: 0
_________________________________________________________________
None
```



```
hyp_model_3 = tuner_3.hypermodel.build(best_hyperparameters_1)

ht_model_stats_4 = hyp_model_3.fit(X_train, Y_train, epochs=20, validation_data=(X_val, Y_val))
hyp_model_3.save('Hypertuned_Pretrained_Model.h5')
```

```
Epoch 1/20
159/159 [==============================] - 22s 83ms/step - loss: 0.3073 - precision_1: 0.3720 - val_loss: 0.2597 - val_precision_1: 0.5462
Epoch 2/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2656 - precision_1: 0.4870 - val_loss: 0.2521 - val_precision_1: 0.6166
Epoch 3/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2580 - precision_1: 0.5631 - val_loss: 0.2533 - val_precision_1: 0.5512
Epoch 4/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2535 - precision_1: 0.5826 - val_loss: 0.2464 - val_precision_1: 0.6188
Epoch 5/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2507 - precision_1: 0.5936 - val_loss: 0.2442 - val_precision_1: 0.6026
Epoch 6/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2486 - precision_1: 0.6003 - val_loss: 0.2449 - val_precision_1: 0.6329
Epoch 7/20
159/159 [==============================] - 12s 73ms/step - loss: 0.2469 - precision_1: 0.5963 - val_loss: 0.2470 - val_precision_1: 0.6090
Epoch 8/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2467 - precision_1: 0.5997 - val_loss: 0.2434 - val_precision_1: 0.5991
Epoch 9/20
159/159 [==============================] - 12s 73ms/step - loss: 0.2446 - precision_1: 0.6100 - val_loss: 0.2494 - val_precision_1: 0.5832
Epoch 10/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2442 - precision_1: 0.6071 - val_loss: 0.2432 - val_precision_1: 0.6262
Epoch 11/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2439 - precision_1: 0.6100 - val_loss: 0.2428 - val_precision_1: 0.6142
Epoch 12/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2431 - precision_1: 0.6083 - val_loss: 0.2420 - val_precision_1: 0.6119
```



```
Epoch 13/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2423 - precision_1: 0.6109 - val_loss: 0.2488 - val_precision_1: 0.5827
Epoch 14/20
159/159 [==============================] - 12s 73ms/step - loss: 0.2412 - precision_1: 0.6119 - val_loss: 0.2425 - val_precision_1: 0.6047
Epoch 15/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2398 - precision_1: 0.6201 - val_loss: 0.2440 - val_precision_1: 0.6260
Epoch 16/20
159/159 [==============================] - 12s 73ms/step - loss: 0.2403 - precision_1: 0.6177 - val_loss: 0.2436 - val_precision_1: 0.5860
Epoch 17/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2397 - precision_1: 0.6191 - val_loss: 0.2471 - val_precision_1: 0.5706
Epoch 18/20
159/159 [==============================] - 12s 73ms/step - loss: 0.2399 - precision_1: 0.6198 - val_loss: 0.2428 - val_precision_1: 0.6244
Epoch 19/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2390 - precision_1: 0.6149 - val_loss: 0.2454 - val_precision_1: 0.6044
Epoch 20/20
159/159 [==============================] - 12s 74ms/step - loss: 0.2371 - precision_1: 0.6247 - val_loss: 0.2452 - val_precision_1: 0.6149
```

Section 5: Diagnostics Curves

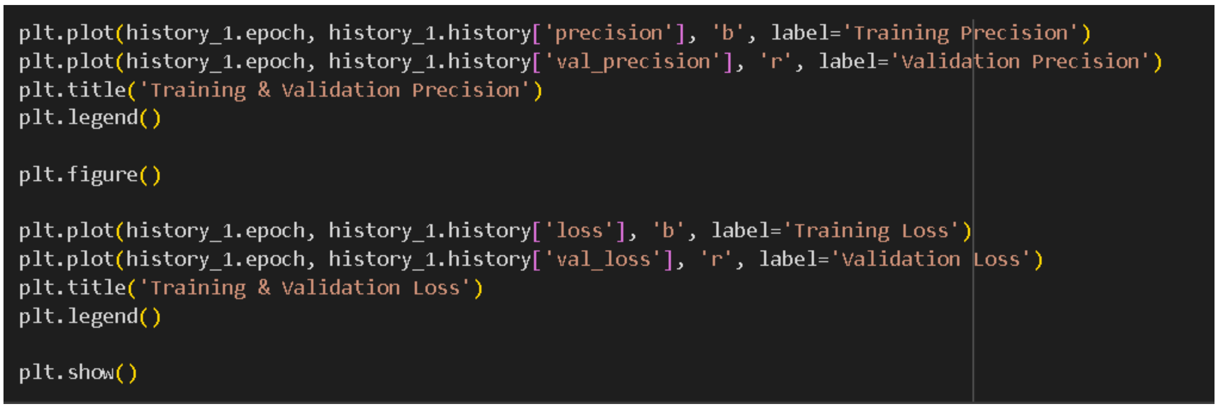
In this section, we will discuss about the training and validation precision and loss curves of our sequential and pre-trained model. The explanation of the curves is mentioned in the sub-sections mentioned below:

5.1 Training and validation curves of the custom-based model

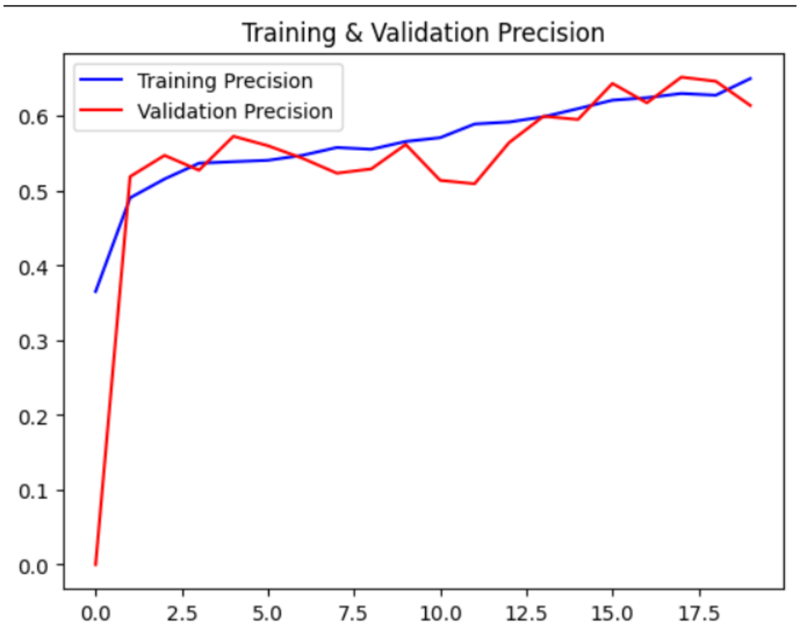
Below you will find the training and validation curves, for model precision and loss for our custom-created sequential model.

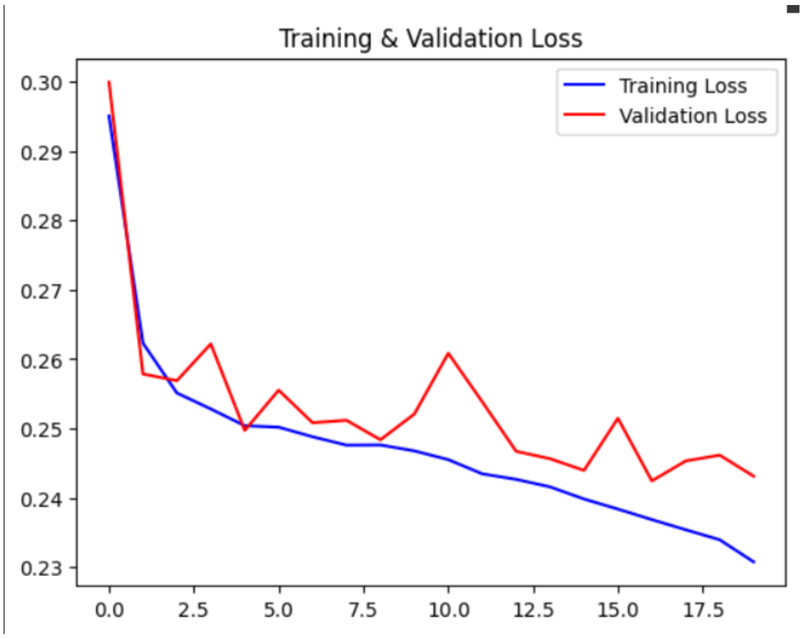
The code and its respective outputs are mentioned below:

Overall, this code mentioned below was used to obtain the training and validation precision and loss curves for our custom, pre-trained and hyper-tuned models. I had to change the variable names with appropriate variables, like for example: ‘history_1’ and the values inside the square brackets to get the correct plots for each model but overall, this code was used in making the plots of each model.



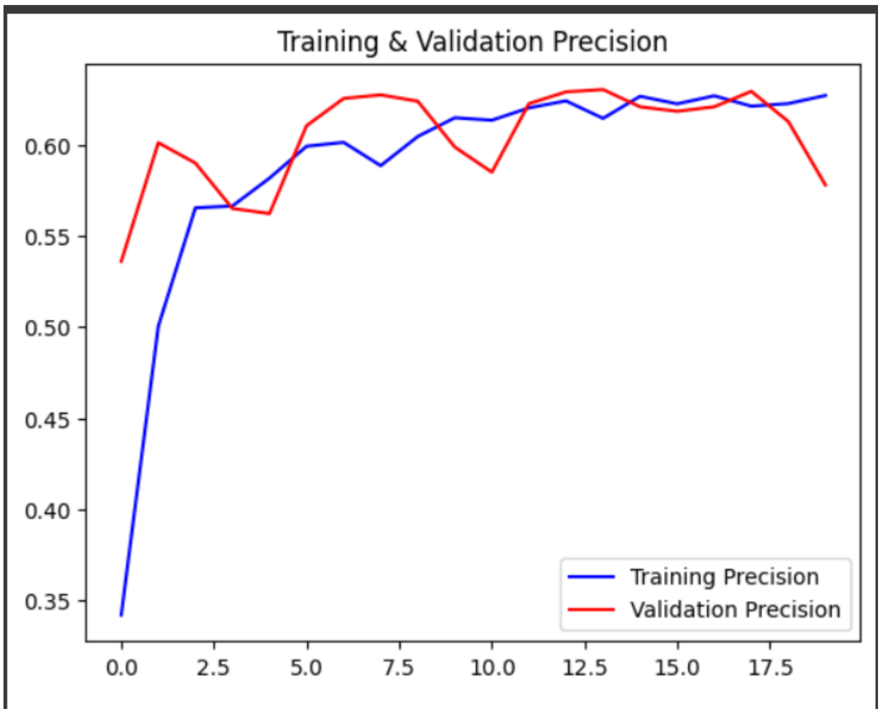
```
plt.plot(history_1.epoch, history_1.history['precision'], 'b', label='Training Precision')
plt.plot(history_1.epoch, history_1.history['val_precision'], 'r', label='Validation Precision')
plt.title('Training & Validation Precision')
plt.legend()

plt.figure()

plt.plot(history_1.epoch, history_1.history['loss'], 'b', label='Training Loss')
plt.plot(history_1.epoch, history_1.history['val_loss'], 'r', label='Validation Loss')
plt.title('Training & Validation Loss')
plt.legend()

plt.show()
```

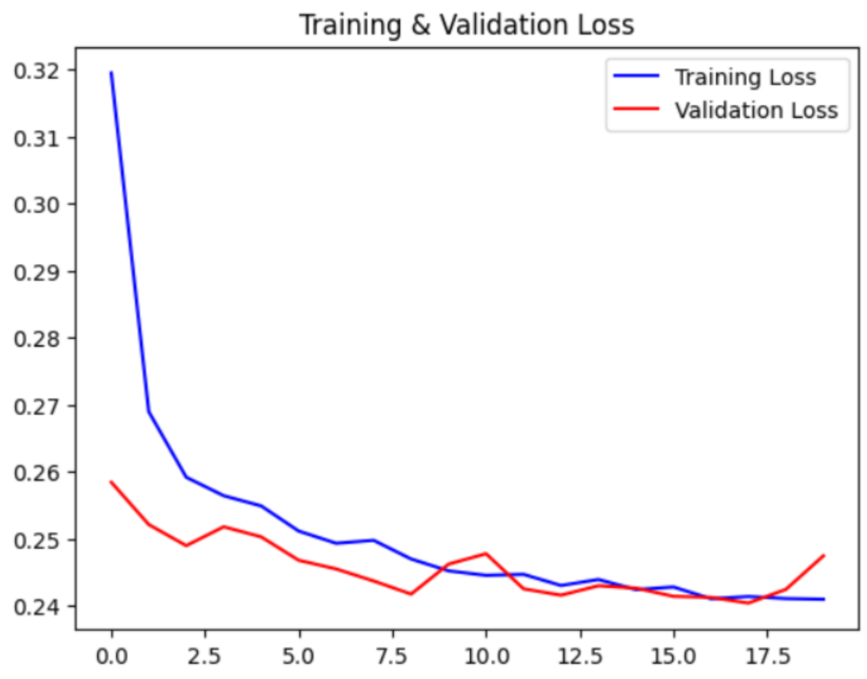




Overall, after comparing the training and validation precision of the sequential model, The training accuracy tends to increase as we train our model. The validation accuracy starts initially at a lower value, but then slowly increases as the model training proceeds. We can also notice that the training and validation precision tends to increase as we train the model further. After analysing the training and validation loss curves, we can see that we lose less training and validation data as we train the model further. However, the data loss within our validation data is decreasing which is a good sign because in that way our model is able to achieve greater accuracy in predicting the genres of movie poster on new or unseen data.

5.2 Training and validation curves of the pre-trained model

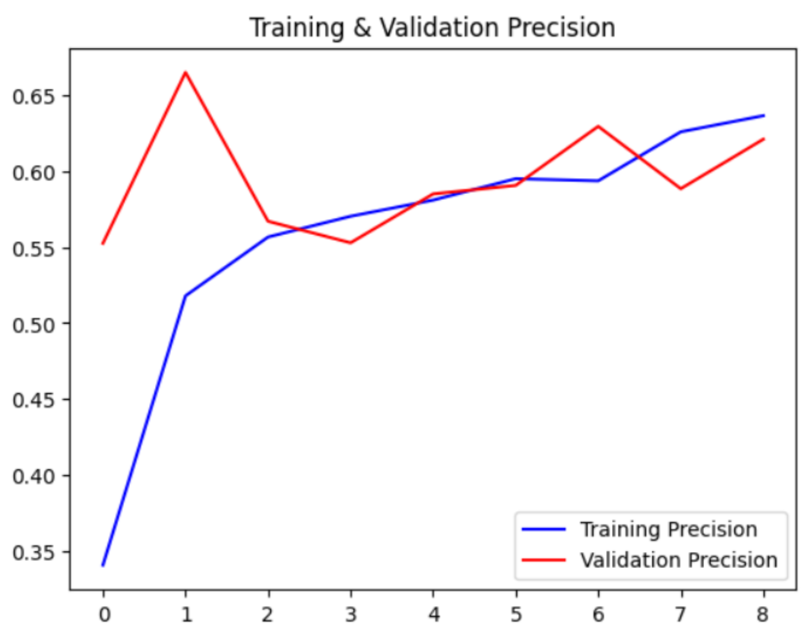


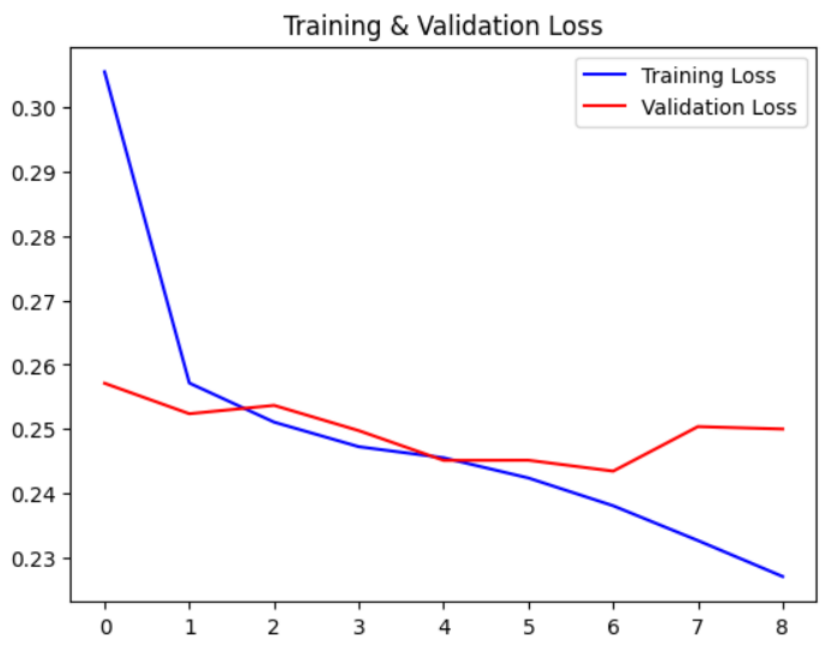


Overall, after comparing the training and validation accuracy curves of our pre-trained model, we observe that the training precision of our model tends to increase we train our pre-trained model further, whereas in the validation precision we notice some fluctuations initially, but then tends to increase as the training proceeds further. The loss in data of training and validation loss curves also shows that as we train our pre-trained model further, we lose less amount of data, leading to higher precision.

5.3: Training and validation curves of hyper-tuned custom model

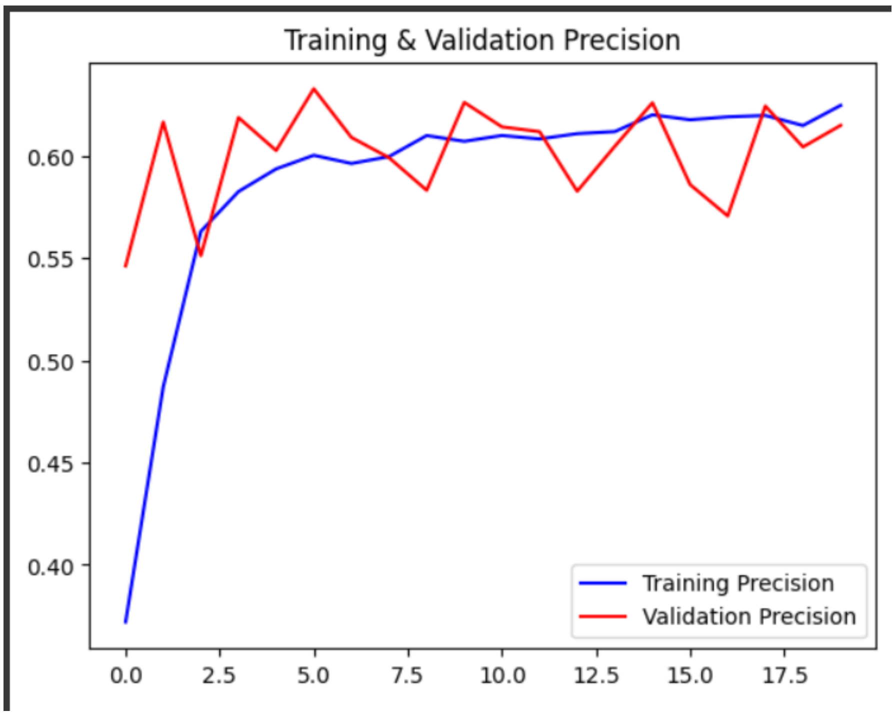
After applying the hyperparameter tuning to our custom sequential model, We get the following accuracy and loss curves for our training and validation data mentioned below. From the plots we can conclude that the training accuracy increases as we trained are hyper-tuned model. There are some fluctuations observed in the validation accuracy, but as we train our model further, the validation accuracy of our model also increases. We also notice some loss of data in the validation set but on the training set the data loss tends to decrease as we train our model further.

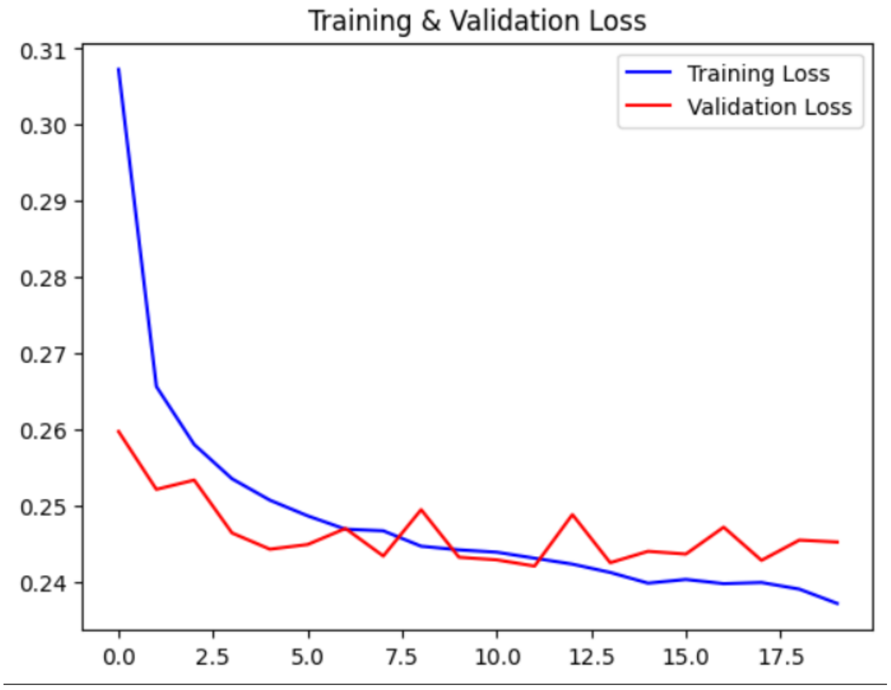




5.4: Training and validation curves of hyper-tuned pre-trained model.

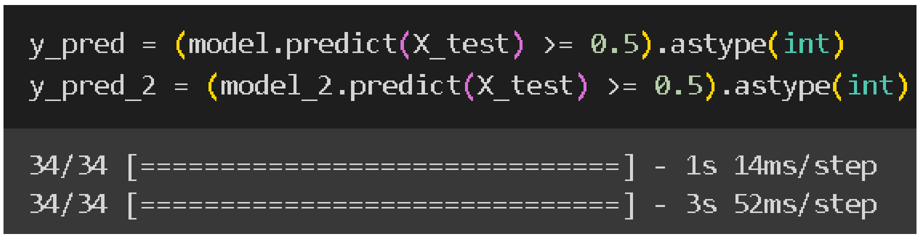
After applying the hyperparameter tuning to our pre-trained model, We get the following accuracy and loss curves for our training and validation data mentioned below. The accuracy values for our pre-trained model increases as we train our model further. There appears to a some fluctuations with validation accuracy, but it does tends to increase. There seem to be minimal amount of loss in data on both the training and validation side after tuning our pre-trained model.





Section 6: Models Performance

In this section, we are going to discuss the model performance of sequential and pre-trained and hyperparameter tuned model. Before calculating the metrics for both models we first have to get the predicted probabilities for both of our models using the testing set defined previously. We can do that using the .predict function. We then apply a threshold of 0.5, meaning any values greater than or equal to 0.5 gets a positive prediction, and anything lower than 0.5 gets a negative prediction. In the code snippet the ‘model’ stands for the sequential model and the ‘model_2’ stands for the pre-trained model.



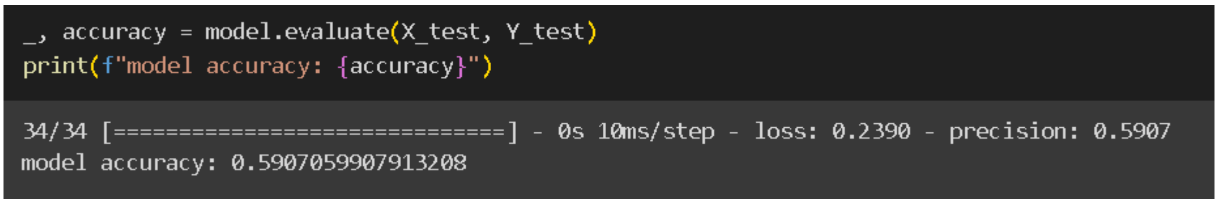
```
y_pred = (model.predict(X_test) >= 0.5).astype(int)
y_pred_2 = (model_2.predict(X_test) >= 0.5).astype(int)
```

```
34/34 [==============================] - 1s 14ms/step
34/34 [==============================] - 3s 52ms/step
```

6.1: Performance of the sequential model

Model Accuracy

The overall accuracy of our sequential model is about 59% which is not too bad, but not too great either. So there is some room for improvement, in efficiently designing the sequential model so that the accuracy of this model can increase.



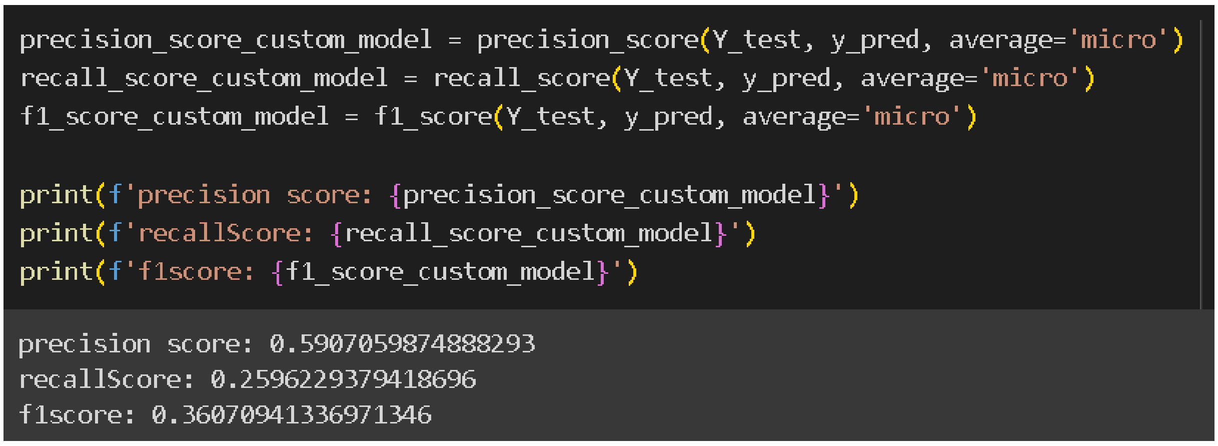
```
_, accuracy = model.evaluate(X_test, Y_test)
print(f"model accuracy: {accuracy}")
```

```
34/34 [==============================] - 0s 10ms/step - loss: 0.2390 - precision: 0.5907
model accuracy: 0.5907059907913208
```

Precision-recall and the F1-Score of sequential model

The precision, recall, and the F1 score of our sequential model are mentioned in the screenshot below. From the screenshot, we can determine that the precision score of our sequential model is: 0.5907059874888293, which can be rounded to 0.59, this means that our sequential model is predicting the correct movie genres about 59% of the time. And the rest 41% of time our model is predicting false positive outcomes.

The recall score of our sequential model is 0.2596229379418696, which can be rounded to 0.26. This means that the sequential model is only able to identify 26% of the correct genres of a particular movie poster. The rest of the time it is incorrectly labels the genres of a movie. The f1-score of our sequential model is: 0.36070941336971346, rounded to 0.36. f1-score is a combination of measurements: precision and recall. The value of f1score: 0.36 states that there is a fair balance between precision and recall of our model wherein our model is correctly classifying the genres of movie posters and captures all the positive instances within our dataset.

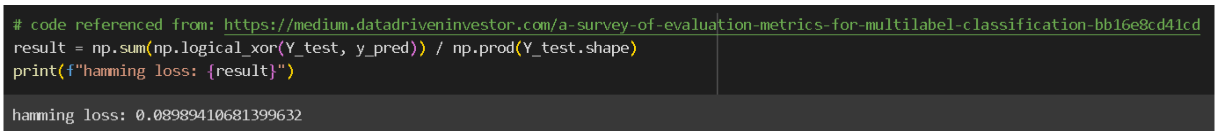


```
precision_score_custom_model = precision_score(Y_test, y_pred, average='micro')
recall_score_custom_model = recall_score(Y_test, y_pred, average='micro')
f1_score_custom_model = f1_score(Y_test, y_pred, average='micro')

print(f'precision score: {precision_score_custom_model}')
print(f'recallScore: {recall_score_custom_model}')
print(f'f1score: {f1_score_custom_model}')
```

```
precision score: 0.5907059874888293
recallScore: 0.2596229379418696
f1score: 0.36070941336971346
```

The Hamming Loss

The hamming loss of our model is: 0.08989410681399632, which means that our model is misclassifying around 8.99% of the genres based on the particular movie poster.

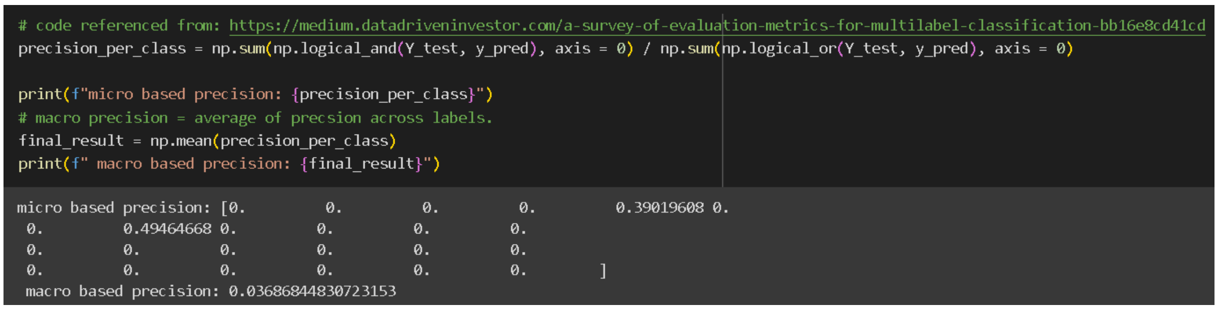


```
# code referenced from: https://medium.datadriveninvestor.com/a-survey-of-evaluation-metrics-for-multilabel-classification-bb16e8cd41cd
result = np.sum(np.logical_xor(Y_test, y_pred)) / np.prod(Y_test.shape)
print(f"hamming loss: {result}")
```

```
hamming loss: 0.08989410681399632
```

The Micro and Macro Precision based accuracy

In micro-based precision accuracy, I determined the precision of each genre. From the result, we can determine that most of the time our model was not able to classify any movie posters as positive for that particular genre. However there were some instances where the model was able to classify movies posters for that particular genre. The macro-based precision is an average precision of all the possible genres of the movie poster. The output value is: 0.03686844830723153, which is about 3.6%. which means that there is relatively low precision on detecting the genre of a particular movie.



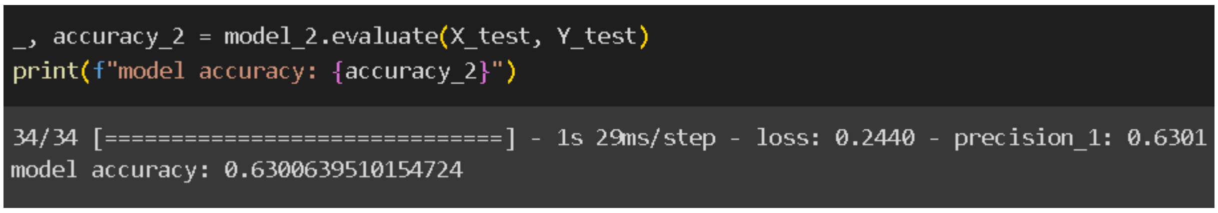
```
# code referenced from: https://medium.datadriveninvestor.com/a-survey-of-evaluation-metrics-for-multilabel-classification-bb16e8cd41cd
precision_per_class = np.sum(np.logical_and(Y_test, y_pred), axis = 0) / np.sum(np.logical_or(Y_test, y_pred), axis = 0)

print(f"micro based precision: {precision_per_class}")
# macro precision = average of precsion across labels.
final_result = np.mean(precision_per_class)
print(f" macro based precision: {final_result}")
```

```
micro based precision: [0.         0.         0.         0.         0.39019608 0.
 0.         0.49464668 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.        ]
 macro based precision: 0.03686844830723153
```

6.2: Performance of the pre-trained model

Model Accuracy

The model accuracy of the pre-trained model is about 63%, this means that our pre-trained model correctly predicted the true genres of a movie poster, in around 63% of all the instances, which is again not too good but not too bad either.



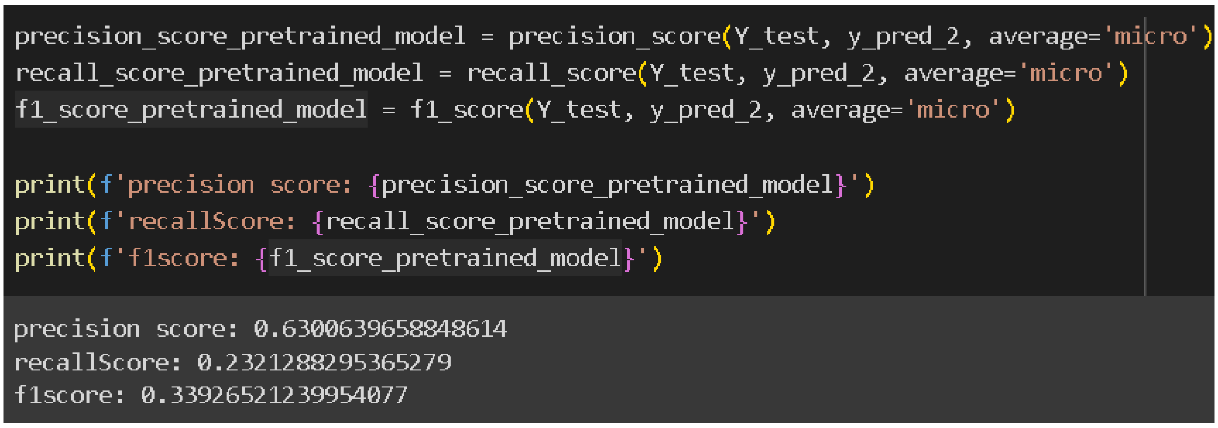
```
_, accuracy_2 = model_2.evaluate(X_test, Y_test)
print(f"model accuracy: {accuracy_2}")
```

```
34/34 [==============================] - 1s 29ms/step - loss: 0.2440 - precision_1: 0.6301
model accuracy: 0.6300639510154724
```

The precision recall and the F1-Score of a pre-trained model

The precision score of our pre-trained model is 0.6300639658848614, which can be rounded to 0.63. This states that our pre-trained model was able to predict the correct genres of a particular movie about 63% of the time. This precision is a bit better than our custom-created sequential model, but not very accurate either and needs to be tuned properly in order to get a more accurate precision.

The recall score of the pre-trained model is: 0.2321288295365279, this means that our pre-trained model was only able to correctly predict the genres of 23% of the movie posters in our dataset, which is relatively very low. The score of our pre-trained model is: 0.33926521239954077, which can be rounded to 0.34. The states that our pre-trained model performed moderately in terms of its precision and recall.

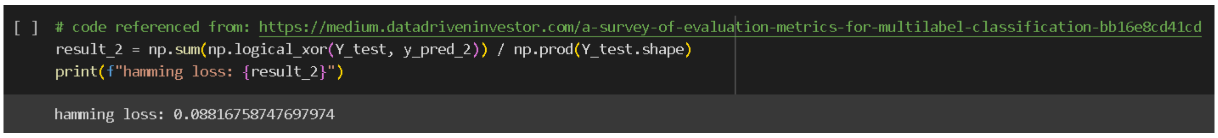


```
precision_score_pretrained_model = precision_score(Y_test, y_pred_2, average='micro')
recall_score_pretrained_model = recall_score(Y_test, y_pred_2, average='micro')
f1_score_pretrained_model = f1_score(Y_test, y_pred_2, average='micro')

print(f'precision score: {precision_score_pretrained_model}')
print(f'recallScore: {recall_score_pretrained_model}')
print(f'f1score: {f1_score_pretrained_model}')
```

```
precision score: 0.6300639658848614
recallScore: 0.2321288295365279
f1score: 0.33926521239954077
```

The Hamming Loss

The hamming loss of the pre-trained model is: 0.08832105586249232, this means that our pretrained model is in-correctly assigning 8.83% of the genres of a movie poster. In our pre-trained model, the hamming loss is relatively very low, which means most of the time our model is correctly assigning the genres of a movie from its particular poster.

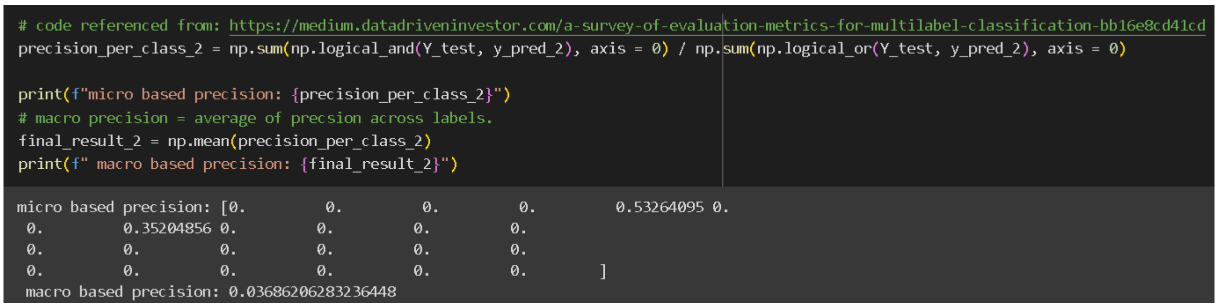


```
[ ] # code referenced from: https://medium.datadriveninvestor.com/a-survey-of-evaluation-metrics-for-multilabel-classification-bb16e8cd41cd
    result_2 = np.sum(np.logical_xor(Y_test, y_pred_2)) / np.prod(Y_test.shape)
    print(f"hamming loss: {result_2}")
```

```
    hamming loss: 0.08816758747697974
```

The micro and macro precision-based accuracy

In the micro-based precision, there were some instances where our pre-trained model was not able to classify any movie posters to that specific genre, hence the precision value is zero. There were however some cases where our pre-trained model was able to classify the movie posters that belong to that specific genre. The value of macro-based precision is: 0.0368620628323644, this is a precision that is obtained by taking the average values across each particular labels. From the value obtained, we can state that the performance of our pre-trained model is relatively very low across each particular label of movie genre.

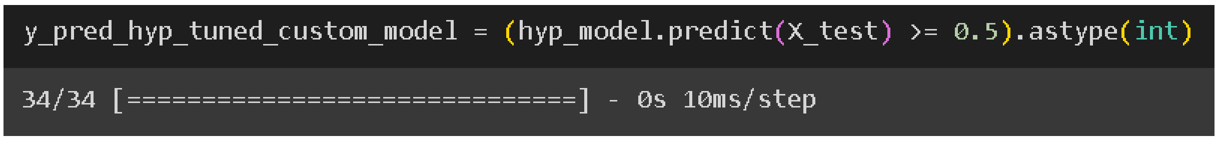


```
# code referenced from: https://medium.datadriveninvestor.com/a-survey-of-evaluation-metrics-for-multilabel-classification-bb16e8cd41cd
precision_per_class_2 = np.sum(np.logical_and(Y_test, y_pred_2), axis = 0) / np.sum(np.logical_or(Y_test, y_pred_2), axis = 0)

print(f"micro based precision: {precision_per_class_2}")
# macro precision = average of precsion across labels.
final_result_2 = np.mean(precision_per_class_2)
print(f" macro based precision: {final_result_2}")
```

```
micro based precision: [0.         0.         0.         0.         0.53264095 0.
 0.         0.35204856 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.        ]
 macro based precision: 0.03686206283236448
```

6.3: Performance of the hyper tuned custom sequential model

In this section, we are going to discuss the model performance of hyper tuned custom-sequential model. We first have to calculate the predicted probabilities of the hyper tuned model using the .predict function. We then repeat the same steps mentioned before to get the prediction. The screenshot of the code is mentioned below:

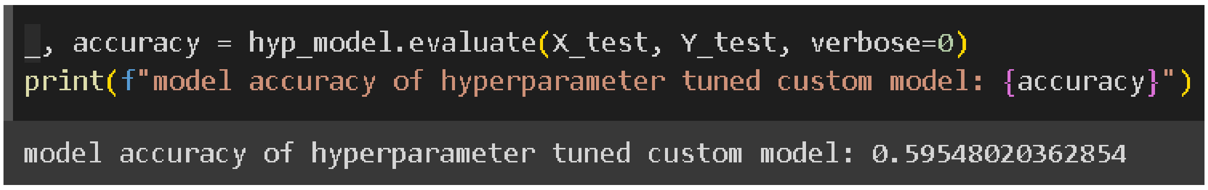


```
y_pred_hyp_tuned_custom_model = (hyp_model.predict(X_test) >= 0.5).astype(int)
```

```
34/34 [==============================] - 0s 10ms/step
```

The hyper tuned custom model accuracy

We can get the accuracy of the model using the .evaluate function. In this case we get the accuracy of the model as: 0.59548020362854, which can be rounded to 0.60 This means that our hyper tuned sequential model is predicting the correct genres of a movie about 60% of the time. The screenshot of the code cell is presented below.

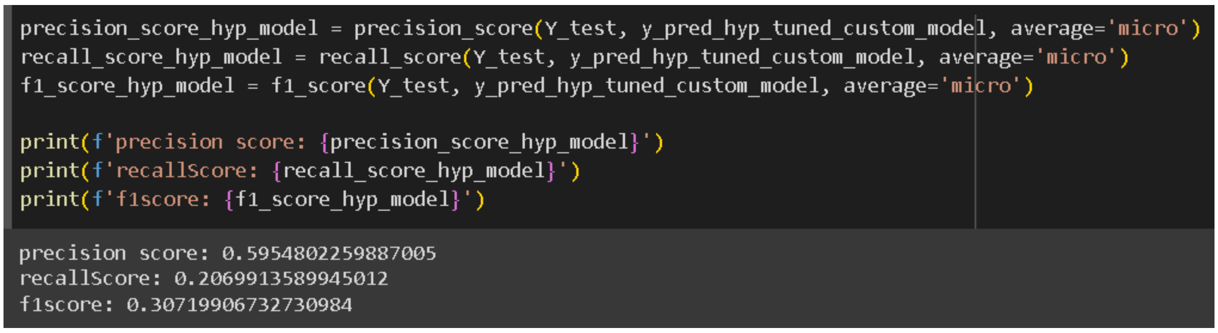


```
_, accuracy = hyp_model.evaluate(X_test, Y_test, verbose=0)
print(f"model accuracy of hyperparameter tuned custom model: {accuracy}")
```

```
model accuracy of hyperparameter tuned custom model: 0.59548020362854
```

The precision, recall and the f1score

The precision score of our hyper tuned sequential model: 0.5954802259887005, this means that our model made about 59.5% of the predictions of the movie genres based of the poster correctly and the remaining 40% of the predictions were false. The recall score of our model is 0.2069913589945012, this suggests that our model was able identify about 20% of the actual correct movie genres, while it identified the 79.3% of actual genres of a movie poster as false negatives. The f1score of our model was: 0.30719906732730984, this means that our model achieved a fair balance between the precision and recall since this score a an average of precision and recall. The screen shot of the code cell is mentioned below:

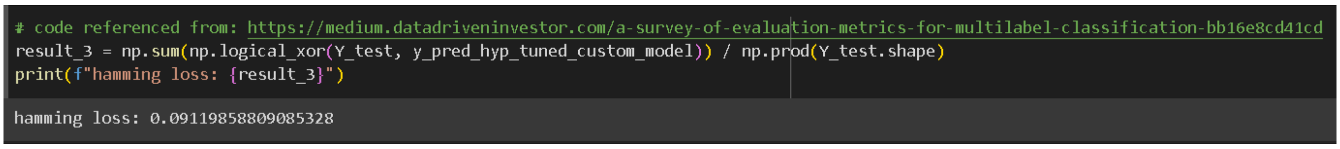


```
precision_score_hyp_model = precision_score(Y_test, y_pred_hyp_tuned_custom_model, average='micro')
recall_score_hyp_model = recall_score(Y_test, y_pred_hyp_tuned_custom_model, average='micro')
f1_score_hyp_model = f1_score(Y_test, y_pred_hyp_tuned_custom_model, average='micro')

print(f'precision score: {precision_score_hyp_model}')
print(f'recallScore: {recall_score_hyp_model}')
print(f'f1score: {f1_score_hyp_model}')
```

```
precision score: 0.5954802259887005
recallScore: 0.2069913589945012
f1score: 0.30719906732730984
```

The hamming Loss

Th hamming loss of our hyper tuned custom sequential model is: 0.09119858809085328. This means that about 9.1% of the genres assigned, based of the movie poster are not correct, so it is incorrectly predicting 9.1% of the genres through each instance. The screenshot of hamming loss is mentioned in below:

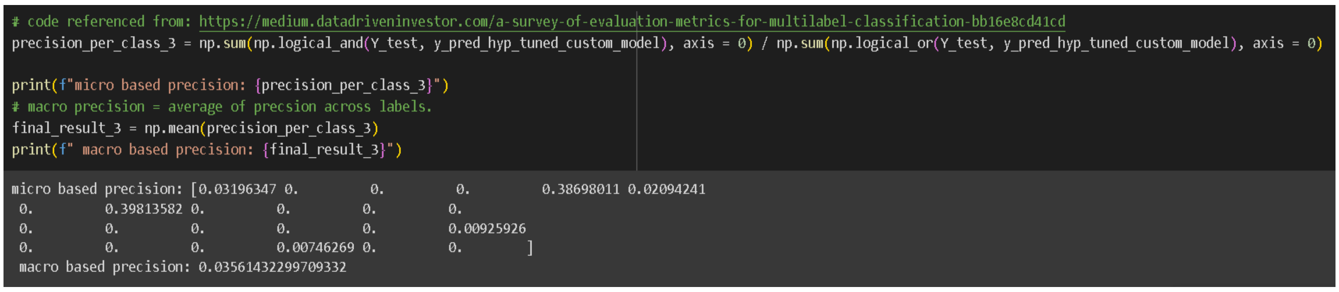


```
# code referenced from: https://medium.datadriveninvestor.com/a-survey-of-evaluation-metrics-for-multilabel-classification-bb16e8cd41cd
result_3 = np.sum(np.logical_xor(Y_test, y_pred_hyp_tuned_custom_model)) / np.prod(Y_test.shape)
print(f"hamming loss: {result_3}")
```

```
hamming loss: 0.09119858809085328
```

The Micro and Macro precision based accuracy

The range from macro based precision accuracy ranges from 0 till 0.398. This means that there were some cases where our model was not able to identify the correct movie posters adhering to that specific genres. This is indicated with the precision value of zero. In the rest of the cases, the model was able to identify the genres of a specific movie poster. The macro-based precision value is: 0.03561432299709332, this means that on average our model is predicting the correct genres of a movie poster about 3.56% f the time. Which is a bit low but it could be improved on further tuning the model. The screenshot of the code and the output is mentioned below.

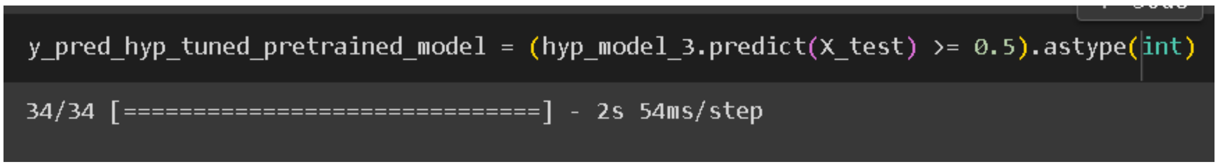


```
# code referenced from: https://medium.datadriveninvestor.com/a-survey-of-evaluation-metrics-for-multilabel-classification-bb16e8cd41cd
precision_per_class_3 = np.sum(np.logical_and(Y_test, y_pred_hyp_tuned_custom_model), axis = 0) / np.sum(np.logical_or(Y_test, y_pred_hyp_tuned_custom_model), axis = 0)

print(f"micro based precision: {precision_per_class_3}")
# macro precision = average of precsion across labels.
final_result_3 = np.mean(precision_per_class_3)
print(f" macro based precision: {final_result_3}")
```

```
micro based precision: [0.03196347 0.         0.         0.         0.38698011 0.02094241
 0.         0.39813582 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.00925926
 0.         0.         0.         0.00746269 0.         0.        ]
 macro based precision: 0.03561432299709332
```

6.4: Performance of the hyper tuned pre-trained model

In this section, we are going to discuss the model performance of the tuned pretrained model. In order to find the model performance, We first have to calculate the predicted probabilities, which we can do using the .predict function. We then repeat the same steps mentioned before to get the prediction. The screenshot of the code is mentioned below:

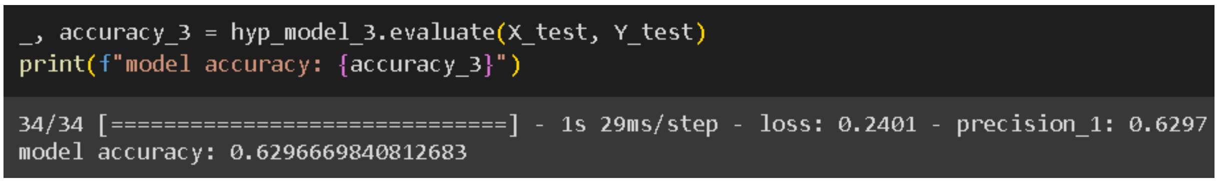


```
y_pred_hyp_tuned_pretrained_model = (hyp_model_3.predict(X_test) >= 0.5).astype(int)
```

```
34/34 [==============================] - 2s 54ms/step
```

Model Accuracy

The accuracy of hyper tuned pretrained model is: 0.6296669840812683, which is about 63%, this means that our pre-trained model is accurately predicted the genres based of the movie poster in about 63% of the time. The rest of the time the model did not predict the correct genres of a movie. The hyper tuned pretrained model performed a bit better than the hyper tuned custom sequential model. The screenshot of the code and the output is mentioned below:

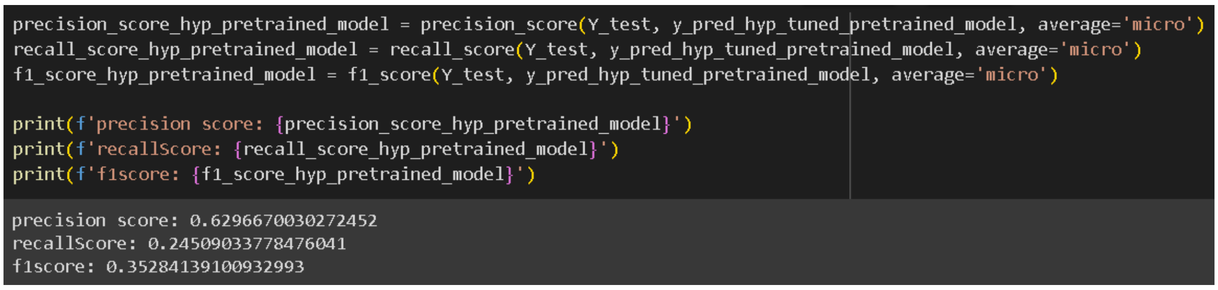


```
_, accuracy_3 = hyp_model_3.evaluate(X_test, Y_test)
print(f"model accuracy: {accuracy_3}")
```

```
34/34 [==============================] - 1s 29ms/step - loss: 0.2401 - precision_1: 0.6297
model accuracy: 0.6296669840812683
```

The precision recall and f1score

The precision recall and the f1score of the model are mentioned in the code snippet below. The precision score of our hyper-tuned pretrained model is: 0.6296670030272452. This means about 63% of the time our model was able to classify the correct genres of a movie based of the movie poster. The recall score of our model is: 0.24509033778476041, which is about 24.5%. This means the our pretrained model was able to identify 24.5% of the movie posters being classified into the correct genres. The overall performance of the model can be deduced from calculating the f1score, which is: 0.35284139100932993. this means the our model was able to perform fairly in predicting the genres based of the movie poster.

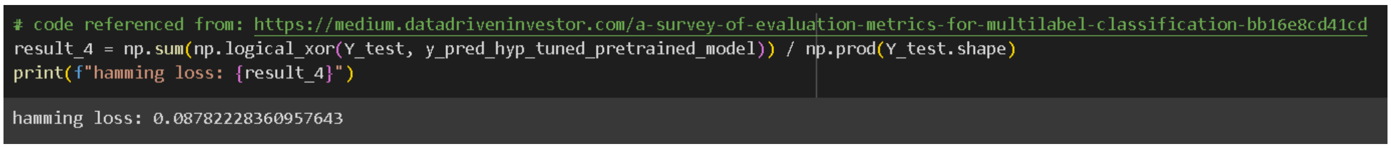


```
precision_score_hyp_pretrained_model = precision_score(Y_test, y_pred_hyp_tuned_pretrained_model, average='micro')
recall_score_hyp_pretrained_model = recall_score(Y_test, y_pred_hyp_tuned_pretrained_model, average='micro')
f1_score_hyp_pretrained_model = f1_score(Y_test, y_pred_hyp_tuned_pretrained_model, average='micro')

print(f'precision score: {precision_score_hyp_pretrained_model}')
print(f'recallScore: {recall_score_hyp_pretrained_model}')
print(f'f1score: {f1_score_hyp_pretrained_model}')
```

```
precision score: 0.6296670030272452
recallScore: 0.24509033778476041
f1score: 0.35284139100932993
```

The Hamming loss

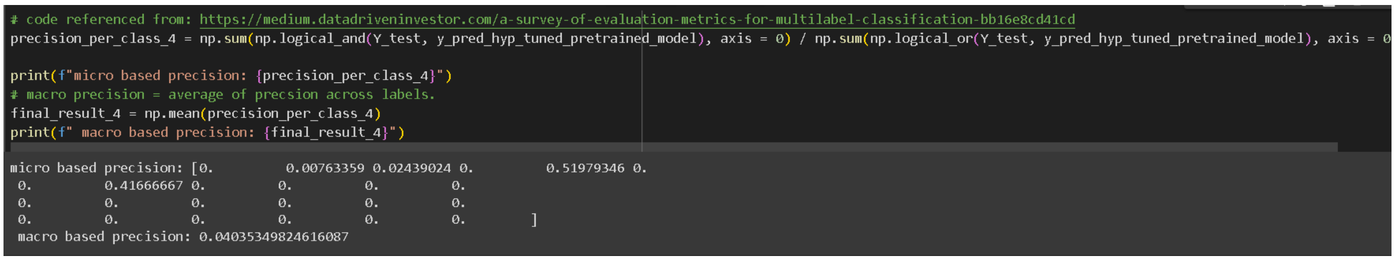
The hamming loss of our model is: 0.08782228360957643. this means that about 8.8% of genres are being incorrectly predicted that do not necessarily belong to the particular movie. The screenshot of the code snippet and the output are mentioned below:



```
# code referenced from: https://medium.datadriveninvestor.com/a-survey-of-evaluation-metrics-for-multilabel-classification-bb16e8cd41cd
result_4 = np.sum(np.logical_xor(Y_test, y_pred_hyp_tuned_pretrained_model)) / np.prod(Y_test.shape)
print(f"hamming loss: {result_4}")

hamming loss: 0.08782228360957643
```

The micro and Macro precision based accuracy

The micro base precision accuracy, we notice that for most instances the precision value is zero. This means the our model was not able to classify the any movie poster belonging to that specific label. There were however some cases where, our model was able to identify correct genres belonging to the movie poster. The macro based precision accuracy states the average precision with classifying the genres to appropriate movie labels is about 4%.

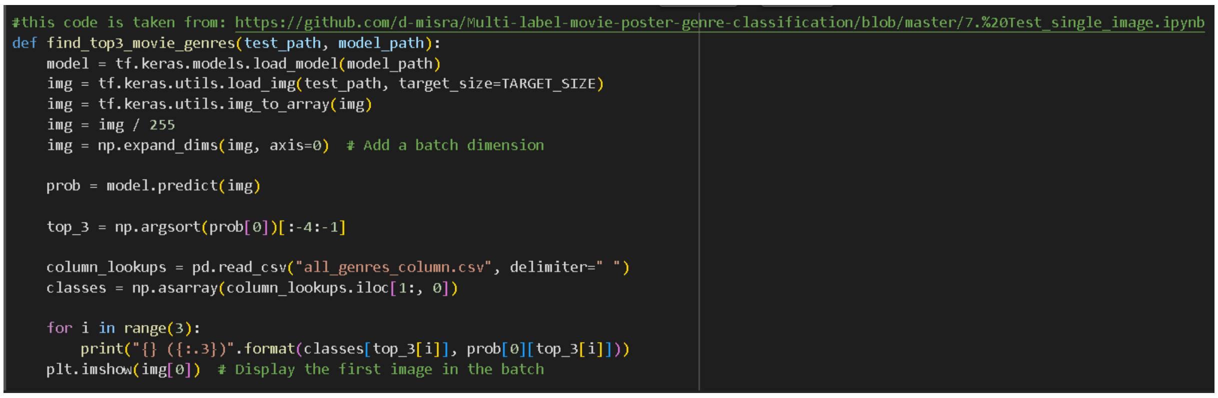


```
# code referenced from: https://medium.datadriveninvestor.com/a-survey-of-evaluation-metrics-for-multilabel-classification-bb16e8cd41cd
precision_per_class_4 = np.sum(np.logical_and(Y_test, y_pred_hyp_tuned_pretrained_model), axis = 0) / np.sum(np.logical_or(Y_test, y_pred_hyp_tuned_pretrained_model), axis = 0

print(f"micro based precision: {precision_per_class_4}")
# macro precision = average of precsion across labels.
final_result_4 = np.mean(precision_per_class_4)
print(f" macro based precision: {final_result_4}")

micro based precision: [0.         0.00763359 0.02439024 0.         0.51979346 0.
 0.         0.41666667 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.         ]
 macro based precision: 0.04035349824616087
```

Movie and genre prediction

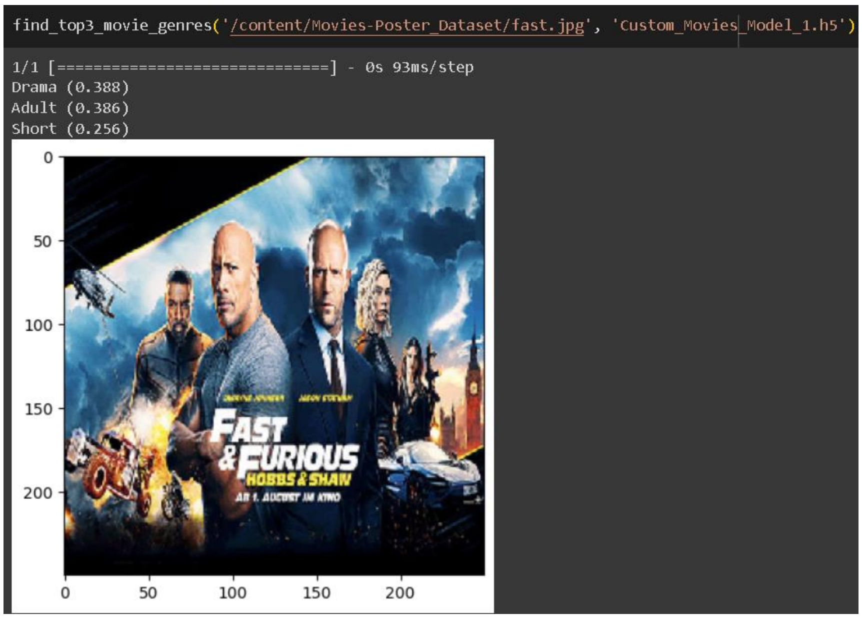
In this section we show the movie poster and top 3 genres of that movie with the custom and pretrained model followed by hyper tuned custom sequential model and hyper tuned pre-trained model. The output of the code is mentioned below

The code created for finding the genre of a movie based its poster image is shown in the screenshot below:



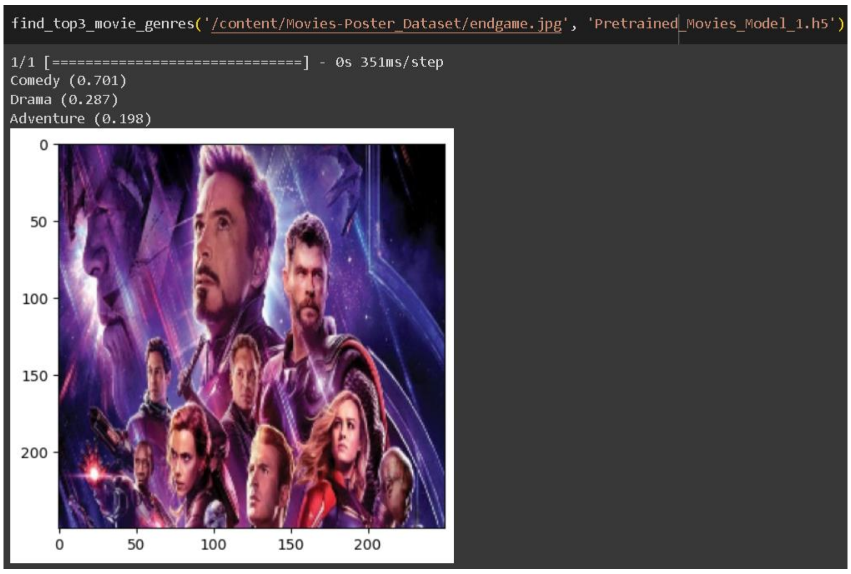
```
#this code is taken from: https://github.com/d-misra/Multi-label-movie-poster-genre-classification/blob/master/7.%20Test_single_image.ipynb
def find_top3_movie_genres(test_path, model_path):
    model = tf.keras.models.load_model(model_path)
    img = tf.keras.utils.load_img(test_path, target_size=TARGET_SIZE)
    img = tf.keras.utils.img_to_array(img)
    img = img / 255
    img = np.expand_dims(img, axis=0)  # Add a batch dimension

    prob = model.predict(img)

    top_3 = np.argsort(prob[0])[:-4:-1]

    column_lookups = pd.read_csv("all_genres_column.csv", delimiter=" ")
    classes = np.asarray(column_lookups.iloc[1:, 0])

    for i in range(3):
        print("{} ({:.3})".format(classes[top_3[i]], prob[0][top_3[i]]))
    plt.imshow(img[0])  # Display the first image in the batch
```

The movie poster and its genres obtained with our custom sequential model is shown below. The top three genres of movie fast and furious Hobbs and Shaw are drama with the prediction of 38.8% Adult with prediction of 38.6% and short with prediction of 25.6%.



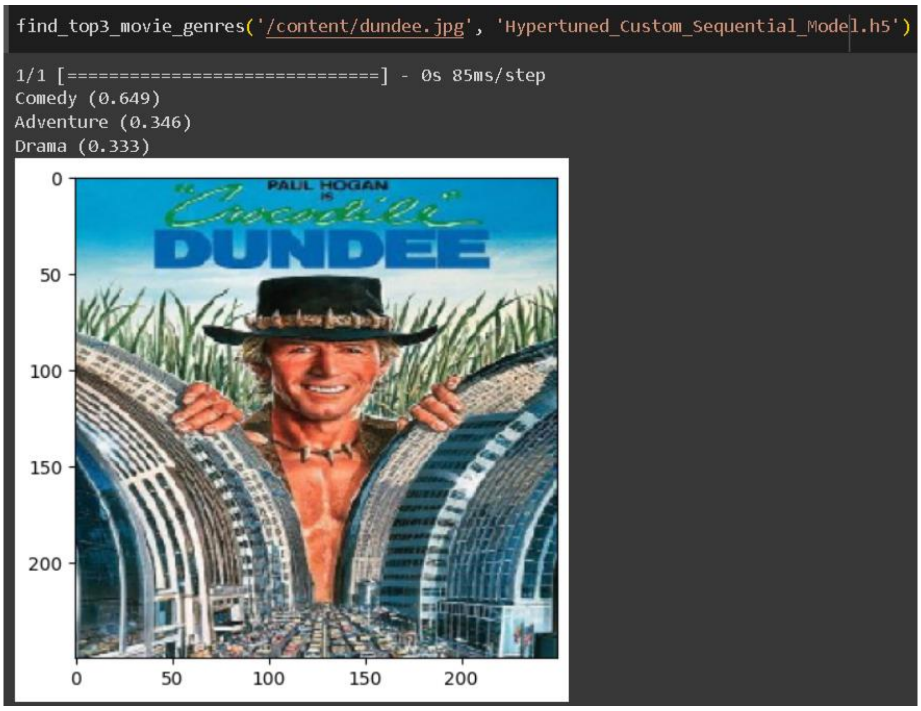
```
find_top3_movie_genres('/content/Movies-Poster_Dataset/fast.jpg', 'Custom_Movies_Model_1.h5')

1/1 [==============================] - 0s 93ms/step
Drama (0.388)
Adult (0.386)
Short (0.256)
```

The movie poster and its genres obtained with our pretrained model is shown below. The top three movie genres of the movie avengers the end game are comedy with the prediction of 70.1%, Drama with prediction 28.7% and Adventure with prediction 19.8%.



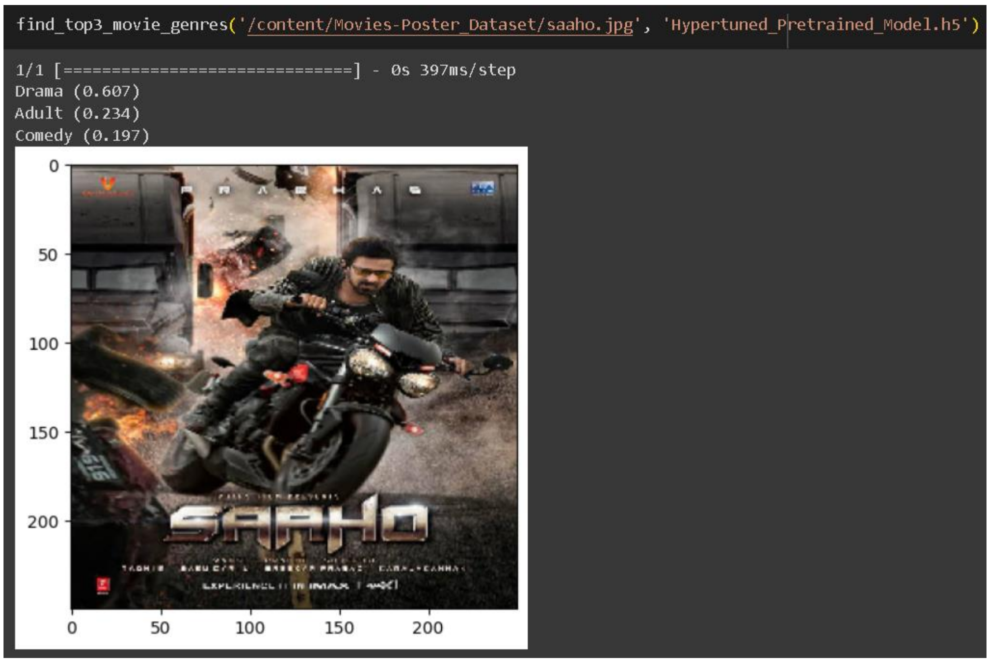
```
find_top3_movie_genres('/content/Movies-Poster_Dataset/endgame.jpg', 'Pretrained_Movies_Model_1.h5')

1/1 [==============================] - 0s 351ms/step
Comedy (0.701)
Drama (0.287)
Adventure (0.198)
```

The movie poster and its genres obtained from running the custom sequential model is shown below. The output predicts that from this movie poster the movie contains the genre 64.9% as comedy 34.6% as adventure and 33.3% as drama.



```
find_top3_movie_genres('/content/dundee.jpg', 'Hypertuned_Custom_Sequential_Model.h5')

1/1 [==============================] - 0s 85ms/step
Comedy (0.649)
Adventure (0.346)
Drama (0.333)
```

The movie poster and its genres obtained from running the hyper tuned pretrained model is shown below. The output below gives out the top 3 genres of the Bollywood movie saaho. I predicts the genre drama with the prediction of about 60%, Adult genre 23.4% and the comedy genre 19.7% from the movie poster. Overall we can also observe that after applying hyperparameter tuning there is a slight increase in the accuracy of classification of movie genres.



```
find_top3_movie_genres('/content/Movies-Poster_Dataset/saaho.jpg', 'Hypertuned_Pretrained_Model.h5')

1/1 [==============================] - 0s 397ms/step
Drama (0.607)
Adult (0.234)
Comedy (0.197)
```

Sources

Rohini.G. (2021, September 23). Everything you need to know about VGG16. Medium. Retrieved from: https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5918

Wadekar.S. (2021, January 10). Hyperparameter Tuning in Keras: TensorFlow 2: With Keras Tuner: RandomSearch, Hyperband, BayesianOptimization. Medium. Retrieved from: https://medium.com/swlh/hyperparameter-tuning-in-keras-tensorflow-2-with-keras-tuner-randomsearch-hyperband-3e212647778f