FINAL PROJECT CSCI E-82: TRANSFORMERS AND CNNs utilized to Predict Animated Media Type

by **YUVRAJ PURI**

```python
import pandas as pd
import numpy as np

import warnings
import seaborn as sns

from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

warnings.filterwarnings('ignore')
```

DATA ANALYSIS

```python
df = pd.read_csv("MAL-anime.csv")

df
```


{"summary":"{\n \"name\": \"df\",\n \"rows\": 12774,\n \"fields\": [\n   {\n   \"column\": \"Unnamed: 0\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 3687,\n   \"min\": 0,\n   \"max\": 12773,\n   \"num_unique_values\": 12774,\n   \"samples\": [\n   10284,\n   5461,\n   1010\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Title\",\n   \"properties\": {\n   \"dtype\": \"string\",\n   \"num_unique_values\": 12774,\n   \"samples\": [\n   \"Tanken Driland\",\n   \"Oyayubi Hime Monogatari\",\n   \"Captain Tsubasa\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Rank\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 3690,\n   \"min\": 1,\n   \"max\": 12788,\n   \"num_unique_values\": 12774,\n   \"samples\": [\n   5886,\n   5473,\n   2334\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Type\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 6,\n

```
\"samples\": [\n          \"TV\",\n          \"Movie\",\n
\"Unknown\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Episodes\",\n      \"properties\": {\n        \"dtype\":
\"category\",\n        \"num_unique_values\": 193,\n
\"samples\": [\n          \"67\",\n          \"66\",\n
\"79\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Aired\",\n      \"properties\": {\n        \"dtype\": \"category\",\
n        \"num_unique_values\": 3631,\n        \"samples\": [\n
\"Jul 2002 - Jul 2002\",\n          \"Jun 2000 - Nov 2001\",\n
\"Apr 1989 - May 1991\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"Members\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 214094,\n        \"min\": 181,\n
\"max\": 3759013,\n        \"num_unique_values\": 9555,\n
\"samples\": [\n          1186,\n          2566,\n          6703\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"page_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12774,\n        \"samples\": [\n
\"https://myanimelist.net/anime/14333/Tanken_Driland\",\n
\"https://myanimelist.net/anime/2783/Oyayubi_Hime_Monogatari\",\n
\"https://myanimelist.net/anime/2116/Captain_Tsubasa\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"image_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12770,\n        \"samples\": [\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/1421/100770.jpg?
s=b04bd7f7f29e244145365bef7c768b93\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/5/64773.jpg?
s=c502e9d53f1668cd236870480d81bdef\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/1224/98799.jpg?
s=777e272ba1b859af11160cda8f72047a\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Score\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\": 0.942194573739637,\n
\"min\": 1.85,\n        \"max\": 9.1,\n        \"num_unique_values\":
564,\n        \"samples\": [\n          8.65,\n          3.31,\n
5.81\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"df"}
```


Here's what the columns refer to:

- `Title`: Name of the anime
- `Rank`: Ranking of the anime
- `Type`: Category of anime e.g. TV, ONA, Movie, Special, etc.
- `Episodes`: Number of episodes of the anime

- `Aired`: Date of airing of an anime
- `Members`: Number of members who have watched/read the anime
- `page_url`: The URL link to the page of the particular anime
- `image_url`: The URL link to the cover image of the particular anime
- `Score`: Average user rating/score of the anime

```
df.dtypes
```

```
Unnamed: 0      int64
Title          object
Rank            int64
Type           object
Episodes       object
Aired          object
Members         int64
page_url       object
image_url      object
Score         float64
dtype: object
```

```
print(df.Type.value_counts(), "\n")
print(df.Type.value_counts() / df.shape[0] * 100)
```

```
Type
TV         4510
Movie      2485
Special    2014
ONA        1883
OVA        1881
Unknown       1
Name: count, dtype: int64

Type
TV         35.306090
Movie      19.453578
Special    15.766401
ONA        14.740880
OVA        14.725223
Unknown     0.007828
Name: count, dtype: float64
```

The majority of the works are TV. We have a single Unknown classification which we shall drop to avoid confounding the data as there is only a single instance of it.

```
print(df[df['Type'] == 'Unknown'])
```

```
      Unnamed: 0                Title  Rank     Type Episodes  \
6260        6260  Sekai Meisaku Douwa  8611  Unknown       20

                    Aired  Members  \
```

```
6260  Oct 1975 - Feb 1983      735

                                                page_url  \
6260  https://myanimelist.net/anime/7398/Sekai_Meisa...

                                               image_url  Score
6260  https://cdn.myanimelist.net/r/100x140/images/a...   6.06

# Drop rows where 'Type' is 'Unknown'
df = df[df['Type'] != 'Unknown']

# Reset the index
df.reset_index(drop=True, inplace=True)

df['Type'].value_counts()

Type
TV         4510
Movie      2485
Special    2014
ONA        1883
OVA        1881
Name: count, dtype: int64
```

Successfully dropped.

Gathering information on the data and adjusting it

Aired

```
df[df['Aired'].str.contains("-")].shape

(12773, 10)
```

All the dates are given in a range format (e.g. March 2017 - March 2018). Therefore, we can use it as the delimiter for the dates for a date-time conversion.

```
df['Start_year'] = df['Aired'].apply(lambda x: x.split('-')[0].strip()
[:])
df['End_year'] = df['Aired'].apply(lambda x: x.split('-')[1].strip()
[:])

df.head()
```


```
{"summary":"{\n  \"name\": \"df\",\n  \"rows\": 12773,\n  \"fields\":
[\n    {\n      \"column\": \"Unnamed: 0\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 3687,\n        \"min\": 0,\n
\"max\": 12773,\n        \"num_unique_values\": 12773,\n
\"samples\": [\n          10284,\n          5461,\n          1010\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
```

```
}\n    },\n    {\n      \"column\": \"Title\",\n      \"properties\":
{\n        \"dtype\": \"string\",\n        \"num_unique_values\":
12773,\n        \"samples\": [\n          \"Tanken Driland\",\n
\"Oyayubi Hime Monogatari\",\n          \"Captain Tsubasa\"\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Rank\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 3690,\n        \"min\": 1,\n        \"max\": 12788,\n
\"num_unique_values\": 12773,\n        \"samples\": [\n
5886,\n          5473,\n          2334\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Type\",\n      \"properties\": {\n
\"dtype\": \"category\",\n        \"num_unique_values\": 5,\n
\"samples\": [\n          \"Movie\",\n          \"OVA\",\n
\"Special\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Episodes\",\n      \"properties\": {\n        \"dtype\":
\"category\",\n        \"num_unique_values\": 193,\n
\"samples\": [\n          \"67\",\n          \"66\",\n
\"79\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Aired\",\n      \"properties\": {\n        \"dtype\": \"category\",\
n        \"num_unique_values\": 3630,\n        \"samples\": [\n
\"Jul 2002 - Jul 2002\",\n          \"Jun 2018 - Apr 2019\",\n
\"Apr 1989 - May 1991\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"Members\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 214102,\n        \"min\": 181,\n
\"max\": 3759013,\n        \"num_unique_values\": 9555,\n
\"samples\": [\n          1186,\n          2566,\n          6703\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"page_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12773,\n        \"samples\": [\n
\"https://myanimelist.net/anime/14333/Tanken_Driland\",\n
\"https://myanimelist.net/anime/2783/Oyayubi_Hime_Monogatari\",\n
\"https://myanimelist.net/anime/2116/Captain_Tsubasa\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"image_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12769,\n        \"samples\": [\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/1421/100770.jpg?
s=b04bd7f7f29e244145365bef7c768b93\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/5/64773.jpg?
s=c502e9d53f1668cd236870480d81bdef\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/1224/98799.jpg?
s=777e272ba1b859af11160cda8f72047a\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Score\",\n      \"properties\": {\
```

```
n        \"dtype\": \"number\",\n        \"std\": 0.9422242904802463,\
n        \"min\": 1.85,\n        \"max\": 9.1,\n
\"num_unique_values\": 564,\n        \"samples\": [\n          8.65,\n
3.31,\n          5.81\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Start_year\",\n      \"properties\": {\n        \"dtype\":
\"object\",\n        \"num_unique_values\": 814,\n        \"samples\":
[\n          \"Aug 2022\",\n          \"Oct 1970\",\n          \"Jul
2016\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"End_year\",\n      \"properties\": {\n        \"dtype\":
\"object\",\n        \"num_unique_values\": 810,\n        \"samples\":
[\n          \"Oct 2021\",\n          \"1993\",\n          \"Dec
2013\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"df"}
```



```
print(df[df['Start_year'] == ''].index.shape)
print(df[df['End_year'] == ''].index.shape)
```

```
(8,)
(113,)
```

The ones that have issues.

```
nostart = df[df['Start_year'] == ''].index.tolist()
noend = df[df['End_year'] == ''].index.tolist()
merged_list = list(set(nostart) | set(noend))
print("Merged without duplicates:", len(merged_list))
```

```
Merged without duplicates: 113
```

```
df.shape
```

```
(12773, 12)
```

```
df = df.drop(merged_list)
df.reset_index(drop=True, inplace=True)
```

```
df.shape
```

```
(12660, 12)
```

We've successfully dropped the shows with invalid dates.

Rank

```
print(min(df['Rank']))
print(max(df['Rank']))
```

```
1
12788
```

```python
df['Rank'].isna().sum()
```

```
0
```

```python
df[df['Rank'] == ''].shape
```

```
(0, 12)
```

No missing ranks but there's a discrepancy in the ranks given and the max rank.

```python
df[df['Rank'] > 12774]
```

{"repr_error":"0","type":"dataframe"}

For some reaosn, there's a discrepancy in the ranks and the number of items in the dataframe.

```python
# Check for duplicate ranks
duplicate_ranks = df[df.duplicated(subset='Rank', keep=False)]
print(f"Duplicate ranks:\n{duplicate_ranks}")
```

```
Duplicate ranks:
Empty DataFrame
Columns: [Unnamed: 0, Title, Rank, Type, Episodes, Aired, Members,
page_url, image_url, Score, Start_year, End_year]
Index: []
```

There are no duplicated ranks so no ties in the ranks. That means some rankings are outright missing.

```python
# Generate the full range of expected ranks
expected_ranks = set(range(1, 12775))  # Full range from 1 to 12774

# Extract the unique ranks from the dataframe
actual_ranks = set(df['Rank'].dropna().astype(int))  # Ensure ranks
are integers and non-NaN

# Find the missing (skipped) ranks
missing_ranks = expected_ranks - actual_ranks

if missing_ranks:
    print(f"Ranks below 12774 that are missing:
{sorted(missing_ranks)}")
else:
    print("No ranks below 12774 are missing.")
```

```
Ranks below 12774 that are missing: [55, 57, 201, 267, 301, 312, 379,
501, 506, 654, 751, 868, 936, 965, 1051, 1181, 1197, 1256, 1274, 1401,
```

```
1600, 1605, 1730, 1782, 1903, 1920, 2028, 2051, 2086, 2161, 2260,
2351, 2398, 2525, 2596, 2621, 2676, 2701, 2710, 2732, 2794, 2795,
2802, 2981, 3087, 3091, 3105, 3330, 3551, 3645, 3653, 3888, 3906,
3924, 3933, 3939, 3948, 4120, 4158, 4314, 4419, 4486, 4577, 4668,
4721, 4725, 4763, 4790, 4811, 4850, 4885, 4945, 5001, 5175, 5292,
5371, 5376, 5419, 5504, 5586, 5770, 5822, 5851, 6145, 6246, 6374,
6729, 7240, 7308, 7327, 7569, 7586, 7734, 7883, 8261, 8297, 8386,
8611, 8804, 8915, 9053, 9159, 9216, 9243, 9379, 9383, 9708, 9780,
9859, 9879, 10009, 10051, 10104, 10136, 10208, 10325, 10369, 10387,
10406, 10539, 10962, 11053, 11222, 11410, 11432, 11830, 12051, 12122]
```

These are the ranks that have gone missing. Some of these are a consequence of dropping the shows with missing airing data. We've already verified that the ranks above 12774 are present. With that said, this isn't an issue to worry about.

Episodes, Members, and Scores

We'll gather some univariate data on these entries. Before we do that, for Episodes, we'll convert it to numbers.

```
df['Episodes'] = pd.to_numeric(df['Episodes'], errors='coerce')
```

Making dates usable

```
# Convert 'Aired' to datetime
df['Start_year'] = pd.to_datetime(df['Start_year'], errors='coerce',
format=None)
df['End_year'] = pd.to_datetime(df['End_year'], errors='coerce',
format=None)

# Check the result
print(df['Start_year'])

0       2017-10-01
1       1997-07-01
2       2015-01-01
3       2001-07-01
4       2018-12-01
           ...
12655   2002-07-01
12656   2018-03-01
12657   2006-11-01
12658   2021-03-01
12659   1997-04-01
Name: Start_year, Length: 12660, dtype: datetime64[ns]
```

Looking at these examples, we see that these have a '-' for their airing date. This is because their airing period is are actually unknown when relying on only the data from MyAnimeList - it isn't listed. Therefore, we will remove these entries from the dataset as well.

Now we will fix the dates.

```
df.rename(columns={'Unnamed: 0': 'id'}, inplace=True)
df.columns
```

```
Index(['id', 'Title', 'Rank', 'Type', 'Episodes', 'Aired', 'Members',
       'page_url', 'image_url', 'Score', 'Start_year', 'End_year'],
      dtype='object')
```

```
# 1. Adjust End_date if it matches Start_date
# If same, make the end date the last date of the month (so run time
isn't 0)
# If not, keep the original end date.
df['End_year'] = np.where(
    df['Start_year'] == df['End_year'],
    df['Start_year'] + pd.offsets.MonthEnd(0),
    df['End_year']
)

df.head()
```


```
{"summary":"{\n  \"name\": \"df\",\n  \"rows\": 12660,\n  \"fields\":
[\n    {\n      \"column\": \"id\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 3689,\n        \"min\": 0,\n
\"max\": 12773,\n        \"num_unique_values\": 12660,\n
\"samples\": [\n          1240,\n          8925,\n          11386\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"Title\",\n      \"properties\":
{\n        \"dtype\": \"string\",\n        \"num_unique_values\":
12660,\n        \"samples\": [\n          \"Mikakunin de
Shinkoukei\",\n          \"Fushigi na Somera-chan: Hajimatteru yo!
Sono Ato no Somera-chan!!\",\n          \"Dr. Slump: Arale-chan
Specials\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Rank\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 3692,\n        \"min\": 1,\n        \"max\": 12788,\n
\"num_unique_values\": 12660,\n        \"samples\": [\n
2221,\n          11391,\n          9875\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Type\",\n      \"properties\": {\n
\"dtype\": \"category\",\n        \"num_unique_values\": 5,\n
\"samples\": [\n          \"Movie\",\n          \"OVA\",\n
\"Special\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Episodes\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 52,\n        \"min\": 1,\n
\"max\": 3057,\n        \"num_unique_values\": 187,\n
\"samples\": [\n          1818,\n          75,\n          76\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
```

```
\"Aired\",\n      \"properties\": {\n        \"dtype\": \"category\",\
n        \"num_unique_values\": 3572,\n        \"samples\": [\n
\"Mar 2018 - Mar 2018\",\n          \"Oct 2017 - Oct 2019\",\n
\"Apr 1983 - Mar 1984\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"Members\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 214125,\n        \"min\": 181,\n
\"max\": 3759013,\n        \"num_unique_values\": 9489,\n
\"samples\": [\n          1783,\n          4730,\n          69966\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"page_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12660,\n        \"samples\": [\n
\"https://myanimelist.net/anime/20541/Mikakunin_de_Shinkoukei\",\n
\"https://myanimelist.net/anime/33659/Fushigi_na_Somera-
chan__Hajimatteru_yo_Sono_Ato_no_Somera-chan\",\n
\"https://myanimelist.net/anime/28369/Dr_Slump__Arale-chan_Specials\"\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"image_url\",\n      \"properties\": {\n        \"dtype\":
\"string\",\n        \"num_unique_values\": 12658,\n
\"samples\": [\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/10/75249.jpg?
s=f2a73e150b488ff392d01b2c7aeffd90\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/3/59141.jpg?
s=8796949469fad34b441826ebd3e7b0b3\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/10/89781.jpg?
s=d5e9193d4bba8247a505dfdefbb02c59\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Score\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\": 0.9429713618087816,\
n        \"min\": 1.85,\n        \"max\": 9.1,\n
\"num_unique_values\": 564,\n        \"samples\": [\n          8.65,\n
3.31,\n          5.81\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Start_year\",\n      \"properties\": {\n        \"dtype\":
\"date\",\n        \"min\": \"1917-05-01 00:00:00\",\n        \"max\":
\"2023-07-01 00:00:00\",\n        \"num_unique_values\": 749,\n
\"samples\": [\n          \"1983-05-01 00:00:00\",\n          \"1987-
03-01 00:00:00\",\n          \"1993-03-01 00:00:00\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"End_year\",\n      \"properties\":
{\n        \"dtype\": \"date\",\n        \"min\": \"1917-05-31
00:00:00\",\n        \"max\": \"2023-07-31 00:00:00\",\n
\"num_unique_values\": 1273,\n        \"samples\": [\n
\"1991-09-01 00:00:00\",\n          \"1933-04-30 00:00:00\",\n
\"2020-12-01 00:00:00\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"df"}
```

```python
# Next get a duration or airtime for the show
df['airtime (days)'] = (df['End_year'] - df['Start_year']).dt.days

df[['Title', 'Aired', 'airtime (days)']]
```

{"summary":"{\n \"name\": \"df[['Title', 'Aired', 'airtime (days)']]\",\n \"rows\": 12660,\n \"fields\": [\n   {\n   \"column\": \"Title\",\n   \"properties\": {\n   \"dtype\": \"string\",\n   \"num_unique_values\": 12660,\n   \"samples\": [\n   \"Mikakunin de Shinkoukei\",\n   \"Fushigi na Somera-chan: Hajimatteru yo! Sono Ato no Somera-chan!!\",\n   \"Dr. Slump: Arale-chan Specials\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Aired\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 3572,\n   \"samples\": [\n   \"Mar 2018 - Mar 2018\",\n   \"Oct 2017 - Oct 2019\",\n   \"Apr 1983 - Mar 1984\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"airtime (days)\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 292,\n   \"min\": -243,\n   \"max\": 9466,\n   \"num_unique_values\": 244,\n   \"samples\": [\n   244,\n   335,\n   4108\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}

Make sure there aren't any clerical errors for the airtime.

```python
df[df['airtime (days)'] < 0]
```

{"summary":"{\n \"name\": \"df[df['airtime (days)'] < 0]\",\n \"rows\": 9,\n \"fields\": [\n   {\n   \"column\": \"id\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 3999,\n   \"min\": 651,\n   \"max\": 11491,\n   \"num_unique_values\": 9,\n   \"samples\": [\n   9500,\n   1144,\n   8896\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Title\",\n   \"properties\": {\n   \"dtype\": \"string\",\n   \"num_unique_values\": 9,\n   \"samples\": [\n   \"Tesla Note: Mickey to Oliver no Agent Yousei Kouza\",\n   \"Bakugan Battle Brawlers: Mechtanium Surge\",\n   \"Estima: Sense of Wonder\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Rank\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 3709,\n   \"min\": 7,\n   \"max\": 12512,\n   \"num_unique_values\": 9,\n   \"samples\": [\n   11337,\n   7803,\n   12512\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Type\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 3,\n   \"samples\": [\n

```
\"Special\",\n          \"ONA\",\n          \"TV\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Episodes\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\": 13,\n
\"min\": 2,\n        \"max\": 46,\n        \"num_unique_values\": 8,\n
\"samples\": [\n          46,\n          13,\n          8\n        ],\
n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"Aired\",\n      \"properties\":
{\n        \"dtype\": \"string\",\n        \"num_unique_values\": 9,\n
\"samples\": [\n          \"Sep 2021 - Jan 2021\",\n          \"Sep
2018 - 2018\",\n          \"Jun 2016 - 2016\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Members\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\": 147355,\n
\"min\": 630,\n        \"max\": 446872,\n
\"num_unique_values\": 9,\n        \"samples\": [\n          673,\n
44262,\n          703\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"page_url\",\n      \"properties\": {\n        \"dtype\":
\"string\",\n        \"num_unique_values\": 9,\n        \"samples\":
[\n
\"https://myanimelist.net/anime/50258/Tesla_Note__Mickey_to_Oliver_no_
Agent_Yousei_Kouza\",\n
\"https://myanimelist.net/anime/10330/Bakugan_Battle_Brawlers__Mechtan
ium_Surge\",\n
\"https://myanimelist.net/anime/33871/Estima__Sense_of_Wonder\"\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"image_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 9,\n        \"samples\": [\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/1774/120810.jpg?
s=633c5004a65fb24a81c178233ba1dd91\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/10/82083.jpg?
s=af3ea16416ca91b7aeeeb452de8b0d35\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/13/81540.jpg?
s=d9b3875fa6438357237a96086c907447\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Score\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\": 1.2548085289973305,\
n        \"min\": 4.58,\n        \"max\": 9.05,\n
\"num_unique_values\": 9,\n        \"samples\": [\n          5.34,\n
6.22,\n          4.58\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Start_year\",\n      \"properties\": {\n        \"dtype\":
\"date\",\n        \"min\": \"2001-02-01 00:00:00\",\n        \"max\":
\"2023-03-01 00:00:00\",\n        \"num_unique_values\": 9,\n
\"samples\": [\n          \"2021-09-01 00:00:00\",\n          \"2018-
09-01 00:00:00\",\n          \"2016-06-01 00:00:00\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
```

```
n    },\n    {\n      \"column\": \"End_year\",\n      \"properties\":
{\n        \"dtype\": \"date\",\n        \"min\": \"2001-01-01
00:00:00\",\n        \"max\": \"2023-01-01 00:00:00\",\n
\"num_unique_values\": 9,\n        \"samples\": [\n          \"2021-
01-01 00:00:00\",\n          \"2018-01-01 00:00:00\",\n
\"2016-01-01 00:00:00\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"airtime (days)\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 77,\n        \"min\": -243,\n
\"max\": -31,\n        \"num_unique_values\": 8,\n        \"samples\":
[\n          -243,\n          -152,\n          -151\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe"}
```


A few errors have popped up because the end year was recorded as a year only and thus the airtime calculator gave us a negative value.

Another two were because of misrecordings of their run dates - id. 11491 took place from December 2008 on to August 2009 and id. 9500 took place from Sep 2021 to Jan 2022.

These will be adjusted by re-setting the dates to follow the setup we have had.

```
id_list = [651, 1144,2565,5883,8322,8896,9018]

# Readjust 'Start_year' and 'End_year' to datetime objects
df['Start_year'] = pd.to_datetime(df['Start_year'], errors='coerce')
df['End_year'] = pd.to_datetime(df['End_year'], errors='coerce')


df['End_year'] = np.where(
    df['id'].isin(id_list),
    df['Start_year'] + pd.offsets.YearEnd(0),  # Adjust to YearEnd
instead of MonthEnd
    df['End_year']
)

df.loc[df['id'] == 9500, 'End_year'] = pd.to_datetime('2022-01-31')
df.loc[df['id'] == 11491, 'Start_year'] = pd.to_datetime('2008-12-01')

df['airtime (days)'] = (df['End_year'] - df['Start_year']).dt.days
```

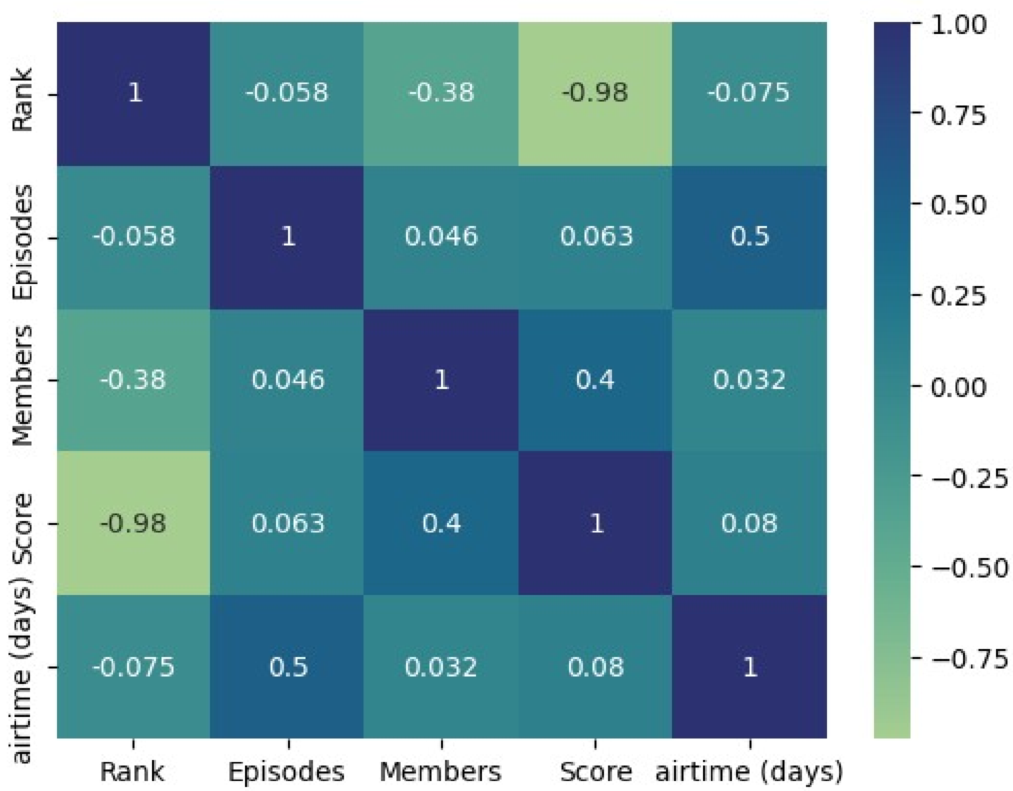
EDA

```
df['Type'].value_counts()
```

```
Type
TV         4448
Movie      2480
Special    2007
```

```
OVA        1875
ONA        1850
Name: count, dtype: int64
```

```
sns.heatmap(df[['Rank', 'Episodes','Members', 'Score', 'airtime
(days)']].corr(), annot=True, cmap='crest');
```



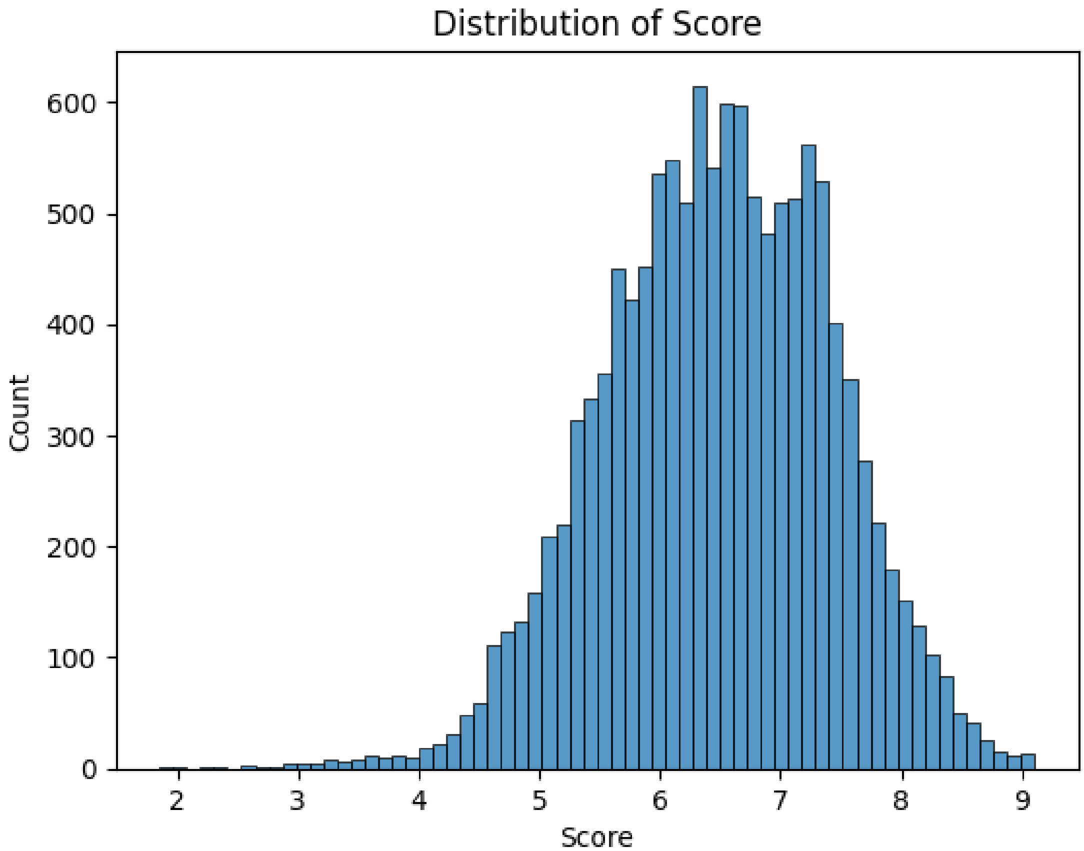
Basic insights - there's a strong negative correlation with Rank and Score (which makes sense - the higher the score, the lower the number for evaluating rank is). We also notice a somewhat positive correlation between Member quantity and score as well as a somewhat negative correlation between Rank and Member count.

To an extent, as the rank goes down, the number of Members - people who have watched the show - go up. To an extent, as the score goes up, the number of people who have watched the show also does.

There appears to be little to no correlation between airtime and these other variables besides Episodes. As airtime increases, to an extent, so does episodes which makes sense. More airtime means that there's more room for the episodes of a show to increase (which is characteristic of the TV shows). Episode count, however, doesn't have much of a correlation with the other variables.

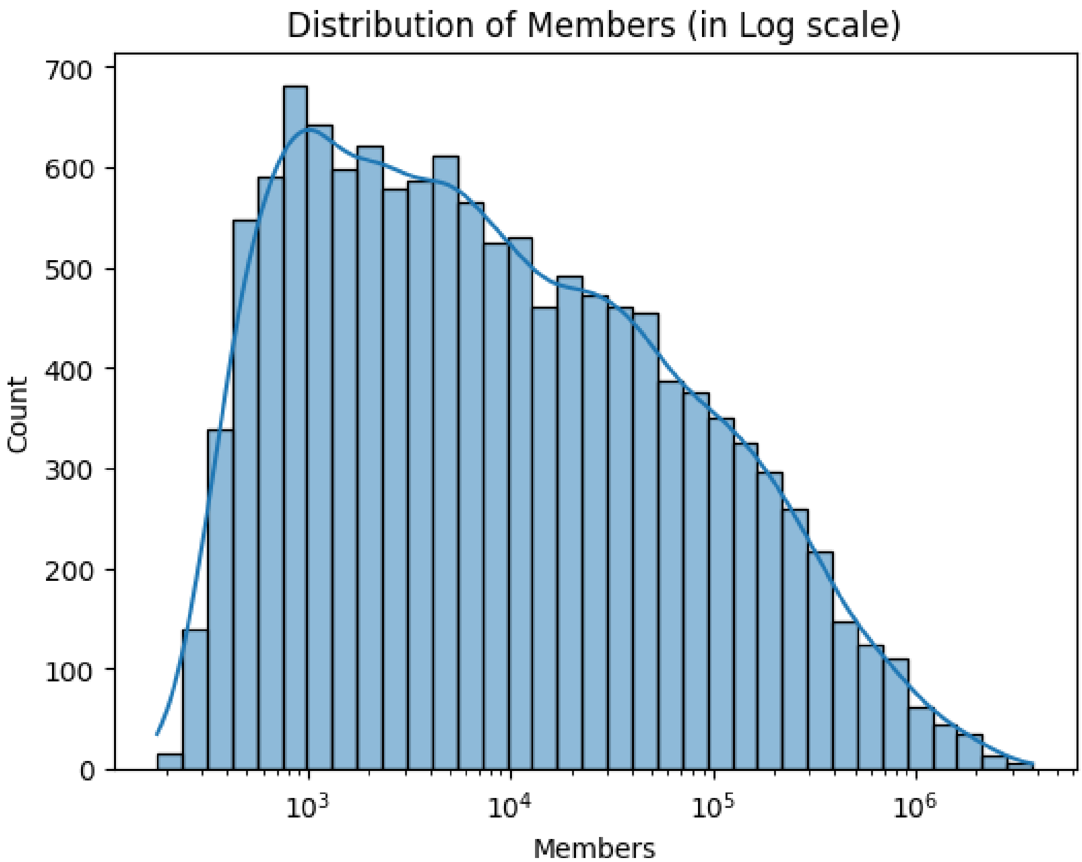
```
%matplotlib inline
import matplotlib.pyplot as plt
sns.histplot(x='Score', data=df)
```

```
plt.title("Distribution of Score")
plt.show()
```



A majority of the scores rating anime (in this dataset) are to be within the 6-7 range. This will be reflected in the univariate statistics calculated below. There also appears to be more of a slight left-sided skew in the Score data here.

```
sns.histplot(x='Members', data=df, log_scale=True, kde=True)
plt.title("Distribution of Members (in Log scale)")
plt.show()
```



Looking at the distribution of members in log scale, there are few shows with incredibly large viewerbases that are registered on MAL compared to the shows with smaller viewerbases - as we see the mode or most frequent count can be found below 10^3 or so. More and more fall between 1000 and 10000 but its not a very steep decline in the distribution.

There's a rightward skew in the distribution of log-transformed member counts for the show, it's not a particularly normal distribution.

```
columns_to_analyze = ['Episodes', 'Members', 'Score', 'airtime
(days)']

# Initialize a dictionary to hold the data
univariate_stats = {}

# Populate the dictionary with statistics for each column
for column in columns_to_analyze:
    univariate_stats[column] = {
        'mean': df[column].mean(),
        'median': df[column].median(),
        'std': df[column].std(),
        'min': df[column].min(),
        'max': df[column].max(),
        'count': df[column].count(),
```

```
        'unique_values': df[column].nunique(),
    }
```

```
pip install tabulate
```

```
Requirement already satisfied: tabulate in
/usr/local/lib/python3.10/dist-packages (0.9.0)
```

```
from tabulate import tabulate

# Example: Displaying the univariate_stats dictionary in a table
format
headers = ["Statistic", "mean", "median", "std", "min", "max",
"count", "unique_values"]
rows = []

# Loop through the univariate_stats dictionary to prepare the rows
# Loop through the dictionary and prepare rows for each statistic
for stat, values in univariate_stats.items():
    rows.append([stat] + list(values.values()))  # Concatenate the
stat name with its values

# Print the table
print(tabulate(rows, headers=headers, tablefmt='grid'))
```

```
+-----------------+-------------+----------+---------------+--------
+-----------------+---------+-----------------+
| Statistic       |        mean |   median |           std |    min |
max |   count |   unique_values |
+=================+=============+==========+===============+========+==
==============+=========+=================+
| Episodes        |    13.3543  |      3   |     52.997    |   1    |
3057           |   12660 |             187 |
+-----------------+-------------+----------+---------------+--------
+-----------------+---------+-----------------+
| Members         | 71021.6     |   6572.5 | 214125        | 181    |
3.75901e+06 |   12660 |            9489 |
+-----------------+-------------+----------+---------------+--------
+-----------------+---------+-----------------+
| Score           |     6.47294 |      6.5 |      0.942971 |   1.85 |
9.1         |   12660 |             564 |
+-----------------+-------------+----------+---------------+--------
+-----------------+---------+-----------------+
| airtime (days) |   126.137   |     59   |    291.942    |  27    |
9466           |   12660 |             237 |
+-----------------+-------------+----------+---------------+--------
+-----------------+---------+-----------------+
```

Image Data

```
pip install requests Pillow
```

```
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (2.32.3)
Requirement already satisfied: Pillow in
/usr/local/lib/python3.10/dist-packages (11.0.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests) (3.4.0)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests) (2024.12.14)
```

NOTE: PRIOR TO THIS NEXT STEP, CREATE A FOLDER CALLED "MAL_pics".

This is where the images are to be stored.

```
import requests
from PIL import Image
from io import BytesIO

def download_and_save_image(url, image_path):
     try:
         # Attempt to fetch the image
         response = requests.get(url)
         response.raise_for_status()  # This will raise an HTTPError
if the status is 4xx or 5xx

         # Open the image and convert it to 'RGB' if necessary
         img = Image.open(BytesIO(response.content))

         # Convert 'P' mode images to 'RGB'
         if img.mode != 'RGB':
             img = img.convert('RGB')

         # Save the image to the specified path
         img.save(image_path)
         return True  # 'True' if the image is successfully saved
     except requests.exceptions.RequestException:

         return False  # Return False if the download fails


image_paths = []
for i, row in df.iterrows():
    url = row['image_url']  # Use the 'image_url' column
```

```
    image_path = f"/content/MAL_pics/image_{i}.jpg"

    # Attempt to download and save the image
    if download_and_save_image(url, image_path):
        image_paths.append(image_path)  # Append the path if
successful
    else:
        image_paths.append(None)  # Append None if download failed


# Add image paths to the dataframe
df['image_path'] = image_paths

import os

# Specify the path to your folder
folder_path = '/content/MAL_pics'

# Get a list of all files in the folder
image_files = [f for f in os.listdir(folder_path) if
f.endswith('.jpg') or f.endswith('.png')]  # Modify extensions if
needed

# Get the quantity of image files
num_images = len(image_files)


# Show the dataframe with image paths
df.head(15)
```


```
{"summary":"{\n  \"name\": \"df\",\n  \"rows\": 12641,\n  \"fields\":
[\n    {\n      \"column\": \"id\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 3689,\n        \"min\": 0,\n
\"max\": 12773,\n        \"num_unique_values\": 12641,\n
\"samples\": [\n          12249,\n          3699,\n          7023\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"Title\",\n      \"properties\":
{\n        \"dtype\": \"string\",\n        \"num_unique_values\":
12641,\n        \"samples\": [\n          \"Lupin the IIIrd: Mine
Fujiko no Uso\",\n          \"Transformers: Scramble City\",\n
\"Kaginado\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Rank\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 3691,\n        \"min\": 1,\n        \"max\": 12788,\n
\"num_unique_values\": 12641,\n        \"samples\": [\n
2590,\n          10024,\n          3508\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Type\",\n      \"properties\": {\n
\"dtype\": \"category\",\n        \"num_unique_values\": 5,\n
\"samples\": [\n          \"Movie\",\n          \"OVA\",\n
\"Special\"\n        ],\n        \"semantic_type\": \"\",\n
```

```
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Episodes\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 52,\n        \"min\": 1,\n
\"max\": 3057,\n        \"num_unique_values\": 186,\n
\"samples\": [\n          74,\n          237,\n          69\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Aired\",\n      \"properties\": {\n        \"dtype\": \"category\",\
n        \"num_unique_values\": 3570,\n        \"samples\": [\n
\"Dec 2012 - Dec 2012\",\n          \"Oct 1926 - Oct 1926\",\n
\"Aug 1987 - Aug 1987\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"Members\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 213712,\n        \"min\": 181,\n
\"max\": 3759013,\n        \"num_unique_values\": 9474,\n
\"samples\": [\n          832,\n          2760,\n          1612\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"page_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12641,\n        \"samples\": [\n
\"https://myanimelist.net/anime/39487/Lupin_the_IIIrd__Mine_Fujiko_no_
Uso\",\n
\"https://myanimelist.net/anime/6800/Transformers__Scramble_City\",\n
\"https://myanimelist.net/anime/48775/Kaginado\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"image_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12639,\n        \"samples\": [\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/7/75927.jpg?
s=aa3d807bfc0d2651fdbb17d15f6e872b\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/1014/123301.jpg?
s=0fd2f71421a0b6eef92a1b31b58f07e8\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/9/84490.jpg?
s=d5e28c679b6d6c31b2e285f6bf99788d\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Score\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\": 0.9427241989169596,\
n        \"min\": 1.85,\n        \"max\": 9.1,\n
\"num_unique_values\": 564,\n        \"samples\": [\n          8.65,\n
3.31,\n          7.77\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Start_year\",\n      \"properties\": {\n        \"dtype\":
\"object\",\n        \"num_unique_values\": 749,\n        \"samples\":
[\n          \"1983-05-01\",\n          \"1987-03-01\",\n
\"2015-10-01\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"End_year\",\n      \"properties\": {\n        \"dtype\":
\"object\",\n        \"num_unique_values\": 1273,\n
\"samples\": [\n          \"1931-10-31\",\n          \"1933-04-30\",\n
```

\"2005-12-31\"\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"airtime (days)\",\n   \"properties\": {\n   \"dtype\":
\"number\",\n   \"std\": 291,\n   \"min\": -243,\n
\"max\": 9466,\n   \"num_unique_values\": 243,\n
\"samples\": [\n   244,\n   335,\n   4108\n
],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n
}\n   },\n   {\n   \"column\": \"image_path\",\n
\"properties\": {\n   \"dtype\": \"string\",\n
\"num_unique_values\": 12641,\n   \"samples\": [\n
\"/content/MAL_pics/image_12141.jpg\",\n
\"/content/MAL_pics/image_3675.jpg\",\n
\"/content/MAL_pics/image_6964.jpg\"\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   }\n]\n}","type":"dataframe","variable_name":"df"}

```
df.shape
```

```
(12641, 14)
```

Of all the images we could not obtain (whether it be in JPG format or RGB format), we only had about 19 lost due to the URL not being found.

```
none_image_paths = df[df['image_path'].isna() | (df['image_path'] ==
None)]
none_image_paths[['Title', 'image_path']]

df = df.dropna(subset=['image_path'])
df.reset_index(drop=True, inplace=True)

df.head()
```

{"summary":"{\n \"name\": \"df\",\n \"rows\": 12641,\n \"fields\":
[\n   {\n   \"column\": \"id\",\n   \"properties\": {\n
\"dtype\": \"number\",\n   \"std\": 3689,\n   \"min\": 0,\n
\"max\": 12773,\n   \"num_unique_values\": 12641,\n
\"samples\": [\n   12249,\n   3699,\n   7023\n
],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n
}\n   },\n   {\n   \"column\": \"Title\",\n   \"properties\":
{\n   \"dtype\": \"string\",\n   \"num_unique_values\":
12641,\n   \"samples\": [\n   \"Lupin the IIIrd: Mine
Fujiko no Uso\",\n   \"Transformers: Scramble City\",\n
\"Kaginado\"\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"Rank\",\n   \"properties\": {\n   \"dtype\": \"number\",\n
\"std\": 3691,\n   \"min\": 1,\n   \"max\": 12788,\n
\"num_unique_values\": 12641,\n   \"samples\": [\n
2590,\n   10024,\n   3508\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"Type\",\n   \"properties\": {\n

```
\"dtype\": \"category\",\n        \"num_unique_values\": 5,\n
\"samples\": [\n          \"Movie\",\n          \"OVA\",\n
\"Special\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Episodes\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 52,\n        \"min\": 1,\n
\"max\": 3057,\n        \"num_unique_values\": 186,\n
\"samples\": [\n          74,\n          237,\n          69\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Aired\",\n      \"properties\": {\n        \"dtype\": \"category\",\
n        \"num_unique_values\": 3570,\n        \"samples\": [\n
\"Dec 2012 - Dec 2012\",\n          \"Oct 1926 - Oct 1926\",\n
\"Aug 1987 - Aug 1987\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"Members\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 213712,\n        \"min\": 181,\n
\"max\": 3759013,\n        \"num_unique_values\": 9474,\n
\"samples\": [\n          832,\n          2760,\n          1612\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"page_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12641,\n        \"samples\": [\n
\"https://myanimelist.net/anime/39487/Lupin_the_IIIrd__Mine_Fujiko_no_
Uso\",\n
\"https://myanimelist.net/anime/6800/Transformers__Scramble_City\",\n
\"https://myanimelist.net/anime/48775/Kaginado\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"image_url\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12639,\n        \"samples\": [\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/7/75927.jpg?
s=aa3d807bfc0d2651fdbb17d15f6e872b\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/1014/123301.jpg?
s=0fd2f71421a0b6eef92a1b31b58f07e8\",\n
\"https://cdn.myanimelist.net/r/100x140/images/anime/9/84490.jpg?
s=d5e28c679b6d6c31b2e285f6bf99788d\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Score\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\": 0.9427241989169596,\
n        \"min\": 1.85,\n        \"max\": 9.1,\n
\"num_unique_values\": 564,\n        \"samples\": [\n          8.65,\n
3.31,\n          7.77\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Start_year\",\n      \"properties\": {\n        \"dtype\":
\"date\",\n        \"min\": \"1917-05-01 00:00:00\",\n        \"max\":
\"2023-07-01 00:00:00\",\n        \"num_unique_values\": 749,\n
\"samples\": [\n          \"1983-05-01 00:00:00\",\n          \"1987-
03-01 00:00:00\",\n          \"2015-10-01 00:00:00\"\n        ],\n
```

```
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"End_year\",\n      \"properties\":
{\n        \"dtype\": \"date\",\n        \"min\": \"1917-05-31
00:00:00\",\n        \"max\": \"2023-07-31 00:00:00\",\n
\"num_unique_values\": 1273,\n        \"samples\": [\n
\"1931-10-31 00:00:00\",\n          \"1933-04-30 00:00:00\",\n
\"2005-12-31 00:00:00\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"airtime (days)\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 291,\n        \"min\": -243,\
n        \"max\": 9466,\n        \"num_unique_values\": 243,\n
\"samples\": [\n          244,\n          335,\n          4108\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"image_path\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 12641,\n        \"samples\": [\n
\"/content/MAL_pics/image_12141.jpg\",\n
\"/content/MAL_pics/image_3675.jpg\",\n
\"/content/MAL_pics/image_6964.jpg\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"df"}
```


```
print(df.shape)
```

```
(12641, 14)
```

For if Breaking the glass is necessary

If we crash or the runtime is disconnected, then you can save and download and re-utilize the data modified dataframe and image folder later.

Uncomment if these are necessary.

```
#df.to_csv('/content/MAL.csv', index=False)

from google.colab import files

# Download the saved CSV file
#files.download('/content/MAL.csv')

import shutil

# Compress the folder into a zip file
#shutil.make_archive('/content/MAL_pics', 'zip', '/content/MAL_pics')

#files.download('/content/MAL_pics.zip')
```

Accessing the zipped folder and making a new directory to hold the pictures. Uncomment if this is needed.

```
import zipfile
import os

# Path to the uploaded zip file
#zip_file_path = '/content/MAL_pics.zip'

# Put contents of the zip file into a new directory
#extract_dir = '/content/MAL_pics'

# Make sure the extraction directory exists
#os.makedirs(extract_dir, exist_ok=True)

# Open and extract the zip file
#with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
#    zip_ref.extractall(extract_dir)
```

Image Testing

```
from PIL import Image
import os
import matplotlib.pyplot as plt
import random

folder_path = "/content/MAL_pics"

# Get the list of image files in the folder
images = os.listdir(folder_path)


random.seed(41)
# Randomly select 4 images

random_images = random.sample(images, 4)

# Set up the figure to display the images
plt.figure(figsize=(10, 8))

# Loop through the randomly selected images
for i, image_name in enumerate(random_images):
    image_path = os.path.join(folder_path, image_name)
    img = Image.open(image_path)

    # Create a subplot for each image (2 rows, 2 columns)
    plt.subplot(2, 2, i + 1)  # (2 rows, 2 columns, position i + 1)
    plt.imshow(img)
    plt.axis('off')
    plt.title(f"Image {i + 1}: {image_name}")

plt.tight_layout()  # Adjust layout to avoid overlap
plt.show()
```

```
<Figure size 1000x800 with 0 Axes>
```

Image 1: image_3056.jpg

Image 2: image_6651.jpg

Image 3: image_6161.jpg

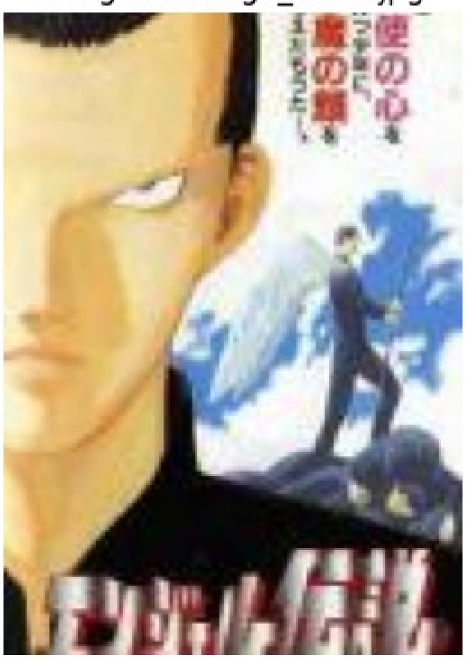


Image 4: image_7642.jpg


```python
for i, image_name in enumerate(random_images):
    image_path = os.path.join(folder_path, image_name)
    img = Image.open(image_path)
```

```
    # Get the size (width, height) of the image
    width, height = img.size
    print(f"Image {i + 1}: {image_name} - Size: {width}x{height}
pixels")
```

```
Image 1: image_3056.jpg - Size: 100x140 pixels
Image 2: image_6651.jpg - Size: 100x140 pixels
Image 3: image_6161.jpg - Size: 100x140 pixels
Image 4: image_7642.jpg - Size: 100x140 pixels
```

All of the images are in 100x140 pixel resolution.

Easy Delete

If the process of downloading the images fail, then you can easily delete the directory and start again. Simply uncomment the text and then make your changes.

```
#df = df.drop(columns=['image_path'])

# Verify if the column is removed
#print(df.head())

import os
import glob

# Specify the folder where the images are stored
#image_folder = '/content/MAL_pics/'

# Get a list of all image files in the folder (assuming they have .jpg
extension)
#image_files = glob.glob(os.path.join(image_folder, '*.jpg'))

# Delete all the image files
#for image_file in image_files:
#    os.remove(image_file)

#print(f"Deleted {len(image_files)} images.")
```

```
Deleted 990 images.
```

CNN

```
import os
import pandas as pd
from tensorflow.keras.preprocessing.image import load_img,
img_to_array
from tensorflow.keras.utils import to_categorical
```

```python
# Set the image width for loading to same dimensions as image
img_width, img_height = 140, 100


label_mapping = {
    'TV': 0,
    'Special': 1,
    'ONA': 2,
    'OVA': 3,
    'Movie': 4
}
# Make a numeric label based on the type
df['label'] = df['Type'].map(label_mapping)


# Ensure the image paths exist
df = df[df['image_path'].apply(lambda x: os.path.exists(x))]

# Loading + preprocessing image method
def load_and_preprocess_image(image_path):
    img = load_img(image_path, target_size=(img_width, img_height))
    img_array = img_to_array(img)  # Convert image to array
    img_array = img_array / 255.0  # Rescale the image to [0, 1]
    return img_array

# Convert all images and labels to lists / arrays
images = np.array([load_and_preprocess_image(path) for path in
df['image_path']])
labels = np.array(df['label'])

# One-hot encode the labels
labels = to_categorical(labels, num_classes=5)
```

CNN First Attempt: Simple 3 CNN Layer network

```python
images.shape
```

```
(12641, 140, 100, 3)
```

We utilize an Adam optimizer with categorical crossentropy loss function. Our activation for the output layer is softmax function - this will remain constant.

```python
from tensorflow.keras import layers, models

# Simple CNN architecture for 100x140 images
# CNN blocks with maxpooling in between + Dense layers at the end
model = models.Sequential([
    layers.InputLayer(input_shape=(img_height, img_width, 3)),
```

```
    # First convolutional block
    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2, 2)),

    # Second convolutional block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2, 2)),

    # Third convolutional block
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2, 2)),

    # Flatten data
    # 2 dense layers with a dropout layer in between
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(5, activation='softmax')  # 5 categories (TV,
Special, ONA, OVA, Movie)
])

# Compile model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Get architecture summary
model.summary()
```

```
Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ conv2d_3 (Conv2D)                     │ (None, 100, 140, 32)        │
896 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d_3 (MaxPooling2D)        │ (None, 50, 70, 32)          │
0 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ conv2d_4 (Conv2D)                     │ (None, 50, 70, 64)          │
18,496 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d_4 (MaxPooling2D)        │ (None, 25, 35, 64)          │
0 │
```

```
├─────────────────────────────────────┼────────────────────────────┼
─────────────────┤
│ conv2d_5 (Conv2D)                   │ (None, 25, 35, 128)        │
73,856 │
├─────────────────────────────────────┼────────────────────────────┼
─────────────────┤
│ max_pooling2d_5 (MaxPooling2D)      │ (None, 12, 17, 128)        │
0 │
├─────────────────────────────────────┼────────────────────────────┼
─────────────────┤
│ flatten_1 (Flatten)                 │ (None, 26112)              │
0 │
├─────────────────────────────────────┼────────────────────────────┼
─────────────────┤
│ dense_2 (Dense)                     │ (None, 512)                │
13,369,856 │
├─────────────────────────────────────┼────────────────────────────┼
─────────────────┤
│ dropout_1 (Dropout)                 │ (None, 512)                │
0 │
├─────────────────────────────────────┼────────────────────────────┼
─────────────────┤
│ dense_3 (Dense)                     │ (None, 5)                  │
2,565 │
└─────────────────────────────────────┴────────────────────────────┴
─────────────────┘

 Total params: 13,465,669 (51.37 MB)

 Trainable params: 13,465,669 (51.37 MB)

 Non-trainable params: 0 (0.00 B)

history = model.fit(
     X_train, y_train,
     validation_data=(X_val, y_val),
     epochs=20,
     batch_size=32,
     callbacks=callbacks
)

Epoch 1/20
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - accuracy: 0.3278 - loss:
1.6520
Epoch 1: val_loss improved from inf to 1.48337, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 22s 69ms/step - accuracy: 0.3279 - loss:
1.6517 - val_accuracy: 0.3846 - val_loss: 1.4834 - learning_rate:
0.0010
Epoch 2/20
```

```
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.3498 - loss:
1.5273
Epoch 2: val_loss improved from 1.48337 to 1.47236, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 6s 26ms/step - accuracy: 0.3500 - loss:
1.5272 - val_accuracy: 0.3969 - val_loss: 1.4724 - learning_rate:
0.0010
Epoch 3/20
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.3900 - loss:
1.4747
Epoch 3: val_loss improved from 1.47236 to 1.43922, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 7s 26ms/step - accuracy: 0.3900 - loss:
1.4746 - val_accuracy: 0.3984 - val_loss: 1.4392 - learning_rate:
0.0010
Epoch 4/20
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.4113 - loss:
1.4242
Epoch 4: val_loss improved from 1.43922 to 1.40029, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 6s 25ms/step - accuracy: 0.4112 - loss:
1.4243 - val_accuracy: 0.4192 - val_loss: 1.4003 - learning_rate:
0.0010
Epoch 5/20
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.4341 - loss:
1.3675
Epoch 5: val_loss did not improve from 1.40029
253/253 ━━━━━━━━━━━━━━━━━━━━ 9s 19ms/step - accuracy: 0.4342 - loss:
1.3675 - val_accuracy: 0.4048 - val_loss: 1.4399 - learning_rate:
0.0010
Epoch 6/20
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.4911 - loss:
1.2592
Epoch 6: val_loss did not improve from 1.40029
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.4911 - loss:
1.2593 - val_accuracy: 0.4142 - val_loss: 1.4457 - learning_rate:
0.0010
Epoch 7/20
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.5671 - loss:
1.0959
Epoch 7: val_loss did not improve from 1.40029

Epoch 7: ReduceLROnPlateau reducing learning rate to
0.0005000000237487257.
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 19ms/step - accuracy: 0.5671 - loss:
1.0959 - val_accuracy: 0.4058 - val_loss: 1.5445 - learning_rate:
0.0010
Epoch 8/20
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7004 - loss:
```

```
0.7979
Epoch 8: val_loss did not improve from 1.40029
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.7005 - loss:
0.7978 - val_accuracy: 0.3940 - val_loss: 1.7662 - learning_rate:
5.0000e-04
Epoch 9/20
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.7918 - loss:
0.5663
Epoch 9: val_loss did not improve from 1.40029
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.7917 - loss:
0.5664 - val_accuracy: 0.3806 - val_loss: 2.0851 - learning_rate:
5.0000e-04
```

Not great. Of the 20 epochs, it ran 9 and the best validation accuracy achieved for this simple CNN was 0.4192. Furthermore, there was instability / divergence towards the end as the validation loss stopped decreasing and instead started increasing. The model needs further refinement.

Second Attempt: Refined CNN with Class Weights w/o Data Augmentation

```
from sklearn.model_selection import train_test_split

# X = images. shape: (12641, 140, 100, 3))
# y = labels. shape: (12641,))

# Split into 80% train and 20% test
X_train, X_test, y_train, y_test = train_test_split(images, labels,
test_size=0.2, random_state=42)

# Further split the train set into 80% train and 20% validation
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.2, random_state=42)

integer_labels = np.argmax(y_train, axis=1)
# For the class weight computation
```

We will utilize class weights to attempt to address class imbalance in the training and test sets. These will be utilized for 2 of the next 3 models.

```
from sklearn.utils.class_weight import compute_class_weight
# Using class weights for the model fit on training data
# Theoretically, this should improve the predictions
class_weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(integer_labels),
    y=integer_labels
)
```

```
class_weights = dict(enumerate(class_weights))
print("Class Weights:", class_weights)
```

```
Class Weights: {0: 0.572065063649222, 1: 1.2521671826625387, 2:
1.3549413735343383, 3: 1.3282430213464695, 4: 1.039049454078356}
```

```
print(X_train.shape)
print(X_val.shape)
print(y_train.shape)
print(y_val.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(8089, 140, 100, 3)
(2023, 140, 100, 3)
(8089, 5)
(2023, 5)
(2529, 140, 100, 3)
(2529, 5)
```

We now have our training, test, and validation sets prepared. The validation set is smaller than the test set but it should serve to fine tune our model.

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import ReduceLROnPlateau

early_stopping = EarlyStopping(
    monitor='val_loss',      # Monitor validation loss
    patience=5,              # Stop after 5 epochs without improvement
    restore_best_weights=True  # Restore the model with the best
weights
)

checkpoint = ModelCheckpoint(
    'best_model.keras',         # Save the model to a file
    monitor='val_loss',      # Monitor validation loss
    save_best_only=True,     # Save the best model only
    mode='min',              # Save the model with the minimum loss
    verbose=1                # Print the message when the model is
saved
)


reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',      # Monitor validation loss
    factor=0.5,              # Reduce the learning rate by a factor of
0.5
    patience=3,              # Wait for 3 epochs before reducing
    min_lr=1e-6,             # Minimum learning rate
```

```
    verbose=1                # Print the message when the learning
rate is reduced
)

from tensorflow.keras import layers, models
import tensorflow as tf
model = models.Sequential([
    layers.InputLayer(input_shape=(img_height, img_width, 3)),

    # First convolutional block
    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),

    # Second convolutional block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),

    # Third convolutional block
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),

    # Global Average Pooling instead of Flatten
    # Will reduce parameters which should limit overfitting
    # Help focus on global context of feature maps rather than spatial
locations
    # Which may be helpful for generalization
    layers.GlobalAveragePooling2D(),

    # Dense layers, reduced dropout value
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(5, activation='softmax')  # Output layer
])


model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001)
,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

Model: "sequential_2"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                ┃
Param # ┃
```

```
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ conv2d_6 (Conv2D)                       │ (None, 100, 140, 32)           │
896 │
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ batch_normalization_6                   │ (None, 100, 140, 32)           │
128 │
│ (BatchNormalization)                    │                                │
│
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ max_pooling2d_6 (MaxPooling2D)          │ (None, 50, 70, 32)             │
0 │
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ conv2d_7 (Conv2D)                       │ (None, 50, 70, 64)             │
18,496 │
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ batch_normalization_7                   │ (None, 50, 70, 64)             │
256 │
│ (BatchNormalization)                    │                                │
│
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ max_pooling2d_7 (MaxPooling2D)          │ (None, 25, 35, 64)             │
0 │
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ conv2d_8 (Conv2D)                       │ (None, 25, 35, 128)            │
73,856 │
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ batch_normalization_8                   │ (None, 25, 35, 128)            │
512 │
│ (BatchNormalization)                    │                                │
│
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ max_pooling2d_8 (MaxPooling2D)          │ (None, 12, 17, 128)            │
0 │
├─────────────────────────────────────────┼────────────────────────────────┼
─────────────────┤
│ global_average_pooling2d_2              │ (None, 128)                    │
0 │
│ (GlobalAveragePooling2D)                │                                │
│
├─────────────────────────────────────────┼────────────────────────────────┼
```

```
──────────────────────┤
│ dense_4 (Dense)                            │ (None, 512)                            │
66,048 │
├────────────────────────────────────────────┼────────────────────────────────────────┼
──────────────────────┤
│ dropout_2 (Dropout)                        │ (None, 512)                            │
0 │
├────────────────────────────────────────────┼────────────────────────────────────────┼
──────────────────────┤
│ dense_5 (Dense)                            │ (None, 5)                              │
2,565 │
└────────────────────────────────────────────┴────────────────────────────────────────┴
──────────────────────┘

 Total params: 162,757 (635.77 KB)

 Trainable params: 162,309 (634.02 KB)

 Non-trainable params: 448 (1.75 KB)
```

```
history = model.fit(X_train, y_train, epochs=50,
validation_data=(X_val, y_val), class_weight=class_weights,
callbacks=callbacks)
```

```
Epoch 1/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step - accuracy: 0.2603 - loss:
1.6239
Epoch 1: val_loss did not improve from 1.52153
253/253 ━━━━━━━━━━━━━━━━━━━━ 27s 87ms/step - accuracy: 0.2604 - loss:
1.6238 - val_accuracy: 0.2007 - val_loss: 2.0504 - learning_rate:
1.0000e-04
Epoch 2/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.3164 - loss:
1.5334
Epoch 2: val_loss did not improve from 1.52153
253/253 ━━━━━━━━━━━━━━━━━━━━ 22s 23ms/step - accuracy: 0.3164 - loss:
1.5334 - val_accuracy: 0.2343 - val_loss: 1.6620 - learning_rate:
1.0000e-04
Epoch 3/50
250/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.3274 - loss:
1.5199
Epoch 3: val_loss did not improve from 1.52153
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 23ms/step - accuracy: 0.3272 - loss:
1.5199 - val_accuracy: 0.3203 - val_loss: 1.5218 - learning_rate:
1.0000e-04
Epoch 4/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.3427 - loss:
1.5092
Epoch 4: val_loss improved from 1.52153 to 1.48167, saving model to
best_model.keras
```

```
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 22ms/step - accuracy: 0.3427 - loss:
1.5092 - val_accuracy: 0.3104 - val_loss: 1.4817 - learning_rate:
1.0000e-04
Epoch 5/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.3439 - loss:
1.4945
Epoch 5: val_loss did not improve from 1.48167
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 22ms/step - accuracy: 0.3440 - loss:
1.4945 - val_accuracy: 0.3465 - val_loss: 1.4863 - learning_rate:
1.0000e-04
Epoch 6/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.3616 - loss:
1.4732
Epoch 6: val_loss improved from 1.48167 to 1.46806, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 20ms/step - accuracy: 0.3615 - loss:
1.4733 - val_accuracy: 0.3742 - val_loss: 1.4681 - learning_rate:
1.0000e-04
Epoch 7/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.3730 - loss:
1.4609
Epoch 7: val_loss improved from 1.46806 to 1.45076, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.3730 - loss:
1.4609 - val_accuracy: 0.3910 - val_loss: 1.4508 - learning_rate:
1.0000e-04
Epoch 8/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.3727 - loss:
1.4564
Epoch 8: val_loss did not improve from 1.45076
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 20ms/step - accuracy: 0.3727 - loss:
1.4565 - val_accuracy: 0.3618 - val_loss: 1.4556 - learning_rate:
1.0000e-04
Epoch 9/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.3647 - loss:
1.4516
Epoch 9: val_loss did not improve from 1.45076
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.3647 - loss:
1.4516 - val_accuracy: 0.3836 - val_loss: 1.4592 - learning_rate:
1.0000e-04
Epoch 10/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.3867 - loss:
1.4455
Epoch 10: val_loss did not improve from 1.45076

Epoch 10: ReduceLROnPlateau reducing learning rate to
4.999999873689376e-05.
253/253 ━━━━━━━━━━━━━━━━━━━━ 6s 25ms/step - accuracy: 0.3867 - loss:
1.4455 - val_accuracy: 0.3515 - val_loss: 1.4607 - learning_rate:
```

```
1.0000e-04
Epoch 11/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.3973 - loss:
1.4262
Epoch 11: val_loss improved from 1.45076 to 1.44050, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 9s 22ms/step - accuracy: 0.3973 - loss:
1.4262 - val_accuracy: 0.3361 - val_loss: 1.4405 - learning_rate:
5.0000e-05
Epoch 12/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.3911 - loss:
1.4338
Epoch 12: val_loss did not improve from 1.44050
253/253 ━━━━━━━━━━━━━━━━━━━━ 6s 22ms/step - accuracy: 0.3911 - loss:
1.4337 - val_accuracy: 0.3880 - val_loss: 1.4413 - learning_rate:
5.0000e-05
Epoch 13/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4106 - loss:
1.4006
Epoch 13: val_loss improved from 1.44050 to 1.42603, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 22ms/step - accuracy: 0.4106 - loss:
1.4007 - val_accuracy: 0.3796 - val_loss: 1.4260 - learning_rate:
5.0000e-05
Epoch 14/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.3950 - loss:
1.4071
Epoch 14: val_loss did not improve from 1.42603
253/253 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.3950 - loss:
1.4072 - val_accuracy: 0.3673 - val_loss: 1.4577 - learning_rate:
5.0000e-05
Epoch 15/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.4012 - loss:
1.3954
Epoch 15: val_loss improved from 1.42603 to 1.42583, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 23ms/step - accuracy: 0.4012 - loss:
1.3955 - val_accuracy: 0.3772 - val_loss: 1.4258 - learning_rate:
5.0000e-05
Epoch 16/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4026 - loss:
1.4141
Epoch 16: val_loss did not improve from 1.42583
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4026 - loss:
1.4141 - val_accuracy: 0.3164 - val_loss: 1.4694 - learning_rate:
5.0000e-05
Epoch 17/50
250/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4097 - loss:
1.4010
```

```
Epoch 17: val_loss did not improve from 1.42583
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 20ms/step - accuracy: 0.4097 - loss:
1.4010 - val_accuracy: 0.3406 - val_loss: 1.4455 - learning_rate:
5.0000e-05
Epoch 18/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4036 - loss:
1.3965
Epoch 18: val_loss improved from 1.42583 to 1.41249, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.4036 - loss:
1.3965 - val_accuracy: 0.4024 - val_loss: 1.4125 - learning_rate:
5.0000e-05
Epoch 19/50
250/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.4188 - loss:
1.3762
Epoch 19: val_loss did not improve from 1.41249
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.4187 - loss:
1.3764 - val_accuracy: 0.3643 - val_loss: 1.4412 - learning_rate:
5.0000e-05
Epoch 20/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.3969 - loss:
1.3999
Epoch 20: val_loss did not improve from 1.41249
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 22ms/step - accuracy: 0.3970 - loss:
1.3998 - val_accuracy: 0.3737 - val_loss: 1.4496 - learning_rate:
5.0000e-05
Epoch 21/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4165 - loss:
1.3829
Epoch 21: val_loss improved from 1.41249 to 1.40528, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.4165 - loss:
1.3829 - val_accuracy: 0.4034 - val_loss: 1.4053 - learning_rate:
5.0000e-05
Epoch 22/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4330 - loss:
1.3767
Epoch 22: val_loss did not improve from 1.40528
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.4330 - loss:
1.3768 - val_accuracy: 0.3816 - val_loss: 1.4211 - learning_rate:
5.0000e-05
Epoch 23/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4103 - loss:
1.3857
Epoch 23: val_loss did not improve from 1.40528
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 20ms/step - accuracy: 0.4104 - loss:
1.3856 - val_accuracy: 0.3870 - val_loss: 1.4273 - learning_rate:
5.0000e-05
Epoch 24/50
```

```
250/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4275 - loss:
1.3701
Epoch 24: val_loss improved from 1.40528 to 1.40398, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4275 - loss:
1.3702 - val_accuracy: 0.3959 - val_loss: 1.4040 - learning_rate:
5.0000e-05
Epoch 25/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4277 - loss:
1.3650
Epoch 25: val_loss did not improve from 1.40398
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.4277 - loss:
1.3650 - val_accuracy: 0.3974 - val_loss: 1.4280 - learning_rate:
5.0000e-05
Epoch 26/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4180 - loss:
1.3612
Epoch 26: val_loss did not improve from 1.40398
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4181 - loss:
1.3613 - val_accuracy: 0.4068 - val_loss: 1.4115 - learning_rate:
5.0000e-05
Epoch 27/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4282 - loss:
1.3496
Epoch 27: val_loss did not improve from 1.40398

Epoch 27: ReduceLROnPlateau reducing learning rate to
2.499999936844688e-05.
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4282 - loss:
1.3497 - val_accuracy: 0.3950 - val_loss: 1.4212 - learning_rate:
5.0000e-05
Epoch 28/50
250/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.4462 - loss:
1.3412
Epoch 28: val_loss improved from 1.40398 to 1.39802, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4461 - loss:
1.3413 - val_accuracy: 0.3925 - val_loss: 1.3980 - learning_rate:
2.5000e-05
Epoch 29/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4427 - loss:
1.3419
Epoch 29: val_loss did not improve from 1.39802
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.4427 - loss:
1.3419 - val_accuracy: 0.3915 - val_loss: 1.4162 - learning_rate:
2.5000e-05
Epoch 30/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4406 - loss:
1.3420
```

```
Epoch 30: val_loss did not improve from 1.39802
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4407 - loss:
1.3420 - val_accuracy: 0.3747 - val_loss: 1.4317 - learning_rate:
2.5000e-05
Epoch 31/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.4454 - loss:
1.3176
Epoch 31: val_loss did not improve from 1.39802

Epoch 31: ReduceLROnPlateau reducing learning rate to
1.249999968422344e-05.
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.4454 - loss:
1.3177 - val_accuracy: 0.3727 - val_loss: 1.4207 - learning_rate:
2.5000e-05
Epoch 32/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4401 - loss:
1.3284
Epoch 32: val_loss did not improve from 1.39802
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4401 - loss:
1.3284 - val_accuracy: 0.3786 - val_loss: 1.4006 - learning_rate:
1.2500e-05
Epoch 33/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.4592 - loss:
1.3177
Epoch 33: val_loss improved from 1.39802 to 1.39088, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 6s 22ms/step - accuracy: 0.4591 - loss:
1.3178 - val_accuracy: 0.4029 - val_loss: 1.3909 - learning_rate:
1.2500e-05
Epoch 34/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4578 - loss:
1.3270
Epoch 34: val_loss improved from 1.39088 to 1.38493, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4578 - loss:
1.3270 - val_accuracy: 0.4014 - val_loss: 1.3849 - learning_rate:
1.2500e-05
Epoch 35/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.4484 - loss:
1.3330
Epoch 35: val_loss did not improve from 1.38493
253/253 ━━━━━━━━━━━━━━━━━━━━ 6s 22ms/step - accuracy: 0.4484 - loss:
1.3329 - val_accuracy: 0.4024 - val_loss: 1.3998 - learning_rate:
1.2500e-05
Epoch 36/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.4507 - loss:
1.3166
Epoch 36: val_loss did not improve from 1.38493
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4507 - loss:
```

```
1.3168 - val_accuracy: 0.3945 - val_loss: 1.3990 - learning_rate:
1.2500e-05
Epoch 37/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.4470 - loss:
1.3288
Epoch 37: val_loss did not improve from 1.38493

Epoch 37: ReduceLROnPlateau reducing learning rate to
6.24999984211172e-06.
253/253 ━━━━━━━━━━━━━━━━━━━━ 6s 22ms/step - accuracy: 0.4470 - loss:
1.3287 - val_accuracy: 0.4004 - val_loss: 1.3933 - learning_rate:
1.2500e-05
Epoch 38/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4545 - loss:
1.3090
Epoch 38: val_loss did not improve from 1.38493
253/253 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.4545 - loss:
1.3091 - val_accuracy: 0.4053 - val_loss: 1.3944 - learning_rate:
6.2500e-06
Epoch 39/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4505 - loss:
1.3119
Epoch 39: val_loss did not improve from 1.38493
253/253 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.4506 - loss:
1.3119 - val_accuracy: 0.4024 - val_loss: 1.3889 - learning_rate:
6.2500e-06
```

```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
80/80 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - accuracy: 0.4018 - loss:
1.3987
Test Loss: 1.4005510807037354
Test Accuracy: 0.41004350781440735
```

```
import matplotlib.pyplot as plt


# Plotting training and validation loss
plt.figure(figsize=(12, 6))

# Plot loss
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
# Plot accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

At best we are achieving a 41% accuracy at predicting anime media type using the cover images alone. We trained for more epochs, but it was pretty choppy. May indicate overfitting. The dataset may not be suitably large for reducing over or underfitting. To address that, let's utilize data augmentation for the next models.

Third Attempt: Refined CNN with Class Weights w/ Data Augmentation

Need to continue to remake the checkpoints so that the loss it compares the model to for its patience resets. This way our training isn't cut short.

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import ReduceLROnPlateau

early_stopping = EarlyStopping(
```

```
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

checkpoint = ModelCheckpoint(
    'best_model.keras',
    monitor='val_loss',
    save_best_only=True,
    mode='min',
    verbose=1
)


reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=3,
    min_lr=1e-6,
    verbose=1
)

callbacks = [early_stopping, checkpoint, reduce_lr]

import tensorflow as tf
print("Num GPUs Available: ",
len(tf.config.experimental.list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

We will be utilizing a Google Colab GPU for testing.

Here we add the data augmentation for the images.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# Use data augmentation
# Width shift, height shift, zoom, shear, fill, flip and rotation
# All to improve generalization
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest'
)

datagen.fit(X_train)
```

```
from tensorflow.keras import layers, models
import tensorflow as tf
model1 = models.Sequential([
    layers.InputLayer(input_shape=(img_height, img_width, 3)),

    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),

    layers.GlobalAveragePooling2D(),

    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(5, activation='softmax')  # Output layer
])


model1.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001
),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model1.summary()
```

```
Model: "sequential_11"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                 ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ conv2d_33 (Conv2D)                    │ (None, 100, 140, 32)         │
896 │
├───────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ batch_normalization_33                │ (None, 100, 140, 32)         │
128 │
│ (BatchNormalization)                  │                              │
│
├───────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
```

```
│ max_pooling2d_33 (MaxPooling2D)       │ (None, 50, 70, 32)          │
0 │
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ conv2d_34 (Conv2D)                    │ (None, 50, 70, 64)          │
18,496 │
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ batch_normalization_34                │ (None, 50, 70, 64)          │
256 │
│ (BatchNormalization)                  │                             │
│
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ max_pooling2d_34 (MaxPooling2D)       │ (None, 25, 35, 64)          │
0 │
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ conv2d_35 (Conv2D)                    │ (None, 25, 35, 128)         │
73,856 │
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ batch_normalization_35                │ (None, 25, 35, 128)         │
512 │
│ (BatchNormalization)                  │                             │
│
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ max_pooling2d_35 (MaxPooling2D)       │ (None, 12, 17, 128)         │
0 │
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ global_average_pooling2d_11           │ (None, 128)                 │
0 │
│ (GlobalAveragePooling2D)              │                             │
│
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dense_22 (Dense)                      │ (None, 512)                 │
66,048 │
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dropout_11 (Dropout)                  │ (None, 512)                 │
0 │
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dense_23 (Dense)                      │ (None, 5)                   │
2,565 │
```

```
└──────────────────────────────────────┴─────────────────────────────┘
─────────────────┘

 Total params: 162,757 (635.77 KB)

 Trainable params: 162,309 (634.02 KB)

 Non-trainable params: 448 (1.75 KB)

# Train the model directly using augmented data with class weighting
history = model1.fit(
    datagen.flow(X_train, y_train, batch_size=32),  # Generates
augmented batches
    epochs=50,
    validation_data=(X_val, y_val),
    class_weight=class_weights,
    callbacks=callbacks
)

Epoch 1/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 190ms/step - accuracy: 0.2514 - loss:
1.6226
Epoch 1: val_loss improved from inf to 1.90492, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 57s 198ms/step - accuracy: 0.2517 - loss:
1.6223 - val_accuracy: 0.1997 - val_loss: 1.9049 - learning_rate:
1.0000e-04
Epoch 2/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 117ms/step - accuracy: 0.3047 - loss:
1.5546
Epoch 2: val_loss improved from 1.90492 to 1.57862, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 59s 120ms/step - accuracy: 0.3047 - loss:
1.5546 - val_accuracy: 0.2625 - val_loss: 1.5786 - learning_rate:
1.0000e-04
Epoch 3/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 118ms/step - accuracy: 0.3191 - loss:
1.5306
Epoch 3: val_loss improved from 1.57862 to 1.44966, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 40s 120ms/step - accuracy: 0.3190 - loss:
1.5306 - val_accuracy: 0.3900 - val_loss: 1.4497 - learning_rate:
1.0000e-04
Epoch 4/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 124ms/step - accuracy: 0.3216 - loss:
1.5279
Epoch 4: val_loss improved from 1.44966 to 1.41227, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 43s 127ms/step - accuracy: 0.3216 - loss:
1.5279 - val_accuracy: 0.4217 - val_loss: 1.4123 - learning_rate:
```

```
1.0000e-04
Epoch 5/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 115ms/step - accuracy: 0.3336 - loss:
1.5346
Epoch 5: val_loss did not improve from 1.41227
253/253 ━━━━━━━━━━━━━━━━━━━━ 38s 116ms/step - accuracy: 0.3336 - loss:
1.5344 - val_accuracy: 0.3905 - val_loss: 1.4278 - learning_rate:
1.0000e-04
Epoch 6/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step - accuracy: 0.3391 - loss:
1.5046
Epoch 6: val_loss did not improve from 1.41227
253/253 ━━━━━━━━━━━━━━━━━━━━ 41s 117ms/step - accuracy: 0.3392 - loss:
1.5045 - val_accuracy: 0.3831 - val_loss: 1.4282 - learning_rate:
1.0000e-04
Epoch 7/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 115ms/step - accuracy: 0.3407 - loss:
1.4888
Epoch 7: val_loss did not improve from 1.41227

Epoch 7: ReduceLROnPlateau reducing learning rate to
4.999999873689376e-05.
253/253 ━━━━━━━━━━━━━━━━━━━━ 30s 117ms/step - accuracy: 0.3407 - loss:
1.4889 - val_accuracy: 0.3999 - val_loss: 1.4245 - learning_rate:
1.0000e-04
Epoch 8/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step - accuracy: 0.3751 - loss:
1.4782
Epoch 8: val_loss did not improve from 1.41227
253/253 ━━━━━━━━━━━━━━━━━━━━ 30s 117ms/step - accuracy: 0.3750 - loss:
1.4782 - val_accuracy: 0.4177 - val_loss: 1.4190 - learning_rate:
5.0000e-05
Epoch 9/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step - accuracy: 0.3617 - loss:
1.4675
Epoch 9: val_loss improved from 1.41227 to 1.40408, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 31s 119ms/step - accuracy: 0.3617 - loss:
1.4675 - val_accuracy: 0.4108 - val_loss: 1.4041 - learning_rate:
5.0000e-05
Epoch 10/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step - accuracy: 0.3707 - loss:
1.4517
Epoch 10: val_loss did not improve from 1.40408
253/253 ━━━━━━━━━━━━━━━━━━━━ 41s 119ms/step - accuracy: 0.3706 - loss:
1.4518 - val_accuracy: 0.4014 - val_loss: 1.4197 - learning_rate:
5.0000e-05
Epoch 11/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 240ms/step - accuracy: 0.3744 - loss:
```

```
1.4578
Epoch 11: val_loss did not improve from 1.40408
253/253 ━━━━━━━━━━━━━━━━━━━━ 72s 242ms/step - accuracy: 0.3744 - loss:
1.4578 - val_accuracy: 0.3984 - val_loss: 1.4510 - learning_rate:
5.0000e-05
Epoch 12/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 119ms/step - accuracy: 0.3757 - loss:
1.4584
Epoch 12: val_loss improved from 1.40408 to 1.39783, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 52s 121ms/step - accuracy: 0.3757 - loss:
1.4584 - val_accuracy: 0.4217 - val_loss: 1.3978 - learning_rate:
5.0000e-05
Epoch 13/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 127ms/step - accuracy: 0.3684 - loss:
1.4464
Epoch 13: val_loss did not improve from 1.39783
253/253 ━━━━━━━━━━━━━━━━━━━━ 33s 129ms/step - accuracy: 0.3684 - loss:
1.4464 - val_accuracy: 0.3811 - val_loss: 1.4391 - learning_rate:
5.0000e-05
Epoch 14/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 117ms/step - accuracy: 0.3828 - loss:
1.4441
Epoch 14: val_loss did not improve from 1.39783
253/253 ━━━━━━━━━━━━━━━━━━━━ 38s 119ms/step - accuracy: 0.3827 - loss:
1.4442 - val_accuracy: 0.3524 - val_loss: 1.4576 - learning_rate:
5.0000e-05
Epoch 15/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 117ms/step - accuracy: 0.3843 - loss:
1.4370
Epoch 15: val_loss did not improve from 1.39783

Epoch 15: ReduceLROnPlateau reducing learning rate to
2.499999936844688e-05.
253/253 ━━━━━━━━━━━━━━━━━━━━ 41s 119ms/step - accuracy: 0.3842 - loss:
1.4371 - val_accuracy: 0.3702 - val_loss: 1.4362 - learning_rate:
5.0000e-05
Epoch 16/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 115ms/step - accuracy: 0.3727 - loss:
1.4488
Epoch 16: val_loss did not improve from 1.39783
253/253 ━━━━━━━━━━━━━━━━━━━━ 40s 117ms/step - accuracy: 0.3727 - loss:
1.4487 - val_accuracy: 0.3856 - val_loss: 1.4480 - learning_rate:
2.5000e-05
Epoch 17/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 115ms/step - accuracy: 0.3807 - loss:
1.4407
Epoch 17: val_loss did not improve from 1.39783
253/253 ━━━━━━━━━━━━━━━━━━━━ 41s 116ms/step - accuracy: 0.3807 - loss:
```

```
1.4407 - val_accuracy: 0.3574 - val_loss: 1.4547 - learning_rate:
2.5000e-05
```

This time it trained for 17 out of the 50 epochs.

```python
import matplotlib.pyplot as plt


# Plotting training and validation loss
plt.figure(figsize=(12, 6))

# Plot loss
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plot accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```


```
test_loss1, test_accuracy1 = model1.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss1}")
print(f"Test Accuracy: {test_accuracy1}")

80/80 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.4256 - loss:
1.4093
Test Loss: 1.411325216293335
Test Accuracy: 0.42467379570007324
```

Test accuracy of 42.47% and test loss of 1.4113. It is still relatively choppy and the model trained for fewer epochs. But its doing surprisingly better than the training accuracy and loss.

May be a sign of underfitting? Could be having flukes or learning more from random noise.

```
predictions1 = model1.predict(X_test)

80/80 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step

# predictions are one-hot encoded. Change back to the numeric class
labels.
predicted_classes1 = np.argmax(predictions1, axis=1)

# y_test is one-hot encoded. Change it back to the numeric class
labels.
true_classes1 = np.argmax(y_test, axis=1)

pd.DataFrame(predicted_classes1).value_counts()

0
0    1374
4     569
2     268
```

```
3     254
1      64
Name: count, dtype: int64
```

We see that it predicted 'Special' the least out of all the classes.

```
df['Type'].value_counts()
```

```
Type
TV         4440
Movie      2477
Special    2005
OVA        1874
ONA        1845
Name: count, dtype: int64
```

```
pd.DataFrame(np.argmax(y_train, axis=1)).value_counts()
```

```
0
0    2828
4    1557
1    1292
3    1218
2    1194
Name: count, dtype: int64
```

```
label_mapping
```

```
{'TV': 0, 'Special': 1, 'ONA': 2, 'OVA': 3, 'Movie': 4}
```

Surprisingly, the model does not predict Specials very often despite them having the 3rd highest portion of the whole data set and the 3rd highest portion of the whole training set, too.

```
print(classification_report(true_classes1, predicted_classes1))
```

```
              precision    recall  f1-score   support

           0       0.48      0.73      0.58       898
           1       0.28      0.05      0.08       390
           2       0.40      0.29      0.34       370
           3       0.26      0.19      0.22       356
           4       0.40      0.44      0.42       515

    accuracy                           0.42      2529
   macro avg       0.36      0.34      0.33      2529
weighted avg       0.39      0.42      0.38      2529
```

Our F1 scores and accuracy are relatively low for every class but especially low for Specials. Utilizing class weights is important, however, if we want to ensure that predictions do come out for every class as the CNN is not naturally capable of discerning OVAs from all the categories (perhaps especially TV shows) on its own, even with data augmentation. This is reflected by an example without weighting.

Addendum to Third Attempt: Augmentation Without class weights.

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import ReduceLROnPlateau

early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

checkpoint = ModelCheckpoint(
    'best_model.keras',
    monitor='val_loss',
    save_best_only=True,
    mode='min',
    verbose=1
)


reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=3,
    min_lr=1e-6,
    verbose=1
)

callbacks = [early_stopping, checkpoint, reduce_lr]

from tensorflow.keras import layers, models
import tensorflow as tf
model = models.Sequential([
    layers.InputLayer(input_shape=(img_height, img_width, 3)),

    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),
```

```
    layers.GlobalAveragePooling2D(),

    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(5, activation='softmax')  # Output layer
])


model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001)
,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

# Train the model directly using only augmented data, no weights
history = model.fit(
    datagen.flow(X_train, y_train, batch_size=32),
    epochs=50,
    validation_data=(X_val, y_val),
    callbacks=callbacks
)
```

```
Epoch 1/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step - accuracy: 0.3639 - loss:
1.5462
Epoch 1: val_loss improved from inf to 1.92142, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 48s 167ms/step - accuracy: 0.3639 - loss:
1.5461 - val_accuracy: 0.1997 - val_loss: 1.9214 - learning_rate:
1.0000e-04
Epoch 2/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 119ms/step - accuracy: 0.3789 - loss:
1.4913
Epoch 2: val_loss improved from 1.92142 to 1.50728, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 31s 121ms/step - accuracy: 0.3789 - loss:
1.4913 - val_accuracy: 0.3327 - val_loss: 1.5073 - learning_rate:
1.0000e-04
Epoch 3/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 115ms/step - accuracy: 0.3861 - loss:
1.4761
Epoch 3: val_loss improved from 1.50728 to 1.43664, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 40s 118ms/step - accuracy: 0.3861 - loss:
1.4761 - val_accuracy: 0.3989 - val_loss: 1.4366 - learning_rate:
1.0000e-04
Epoch 4/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 119ms/step - accuracy: 0.3902 - loss:
```

```
1.4526
Epoch 4: val_loss did not improve from 1.43664
253/253 ━━━━━━━━━━━━━━━━━━━━ 32s 121ms/step - accuracy: 0.3903 - loss:
1.4526 - val_accuracy: 0.4053 - val_loss: 1.4546 - learning_rate:
1.0000e-04
Epoch 5/50
251/253 ━━━━━━━━━━━━━━━━━━━━ 0s 121ms/step - accuracy: 0.4061 - loss:
1.4339
Epoch 5: val_loss improved from 1.43664 to 1.41358, saving model to
best_model.keras
253/253 ━━━━━━━━━━━━━━━━━━━━ 41s 123ms/step - accuracy: 0.4061 - loss:
1.4340 - val_accuracy: 0.4147 - val_loss: 1.4136 - learning_rate:
1.0000e-04
Epoch 6/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 117ms/step - accuracy: 0.4204 - loss:
1.4268
Epoch 6: val_loss did not improve from 1.41358
253/253 ━━━━━━━━━━━━━━━━━━━━ 40s 118ms/step - accuracy: 0.4204 - loss:
1.4268 - val_accuracy: 0.4197 - val_loss: 1.4249 - learning_rate:
1.0000e-04
Epoch 7/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step - accuracy: 0.4194 - loss:
1.4182
Epoch 7: val_loss did not improve from 1.41358
253/253 ━━━━━━━━━━━━━━━━━━━━ 41s 118ms/step - accuracy: 0.4193 - loss:
1.4183 - val_accuracy: 0.4142 - val_loss: 1.4418 - learning_rate:
1.0000e-04
Epoch 8/50
253/253 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step - accuracy: 0.4178 - loss:
1.4130
Epoch 8: val_loss did not improve from 1.41358

Epoch 8: ReduceLROnPlateau reducing learning rate to
4.999999873689376e-05.
253/253 ━━━━━━━━━━━━━━━━━━━━ 31s 118ms/step - accuracy: 0.4178 - loss:
1.4130 - val_accuracy: 0.4098 - val_loss: 1.4671 - learning_rate:
1.0000e-04
Epoch 9/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step - accuracy: 0.4239 - loss:
1.4052
Epoch 9: val_loss did not improve from 1.41358
253/253 ━━━━━━━━━━━━━━━━━━━━ 31s 118ms/step - accuracy: 0.4239 - loss:
1.4052 - val_accuracy: 0.4128 - val_loss: 1.4621 - learning_rate:
5.0000e-05
Epoch 10/50
252/253 ━━━━━━━━━━━━━━━━━━━━ 0s 118ms/step - accuracy: 0.4284 - loss:
1.3866
Epoch 10: val_loss did not improve from 1.41358
253/253 ━━━━━━━━━━━━━━━━━━━━ 41s 119ms/step - accuracy: 0.4283 - loss:
```

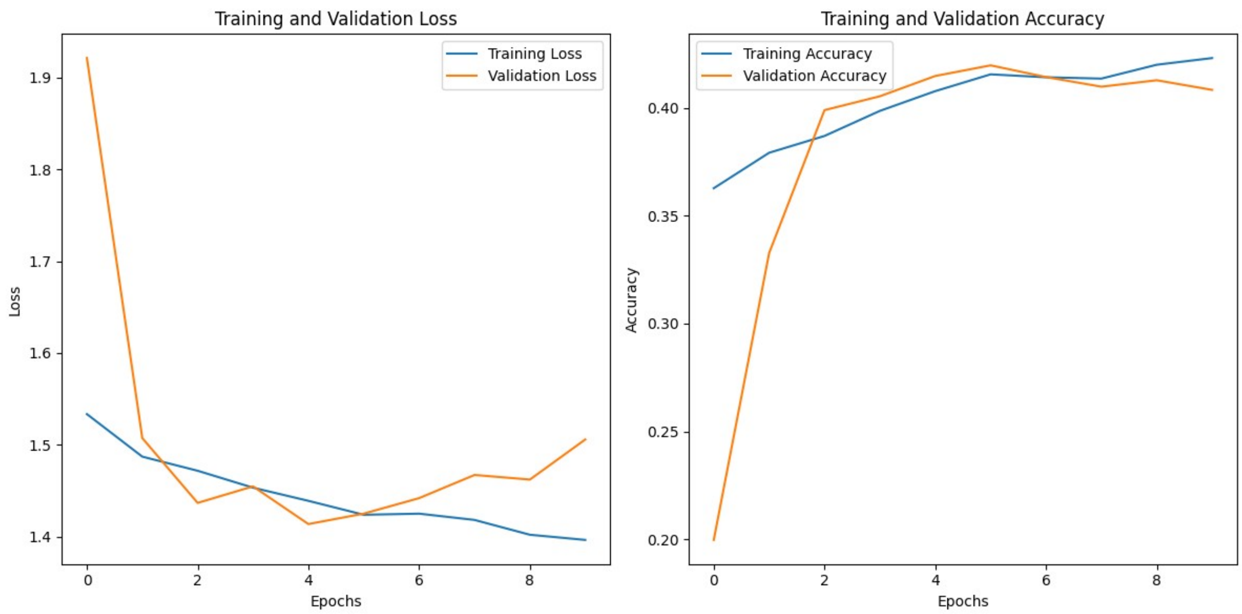
```
1.3867 - val_accuracy: 0.4083 - val_loss: 1.5057 - learning_rate:
5.0000e-05
```

It trained for 10 epochs out of 50 this time.

```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
80/80 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.4290 - loss:
1.4068
Test Loss: 1.4180619716644287
Test Accuracy: 0.4215104877948761
```

```
import matplotlib.pyplot as plt

# Assuming 'history' is the object returned by model.fit()
# You can access the history of training metrics through the 'history'
attribute

# Plotting training and validation loss
plt.figure(figsize=(12, 6))

# Plot loss
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plot accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```



We obtained comparable accuracy to our weighted model and the training appears slightly less choppy or unstable, though it has trained for fewer epochs.

Test accuracy of 42.15%, test loss of 1.4181.

```
predictions = keras_mod.predict(X_test)
```

```
80/80 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step
```

```
predicted_classes = np.argmax(predictions, axis=1)
```

```
# y_test is one-hot encoded. Convert to class labels (numbers)
true_classes = np.argmax(y_test, axis=1)
```

```
true_classes
```

```
array([4, 3, 0, ..., 0, 2, 0])
```

Refer back to the label mapping as we read this.

```
label_mapping
```

```
{'TV': 0, 'Special': 1, 'ONA': 2, 'OVA': 3, 'Movie': 4}
```

```
from sklearn.metrics import classification_report
```

```
print(classification_report(true_classes, predicted_classes))
```

```
              precision    recall  f1-score   support

           0       0.42      0.95      0.59       898
           1       0.30      0.05      0.09       390
```

```
           2       0.43      0.21      0.28       370
           3       0.00      0.00      0.00       356
           4       0.44      0.22      0.29       515

    accuracy                           0.42      2529
   macro avg       0.32      0.29      0.25      2529
weighted avg       0.35      0.42      0.32      2529
```

```
pd.DataFrame(predicted_classes).value_counts()
```

```
0
0    2025
4     252
2     182
1      70
Name: count, dtype: int64
```

However, this "comparable" success is somewhat hollow as there was a few major issues with the model's predictions.

It did not once predict OVAs. Even with augmented data, it was not capable of doing so. The majority of the predictions seem to be tied up with making predictions for TV shows as it makes up a large portion of the test data.

As a side note, we also attempted utilizing a pre-trained model (ImageNet).

While we would like to use a pre-trained model to improve our predictive capacity, this has computational issues with memory and has crashed the session almost immediately.

Future work for classifying the images would benefit from utilizing a OCR to not only identify any text on the image but use that identified text and read it or translate it to, theoretically, cheat through any predictions (as some may identify they are a "Special" or "Movie" within their names).

Pytorch Model for Transformers on Title

Let's begin our work with BERT.

```
pip install transformers torch
```

```
Requirement already satisfied: transformers in
/usr/local/lib/python3.10/dist-packages (4.47.0)
Requirement already satisfied: torch in
/usr/local/lib/python3.10/dist-packages (2.5.1+cu121)
Requirement already satisfied: filelock in
/usr/local/lib/python3.10/dist-packages (from transformers) (3.16.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.24.0 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.27.0)
```

```
Requirement already satisfied: numpy>=1.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (1.26.4)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from transformers) (24.2)
Requirement already satisfied: pyyaml>=5.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers)
(2024.11.6)
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (from transformers) (2.32.3)
Requirement already satisfied: tokenizers<0.22,>=0.21 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.21.0)
Requirement already satisfied: safetensors>=0.4.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.4.5)
Requirement already satisfied: tqdm>=4.27 in
/usr/local/lib/python3.10/dist-packages (from transformers) (4.67.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/usr/local/lib/python3.10/dist-packages (from torch) (4.12.2)
Requirement already satisfied: networkx in
/usr/local/lib/python3.10/dist-packages (from torch) (3.4.2)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.10/dist-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in
/usr/local/lib/python3.10/dist-packages (from torch) (2024.10.0)
Requirement already satisfied: sympy==1.13.1 in
/usr/local/lib/python3.10/dist-packages (from torch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch)
(1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(3.4.0)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(2024.12.14)
```

```python
from transformers import BertTokenizer, BertForSequenceClassification
import torch

# Load pre-trained mBERT and tokenizer
model_name = "bert-base-multilingual-cased"
```

```
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name,
num_labels=5)
```

{"model_id":"05c20d0de2cf40fbb94c847434f6b426","version_major":2,"version_minor":0}

{"model_id":"a55cb30279e14cb68bca0292a7bcdbcb","version_major":2,"version_minor":0}

{"model_id":"90b72d248ff84f52947ddf3ef2a4ece1","version_major":2,"version_minor":0}

{"model_id":"b2b2791a910a468ca3772c98323f4bbd","version_major":2,"version_minor":0}

{"model_id":"d8886a1712c54f44a474abc42b736f33","version_major":2,"version_minor":0}

```
Some weights of BertForSequenceClassification were not initialized
from the model checkpoint at bert-base-multilingual-cased and are
newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able
to use it for predictions and inference.
```

A number of animated shows are written with Romaji in the titles (for example: '*Net-Juu no Susume*', the first entry in the dataset).

To process those words, we'll need to use Multilingual BERT tokenizer.

```
import torch
from torch.utils.data import DataLoader, TensorDataset
from transformers import BertTokenizer, BertForSequenceClassification
import pandas as pd

# Load pre-trained multilingual BERT and tokenizer
model_name = "bert-base-multilingual-cased"
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name,
num_labels=5)
```

```
Some weights of BertForSequenceClassification were not initialized
from the model checkpoint at bert-base-multilingual-cased and are
newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able
to use it for predictions and inference.
```

```
import string
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS

# Preprocess the titles
```

```
def preprocess_title(title):
    title = title.lower()  # Convert to lowercase
    title = title.translate(str.maketrans('', '', string.punctuation))
# Remove punctuation
    title = ' '.join([word for word in title.split() if word not in
ENGLISH_STOP_WORDS])  # Remove stop words
    return title

df['processed_title'] = df['Title'].apply(preprocess_title)

# Tokenize all titles at once with padding and truncation
inputs = tokenizer(df['processed_title'].tolist(),
                   padding=True,
                   truncation=True,
                   return_tensors="pt",
                   max_length=128)
```

Now all that's left is to convert the data into a usable form and start training the model.

```
# Convert labels to Pytorch tensor
labels = torch.tensor(df['label'].tolist())

# Split data into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(inputs['input_ids'],
labels, test_size=0.2, random_state=42)

print(X_train.shape)
print(X_val.shape)
print(y_train.shape)
print(y_val.shape)

torch.Size([10112, 42])
torch.Size([2529, 42])
torch.Size([10112])
torch.Size([2529])

# Create TensorDatasets and DataLoaders for batching
train_dataset = TensorDataset(X_train, inputs['attention_mask']
[:len(X_train)], y_train)
val_dataset = TensorDataset(X_val, inputs['attention_mask']
[len(X_train):], y_val)

train_dataloader = DataLoader(train_dataset, batch_size=32,
shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=32, shuffle=False)

# Move model to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(119547, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768,
bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072,
bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768,
bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
```

```
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=5, bias=True)
)

# Training setup for multiclass classification
optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)
# Enter training mode
model.train()

BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(119547, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768,
bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072,
bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768,
bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12,
```

```
elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=5, bias=True)
)
```

```python
from sklearn.metrics import accuracy_score
import numpy as np

# Make an early stopping callback
class EarlyStopping:
    def __init__(self, patience=3, delta=0, model=None):
        self.patience = patience
        self.delta = delta
        self.best_loss = np.inf
        self.best_epoch = 0
        self.counter = 0

        # Save best weights here
        self.best_model_wts = None

        # Pass the model to save best weights
        self.model = model

    def should_stop(self, val_loss, epoch):
        if val_loss < self.best_loss - self.delta:
            self.best_loss = val_loss
            self.best_epoch = epoch
            self.counter = 0
            # Save the model weights
            self.best_model_wts = self.model.state_dict()
            return False
        else:
            self.counter += 1
            if self.counter >= self.patience:
                print(f"Early stopping at epoch {epoch+1}.")
                return True
            return False

# Train model with early stopping callback and model-saving.
early_stopping = EarlyStopping(patience=3, delta=0.01, model=model)
```

```
for epoch in range(20):
    total_loss = 0
    model.train()
    for batch in train_dataloader:
        input_ids, attention_mask, labels = [item.to(device) for item
in batch]
        optimizer.zero_grad()
        outputs = model(input_ids, attention_mask=attention_mask,
labels=labels)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    print(f"Epoch {epoch+1}, Training Loss: {total_loss /
len(train_dataloader)}")

    # Validation step - go into evaluation mode
    model.eval()
    val_labels = []
    val_preds = []
    val_loss = 0
    with torch.no_grad():
        for batch in val_dataloader:
            input_ids, attention_mask, labels = [item.to(device) for
item in batch]
            outputs = model(input_ids, attention_mask=attention_mask,
labels=labels)
            loss = outputs.loss
            logits = outputs.logits
            predictions = torch.argmax(logits, dim=-1)
            val_loss += loss.item()
            val_labels.extend(labels.cpu().numpy())
            val_preds.extend(predictions.cpu().numpy())

    avg_val_loss = val_loss / len(val_dataloader)
    print(f"Epoch {epoch+1}, Test Loss: {avg_val_loss}")

    accuracy = accuracy_score(val_labels, val_preds)
    print(f"Epoch {epoch+1}, Test Accuracy: {accuracy}")
    print()

    # Check for early stopping and save best model
    if early_stopping.should_stop(avg_val_loss, epoch):
        # Load the best weights (i.e., model with lowest validation
loss)
        model.load_state_dict(early_stopping.best_model_wts)
        # Save the best model
        torch.save(model.state_dict(), 'best_model_weights.pth')
        break
```

```
Epoch 1, Training Loss: 1.487812046008774
Epoch 1, Test Loss: 1.3541480630636216
Epoch 1, Test Accuracy: 0.46026097271648875

Epoch 2, Training Loss: 1.354761969439591
Epoch 2, Test Loss: 1.3196407958865166
Epoch 2, Test Accuracy: 0.47568208778173193

Epoch 3, Training Loss: 1.2935670755709274
Epoch 3, Test Loss: 1.2951722607016563
Epoch 3, Test Accuracy: 0.49466192170818507

Epoch 4, Training Loss: 1.228751333265365
Epoch 4, Test Loss: 1.2671971440315246
Epoch 4, Test Accuracy: 0.5029655990510083

Epoch 5, Training Loss: 1.1250453317844415
Epoch 5, Test Loss: 1.313999319076538
Epoch 5, Test Accuracy: 0.4974298141557928

Epoch 6, Training Loss: 1.0036279055513913
Epoch 6, Test Loss: 1.3721722435206174
Epoch 6, Test Accuracy: 0.49505733491498616

Epoch 7, Training Loss: 0.8685497153031675
Epoch 7, Test Loss: 1.4286280617117881
Epoch 7, Test Accuracy: 0.49545274812178725

Early stopping at epoch 7.
```

Best model had a test loss of 1.2672 and a test accuracy of 0.5030.

Let's evaluate how these predictions were.

```
# Load the best model weights
model.load_state_dict(torch.load('best_model_weights.pth'))
model.eval()  # Set the model to evaluation mode

BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(119547, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
```

```
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768,
bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072,
bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768,
bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=5, bias=True)
)
```

Let's look at the testing / validation dataset for gaining insights as to how it made its predictions.

```
from sklearn.metrics import classification_report


# Define the device (gpu if available)
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Move model to the selected device
model.to(device)

# Assuming `y_val` is already a tensor, move it to the same device as
the model
y_val = y_val.to(device)

with torch.no_grad():
    for batch in val_dataloader:
        # Move tensors to the same device as the model (gpu or cpu)
        # Need to keep track or else errors come up about being in gpu
or cpu
        input_ids, attention_mask, labels = [tensor.to(device) for
tensor in batch]

        # Run the model
        outputs = model(input_ids, attention_mask=attention_mask)
        logits = outputs.logits
        predictions = torch.argmax(logits, dim=-1)

        # Direct comparison of predictions with y_val (both on the
same device)
        # Convert the labels and predictions to numpy on the cpu
        val_preds.extend(predictions.cpu().numpy())
        val_labels.extend(labels.cpu().numpy())

print(classification_report(val_labels, val_preds))
```

```
              precision    recall  f1-score   support

           0       0.50      0.69      0.58       898
           1       0.50      0.46      0.48       390
           2       0.61      0.36      0.46       370
           3       0.36      0.24      0.29       356
           4       0.51      0.44      0.47       515

    accuracy                           0.50      2529
   macro avg       0.49      0.44      0.46      2529
weighted avg       0.50      0.50      0.48      2529
```

```
label_mapping
# For reference
```

```
{'TV': 0, 'Special': 1, 'ONA': 2, 'OVA': 3, 'Movie': 4}
```

Highest F1 score goes to TV Shows with an F1 of 0.58. Lowest F1 score goes to OVAs, with an F1 of 0.29. Much like with the CNNs, it has a hard time distinguishing OVAs from the other types of media.

Lastly, lets try getting some predictions in and out. I will be utilizing the Romaji / English names (aka opposite name) or alternative name from MAL from what is in the dataset.

```
examples = ['Shingeki no Kyojin', 'Fullmetal Alchemist: Brotherhood
OVA Collection', 'Fuuto PI', "Howl's Moving Castle", "Star Blazers:
Space Battleship Yamato 2199", 'Demon Lord 2099', 'Maou 2099']

class_labels = ['TV', 'Special', 'ONA', 'OVA', 'Movie']

inputs = tokenizer(examples, padding=True, truncation=True,
return_tensors="pt", max_length=128)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
inputs = {key: value.to(device) for key, value in inputs.items()}

# Make sure model is also on the correct device
model.to(device)

# Make predictions
with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
    predictions = torch.argmax(logits, dim=-1)  # Get the predicted
class indices

# Convert predictions to a list (if needed) or numpy array
predictions = predictions.cpu().numpy()

predicted_labels = [class_labels[pred] for pred in predictions]
# Assign labels to the predictions

# Print predictions
for title, label in zip(examples, predicted_labels):
    print(f"Title: {title}, Predicted Type: {label}")
```

```
Title: Shingeki no Kyojin, Predicted Type: TV
Title: Fullmetal Alchemist: Brotherhood OVA Collection, Predicted
Type: Special
Title: Fuuto PI, Predicted Type: TV
Title: Howl's Moving Castle, Predicted Type: Movie
Title: Star Blazers: Space Battleship Yamato 2199, Predicted Type:
Movie
Title: Demon Lord 2099, Predicted Type: TV
Title: Maou 2099, Predicted Type: TV
```

6/7 correct. The only incorrect one is Space Battleship Yamato. All of the first 5 are within the dataset but Demon Lord 2099 / Maou 2099 are new.

Baseline Models

Finally, we will construct baseline models to compare to CNN model and our Transformer model.

Episodes

As episode counts are nonstandard or not limited to a group of values, we'll utilize a StandardScaler to limit the impact of long running series on the predictions (though in a sense that might help predictions too).

With a dataset of more than 10000 entries, we'll utilize the L-BFGS solver to speed up calculations and efficiency for a "larger" dataset.

```
ep = df['Episodes']  # Feature = episode count
lab = df['label']  # Target = media type label

from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Split into training and testing sets
eptr, epte, labtr, labte = train_test_split(ep, lab, test_size=0.2,
random_state=42)

eptr = eptr.values.reshape(-1, 1)  # Reshape the Series to 2D

# Scale the features for logistic regression
scaler = StandardScaler()
eptr_scaled = scaler.fit_transform(eptr)
epte_scaled = scaler.transform(epte.values.reshape(-1, 1))

# Make Logistic Regression model using the 'lbfgs' solver
logreg = LogisticRegression(max_iter=200, solver='lbfgs',
multi_class='auto')
logreg.fit(eptr_scaled, labtr)

# Evaluate the model
eptr_acc = logreg.score(eptr_scaled, labtr)  # Training data
epte_acc = logreg.score(epte_scaled, labte)  # Test data

print(f"Training Accuracy: {eptr_acc}")
print(f"Testing Accuracy: {epte_acc}")
```

```
Training Accuracy: 0.5544897151898734
Testing Accuracy: 0.5709766706207987
```

```
eptr, epte, labtr, labte = train_test_split(ep, lab, test_size=0.2,
random_state=42)
eptr = eptr.values.reshape(-1, 1)  # Reshape the Series to 2D
```

```
# Scale the features for SVM
scaler = StandardScaler()
eptr_scaled = scaler.fit_transform(eptr)
epte_scaled = scaler.transform(epte.values.reshape(-1, 1))

# Create and train the Support Vector Machine model
svm = SVC(kernel='rbf', max_iter=200)  # Using RBF kernel, best one
svm.fit(eptr_scaled, labtr)

# Evaluate the model
SVMtr_predictions_eptr = svm.predict(eptr_scaled)
SVMte_predictions_epte = svm.predict(epte_scaled)

SVM_accuracy_eptr = accuracy_score(labtr, SVMtr_predictions_eptr) #
Train
SVM_accuracy_epte = accuracy_score(labte, SVMte_predictions_epte) #
Test

print(f"Training Accuracy: {SVM_accuracy_eptr}")
print(f"Testing Accuracy: {SVM_accuracy_epte}")
```

```
Training Accuracy: 0.3203125
Testing Accuracy: 0.3250296559905101
```

Score

Given that score operates on a 0-10 scale, we will utilize a MinMaxScaler to scale the data.

```
sc = df['Score']  # Feature = score
lab = df['label']  # Target = media type label

sctr, scte, labtr, labte = train_test_split(sc, lab, test_size=0.2,
random_state=42)

sctr = sctr.values.reshape(-1, 1)


scaler = MinMaxScaler()
sctr_scaled = scaler.fit_transform(sctr)
scte_scaled = scaler.transform(scte.values.reshape(-1, 1))

logreg = LogisticRegression(max_iter=200, solver='lbfgs',
multi_class='auto')
logreg.fit(sctr_scaled, labtr)

# Evaluate the model
sctr_acc = logreg.score(sctr_scaled, labtr)
scte_acc = logreg.score(scte_scaled, labte)
```

```
print(f"Training Accuracy: {sctr_acc}")
print(f"Testing Accuracy: {scte_acc}")
```

```
Training Accuracy: 0.3705498417721519
Testing Accuracy: 0.3819691577698695
```

```
sctr, scte, labtr, labte = train_test_split(sc, lab, test_size=0.2,
random_state=42)
sctr = sctr.values.reshape(-1, 1)


scaler = MinMaxScaler()
sctr_scaled = scaler.fit_transform(sctr)
scte_scaled = scaler.transform(scte.values.reshape(-1, 1))


svm = SVC(kernel='linear', max_iter=200)  # 'linear' kernel best here
svm.fit(sctr_scaled, labtr)

# Evaluate the model
SVMtr_predictions_sctr = svm.predict(sctr_scaled)
SVMte_predictions_scte = svm.predict(scte_scaled)

SVM_accuracy_sctr = accuracy_score(labtr, SVMtr_predictions_sctr)
SVM_accuracy_scte = accuracy_score(labte, SVMte_predictions_scte)

print(f"Training Accuracy: {SVM_accuracy_sctr}")
print(f"Testing Accuracy: {SVM_accuracy_scte}")
```

```
Training Accuracy: 0.2633504746835443
Testing Accuracy: 0.2708580466587584
```

Airtime (days)

```
da = df['airtime (days)']  # Feature = airtime in days
lab = df['label']  # Target = media type label


datr, date, labtr, labte = train_test_split(da, lab, test_size=0.2,
random_state=42)

datr = datr.values.reshape(-1, 1)


scaler = StandardScaler()
datr_scaled = scaler.fit_transform(datr)
date_scaled = scaler.transform(date.values.reshape(-1, 1))


logreg = LogisticRegression(max_iter=200, solver='lbfgs',
multi_class='auto')
```

```
logreg.fit(datr_scaled, labtr)


datr_acc = logreg.score(datr_scaled, labtr)
date_acc = logreg.score(date_scaled, labte)

print(f"Training Accuracy: {datr_acc}")
print(f"Testing Accuracy: {date_acc}")
```

```
Training Accuracy: 0.5311511075949367
Testing Accuracy: 0.5476472914195334
```

```
datr, date, labtr, labte = train_test_split(da, lab, test_size=0.2,
random_state=42)
datr = datr.values.reshape(-1, 1)


scaler = StandardScaler()
datr_scaled = scaler.fit_transform(datr)
date_scaled = scaler.transform(date.values.reshape(-1, 1))


svm = SVC(kernel='rbf', max_iter=200)  # 'rbf' kernel best one
svm.fit(datr_scaled, labtr)

# Evaluate the model
SVMtr_predictions_datr = svm.predict(datr_scaled)
SVMte_predictions_date = svm.predict(date_scaled)

SVM_accuracy_datr = accuracy_score(labtr, SVMtr_predictions_datr)
SVM_accuracy_date = accuracy_score(labte, SVMte_predictions_date)

print(f"Training Accuracy: {SVM_accuracy_datr}")
print(f"Testing Accuracy: {SVM_accuracy_date}")
```

```
Training Accuracy: 0.24960443037974683
Testing Accuracy: 0.258204824041123
```

Baseline Results

Utilizing tabulate to provide a better looking presentation of the test accuracies results.

```
pip install tabulate
```

```
Requirement already satisfied: tabulate in
/usr/local/lib/python3.10/dist-packages (0.9.0)
```

```
from tabulate import tabulate

headers = ["Model", "Test Accuracy"]
```

```python
baseline_results = [
    ["LogReg Episodes", epte_acc],
    ["SVM Episodes", SVM_accuracy_epte],
    ["LogReg Scores", scte_acc],
    ["SVM Scores", SVM_accuracy_scte],
    ["LogReg Airtime (days)", date_acc],
    ["SVM Airtime (days)", SVM_accuracy_date]
]

pretty_table = tabulate(baseline_results, headers=headers,
tablefmt="grid")

print("Baseline Models Performance:\n")
print(pretty_table)
```

```
Baseline Models Performance:

+-----------------------+-----------------+
| Model                 |   Test Accuracy |
+=======================+=================+
| LogReg Episodes       |        0.570977 |
+-----------------------+-----------------+
| SVM Episodes          |        0.32503  |
+-----------------------+-----------------+
| LogReg Scores         |        0.381969 |
+-----------------------+-----------------+
| SVM Scores            |        0.270858 |
+-----------------------+-----------------+
| LogReg Airtime (days) |        0.547647 |
+-----------------------+-----------------+
| SVM Airtime (days)    |        0.258205 |
+-----------------------+-----------------+
```

To an extent, these baseline results were expected. The highest test accuracy was for the Logistic Regression models of using Episodes and Airtime (days) to predict the type of show / medium it was.

If a show has multiple episodes, it's more likely to be a TV series. One-episode shows are very likely to be a movie or a special - single entries disproportionately make up those entries. Similarly, longer / seasonal run times are more characteristic of TV shows.

These sorts of simpler, more linear separations may be why we see the logistic regression models for "airtime (days)" and "Episodes" seem to have the highest baseline test accuracy, even more than our cover-image based model.