DS2_Take-Home_NF

April 19, 2024

1 DS2 Take-Home Assignment - Nicolas Fernandez

1.0.1 Kaggle Competition - Predicting Online News Popularity

The task is to create the best possible model for predicting the popularity of a news article from mashable.com and perform best in a kaggle.com competition being held.

The dataset being used can be found here and a link to the kaggle competition overview can be found here.

The data itself, as mentioned above, comes from mashable.com and is data from articles that appeared on the site two years past January 8, 2015 (when the dataset was acquired) or later. The goal is to predict which articles are shared the most on social media using the binary `is_popular` column within the dataset denoting an article being popular with a 1, 0 if not. From the competition, the data is already split into `train.csv` and `test.csv` with the latter being used to generate predictions on popularity for submission to the competition. `train.csv` will be split into training and test sets for creating and testing predictive models.

Both the train and test csv's have a column `article_id` that is solely the index of the articles in the data. This will be used as the index for the dataframes when loaded in. The numbers don't exactly match up with the amount of observations in the dataset but that's not a problem. This will be necessary later when creating the Kaggle submission. Along with this, the `timedelta` variable will be dropped since it is considered non-predictive according to the data description.

The submission scores will be calculated using AUC scores as the loss function per specifications of the competition.

1.1 Loading Data and EDA

[1]:
```python
# Importing required libraries
import pandas as pd
import numpy as np

# Loading train data from csv locally and viewing the contents
raw_data = pd.read_csv('Data/train.csv', index_col='article_id')
raw_data.drop(columns=['timedelta'], inplace=True)
display(raw_data.head())
raw_data.info()
```

```
            n_tokens_title  n_tokens_content  n_unique_tokens  \
article_id
```

```
1                      9            702        0.454545
3                      8           1197        0.470143
5                      9            214        0.618090
6                      8            249        0.621951
7                     12           1219        0.397841

            n_non_stop_words  n_non_stop_unique_tokens  num_hrefs  \
article_id
1                        1.0                  0.620438         11
3                        1.0                  0.666209         21
5                        1.0                  0.748092          5
6                        1.0                  0.664740         16
7                        1.0                  0.583578         21

            num_self_hrefs  num_imgs  num_videos  average_token_length  …  \
article_id                                                               …
1                        2         1           0              4.790598  …
3                        6         2          13              4.622389  …
5                        2         1           0              4.476636  …
6                        5         8           0              5.180723  …
7                        1         1           2              4.659557  …

            min_positive_polarity  max_positive_polarity  \
article_id
1                            0.10               1.000000
3                            0.05               1.000000
5                            0.10               0.433333
6                            0.10               0.500000
7                            0.05               0.800000

            avg_negative_polarity  min_negative_polarity  \
article_id
1                       -0.153395                   -0.4
3                       -0.308167                   -1.0
5                       -0.141667                   -0.2
6                       -0.500000                   -0.8
7                       -0.441111                   -1.0

            max_negative_polarity  title_subjectivity  \
article_id
1                           -0.10                 0.0
3                           -0.10                 0.0
5                           -0.05                 0.0
6                           -0.40                 0.0
7                           -0.05                 0.0

            title_sentiment_polarity  abs_title_subjectivity  \
article_id
```

```
1                                0.0                      0.5
3                                0.0                      0.5
5                                0.0                      0.5
6                                0.0                      0.5
7                                0.0                      0.5

            abs_title_sentiment_polarity  is_popular
article_id
1                                    0.0           0
3                                    0.0           0
5                                    0.0           0
6                                    0.0           0
7                                    0.0           0

[5 rows x 59 columns]
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 29733 entries, 1 to 39643
Data columns (total 59 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   n_tokens_title                 29733 non-null  int64
 1   n_tokens_content               29733 non-null  int64
 2   n_unique_tokens                29733 non-null  float64
 3   n_non_stop_words               29733 non-null  float64
 4   n_non_stop_unique_tokens       29733 non-null  float64
 5   num_hrefs                      29733 non-null  int64
 6   num_self_hrefs                 29733 non-null  int64
 7   num_imgs                       29733 non-null  int64
 8   num_videos                     29733 non-null  int64
 9   average_token_length           29733 non-null  float64
 10  num_keywords                   29733 non-null  int64
 11  data_channel_is_lifestyle      29733 non-null  int64
 12  data_channel_is_entertainment  29733 non-null  int64
 13  data_channel_is_bus            29733 non-null  int64
 14  data_channel_is_socmed         29733 non-null  int64
 15  data_channel_is_tech           29733 non-null  int64
 16  data_channel_is_world          29733 non-null  int64
 17  kw_min_min                     29733 non-null  int64
 18  kw_max_min                     29733 non-null  float64
 19  kw_avg_min                     29733 non-null  float64
 20  kw_min_max                     29733 non-null  int64
 21  kw_max_max                     29733 non-null  int64
 22  kw_avg_max                     29733 non-null  float64
 23  kw_min_avg                     29733 non-null  float64
 24  kw_max_avg                     29733 non-null  float64
 25  kw_avg_avg                     29733 non-null  float64
 26  self_reference_min_shares      29733 non-null  float64
```

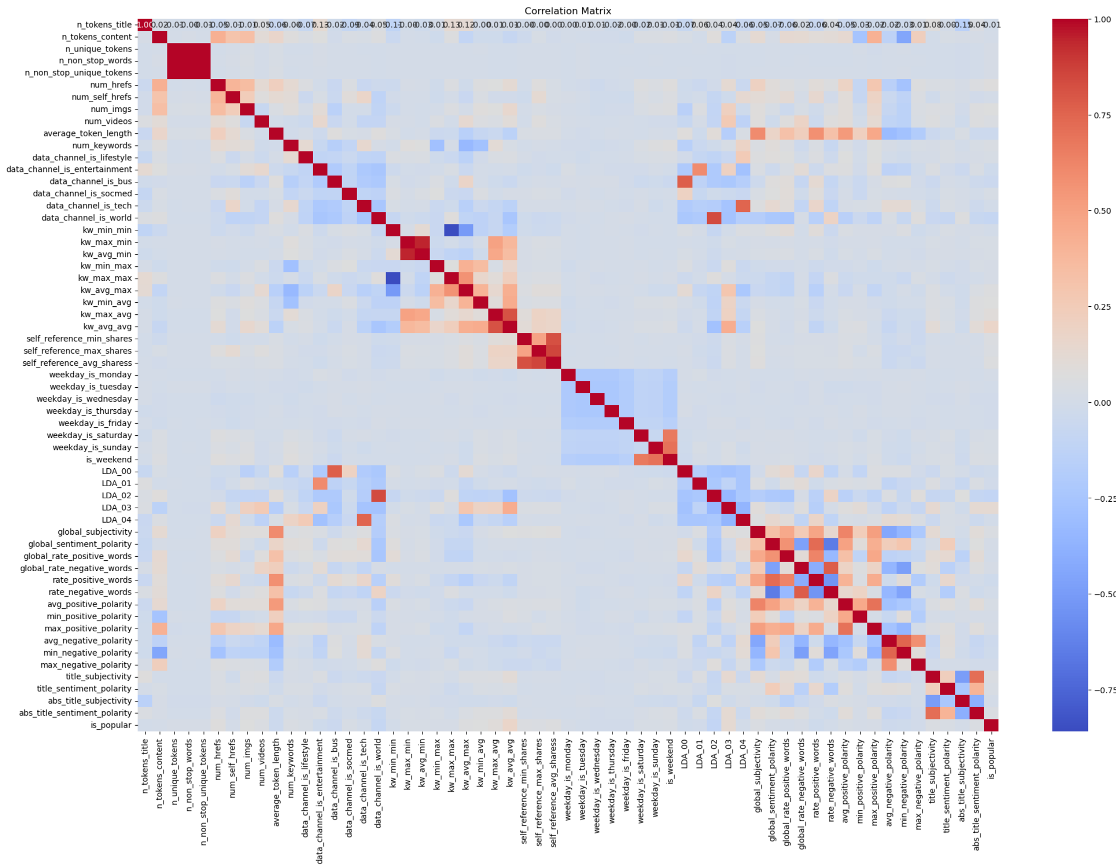
```
 27  self_reference_max_shares      29733 non-null  float64
 28  self_reference_avg_sharess     29733 non-null  float64
 29  weekday_is_monday              29733 non-null  int64
 30  weekday_is_tuesday             29733 non-null  int64
 31  weekday_is_wednesday           29733 non-null  int64
 32  weekday_is_thursday            29733 non-null  int64
 33  weekday_is_friday              29733 non-null  int64
 34  weekday_is_saturday            29733 non-null  int64
 35  weekday_is_sunday              29733 non-null  int64
 36  is_weekend                     29733 non-null  int64
 37  LDA_00                         29733 non-null  float64
 38  LDA_01                         29733 non-null  float64
 39  LDA_02                         29733 non-null  float64
 40  LDA_03                         29733 non-null  float64
 41  LDA_04                         29733 non-null  float64
 42  global_subjectivity            29733 non-null  float64
 43  global_sentiment_polarity      29733 non-null  float64
 44  global_rate_positive_words     29733 non-null  float64
 45  global_rate_negative_words     29733 non-null  float64
 46  rate_positive_words            29733 non-null  float64
 47  rate_negative_words            29733 non-null  float64
 48  avg_positive_polarity          29733 non-null  float64
 49  min_positive_polarity          29733 non-null  float64
 50  max_positive_polarity          29733 non-null  float64
 51  avg_negative_polarity          29733 non-null  float64
 52  min_negative_polarity          29733 non-null  float64
 53  max_negative_polarity          29733 non-null  float64
 54  title_subjectivity             29733 non-null  float64
 55  title_sentiment_polarity       29733 non-null  float64
 56  abs_title_subjectivity         29733 non-null  float64
 57  abs_title_sentiment_polarity   29733 non-null  float64
 58  is_popular                     29733 non-null  int64
dtypes: float64(34), int64(25)
memory usage: 13.6 MB
```

There are no NaN values within the data however there could imputed values that can be interpreted as missing. This will be explored further. One question to be explored if a dummy variable is necessary for each day of the week or only weekends are important, and same for the genre of the article.

First however is to check the correlation of the features of this dataset.

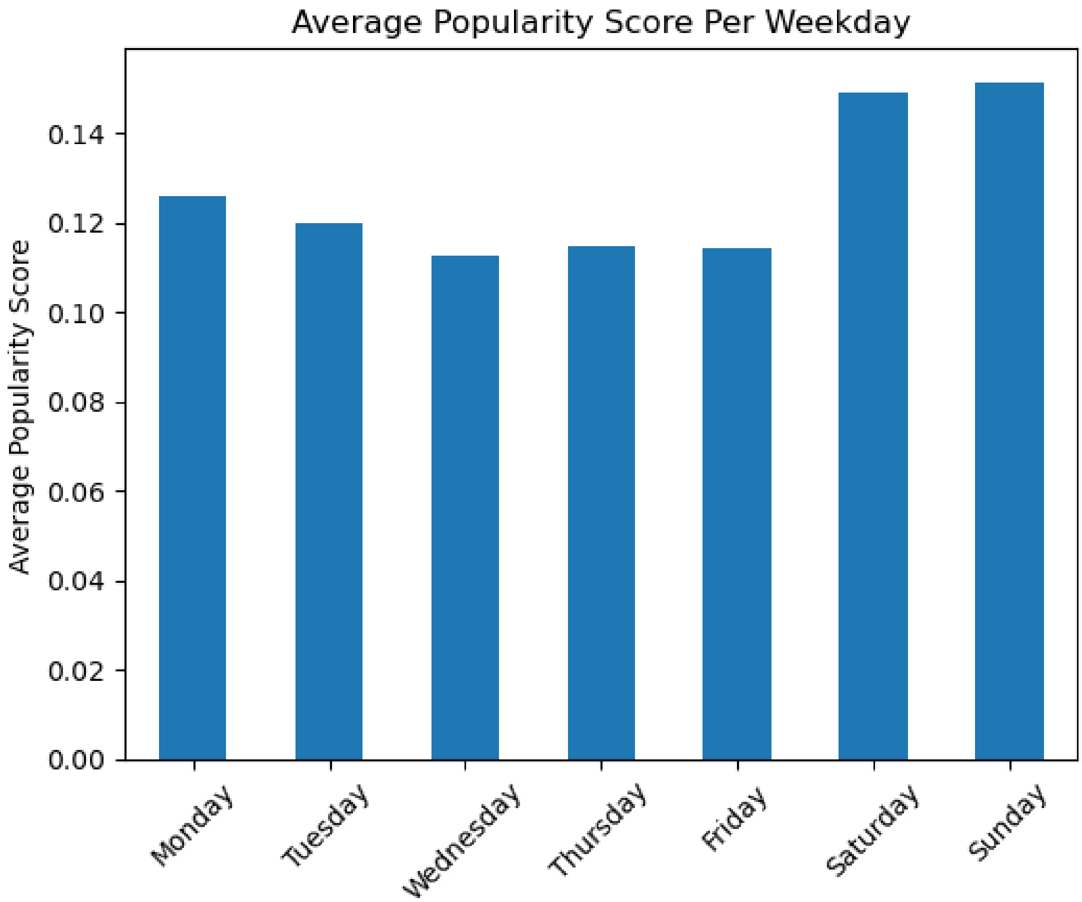
[2]:
```
# Importing required libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Creating a correlation matrix plot
plt.figure(figsize=(24,16))
```

```
sns.heatmap(raw_data.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix')

plt.show()
```



From viewing the correlation matrix, we can infer the following: - The `LDA_` columns refer specifically to the closeness of the article of a genre specified in the data. `LDA_00` refers to Business, `LDA_01` to Entertainment, `LDA_02` to World, and `LDA_04` to Tech. `LDA_03` given this ordering would imply it refers to Social Media however it does not appear to be significantly correlated as such. In any case these variables are now interpretable and will be kept. - Amongst all the `kw_` columns, the `kw_avg_avg` column is positively correlated with all other `kw_` columns. Therefore out of all of these columns only `kw_avg_avg` will be kept as it is also the most interpretable of the features since it is the average amount of shares of all keywords in an article - The `self_reference` columns are all highly correlated to each other. From these three, only `self_reference_avg_sharess` will be kept. This refers to the average shares of articles referenced within Mashable, which might be good for predicting for this dataset but also would harm any potential external validity for models trained on it - Each `weekday_is_` column seems negatively correlated with each other except for Saturday and Sunday. This will be explored further below - `n_unique_tokens`, `n_non_stop_unique_tokens`, and `n_non_stop_unique_tokens` are all extremely correlated. Only `n_unique_tokens` will be kept - `num_hrefs` and `num_self_hrefs` are measuring similar things and are correlated. Only `num_hrefs` will be kept in an attempt to aid external validity since it refers to all links and not just links to other Mashable.com articles - All of the `polarity` and `rate_positive/negative` columns are very correlated, either positively or negatively. Only the global rates and average columns out of these will be kept to make sure this information is captured without too many redundancies - For the `title_` columns, the `abs_` columns will be dropped as they are harder to interpret and also are captured already in the previous two features

[3]:
```
# Creating a copy of raw_data to preserve it just in case
data = raw_data.copy()

# Creating list of features to exclude from initial feature selection
exclude_cols = []
for col in data.columns:
    if col.startswith('kw_') and col != 'kw_avg_avg':
        exclude_cols.append(col)
    elif col.startswith('self_') and not 'avg' in col:
        exclude_cols.append(col)
    elif col.startswith('n_non_'):
        exclude_cols.append(col)
    elif col == 'num_self_href':
        exclude_cols.append(col)
    elif col.startswith('rate_'):
        exclude_cols.append(col)
    elif 'polarity' in col and not col.startswith('avg') and not col.
 ↪startswith('global') and not col.startswith('title'):
        exclude_cols.append(col)
    elif 'subjectivity' in col and not col.startswith('global') and not col.
 ↪startswith('title'):
        exclude_cols.append(col)

# Getting all weekday columns set to a list
weekdays = [col for col in data.columns if col.startswith('weekday_')]
weekday_names = [day.replace('weekday_is_', '').capitalize() for day in␣
 ↪weekdays]

# Creating dataframe that shows average popularity score for each weekday
avg_weekday_pop_df = data.groupby(weekdays)['is_popular'].mean()[::-1]

# Creating a bar plot to show the average scores per weekday
avg_weekday_pop_df.plot(kind='bar')
plt.xlabel(None)
plt.ylabel('Average Popularity Score')
plt.title('Average Popularity Score Per Weekday')

plt.xticks(range(len(weekday_names)), weekday_names, rotation=45)

plt.show()
```

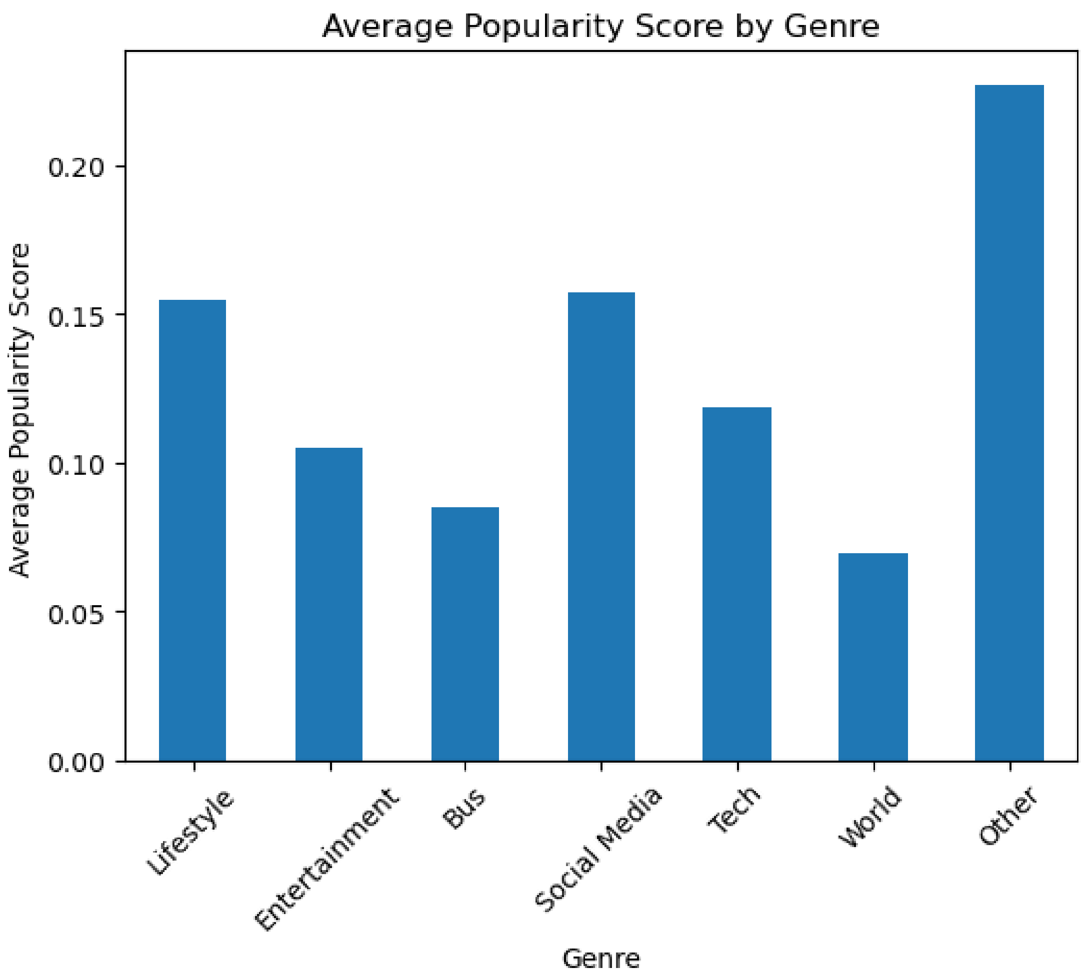


From this plot we can see that the most influential days from the data are the weekends (Saturday and Sunday) with the earlier weekdays being slightly more influential than the latter ones. Using this information, the dummy variables for each weekday will be dropped, the already existing `is_weekend` column will be kept but renamed to `d_weekend` to flag it as a dummy variable, and a new dummy variable `d_mon_tues` will be created to flag whether the article was released on either Monday or Tuesday.

Next, the same examination will be done for article genres.

[4]:
```
# Creating Monday/Tuesday dummy variable
data['d_mon_tues'] = data['weekday_is_monday'] + data['weekday_is_tuesday']

# Renaming is_weekend to d_weekend
data.rename(columns={'is_weekend': 'd_weekend'}, inplace=True)

# Dropping all weekday_is_ columns from data since they are no longer necessary
data.drop(columns=weekdays, inplace=True)

# Repeating the same plot above but with article genres
```

```
genres = [col for col in data.columns if col.startswith('data_channel_is_')]
genre_names = [genre.replace('data_channel_is_', '').capitalize() for genre in␣
↪genres]
genre_names[genre_names.index('Socmed')] = 'Social Media'
genre_names.append('Other')

genre_pop_df = data.groupby(genres)['is_popular'].mean()[::-1]

genre_pop_df.plot(kind='bar')

plt.xlabel('Genre')
plt.ylabel('Average Popularity Score')
plt.title('Average Popularity Score by Genre')

plt.xticks(range(len(genre_names)), genre_names, rotation=45)

plt.show()
```

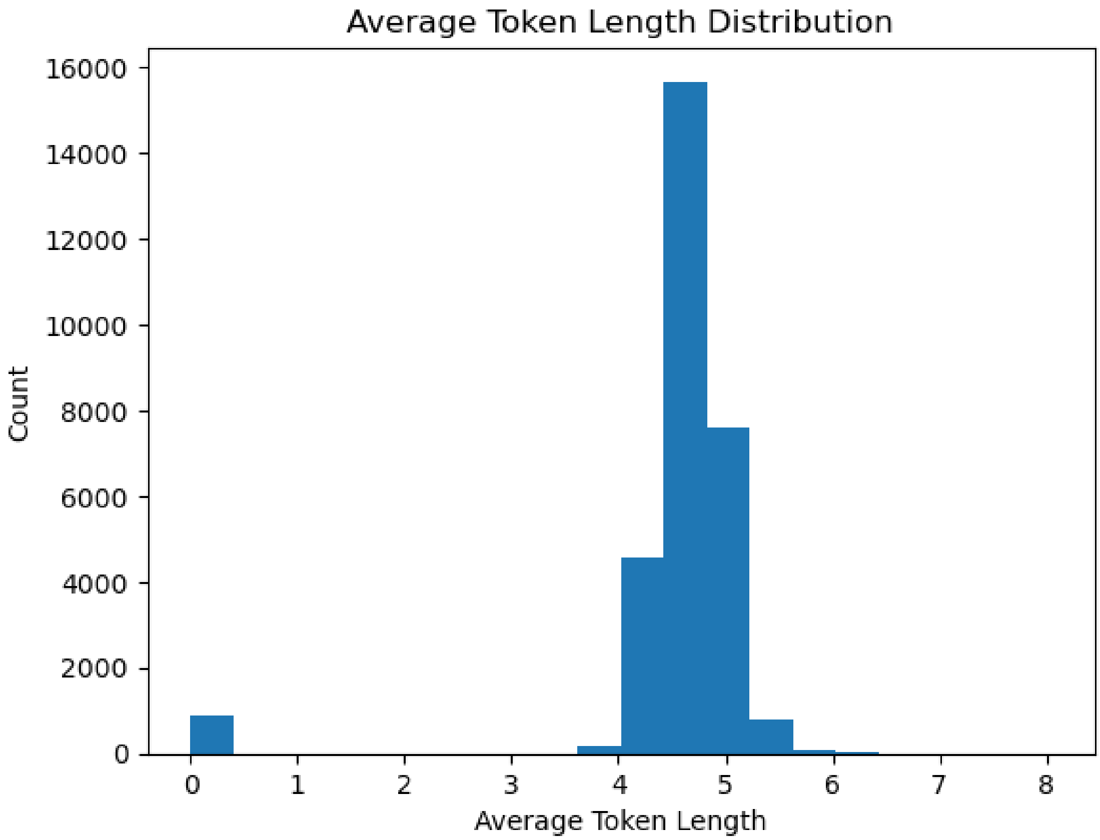


From reviewing the data there were a couple observations: - Despite there being 6 genre classifiers in the data there is a 7th occurence for articles that don't fall under any of the labelled genres. For the purposes of the plot they were labelled as `Other` - Articles not falling into any of the 6 classified genres are significantly more popular than the rest - Amongst the articles denoted by a specific genre, the `Lifestyle` and `Social Media` genres are the most significant predictors, roughly the same - The `World` genre appears to be the least popular genre, somewhat significantly so

From this analysis it appears that the genre of the article has significance for determining popularity but that different groupings may not make sense. These columns will be left as is within the data but renamed to flag them as dummy variables with the following syntax: `d_genre_`

Next a histogram of the `average_token_length` will be examined.

[5]:
```
# Renaming genre dummy variables
[data.rename(columns={genre: genre.replace('data_channel_is_', 'd_genre_')},␣
 ↪inplace=True) for genre in genres]

# Generating a historgram of `average_token_length`
plt.hist(data['average_token_length'], bins=20)

plt.xlabel('Average Token Length')
plt.ylabel('Count')
plt.title('Average Token Length Distribution')

plt.show()
```



From this we can see that the distribution of average token length of articles in the data is approximating a normal distribution around a mean of roughly 4.5. The noteworthy values here are the significant portion of values with an average token length of 0. These occurrences will be explored in more depth to figure out what exactly these represent. It is likely that these articles are slideshows and/or videos rather than traditional prose (or a combination thereof).

[6]:
```
# Viewing descriptive statistics for occurences where `average_token_length`␣
↪equals 0
data[data['average_token_length'] == 0].describe()
```

[6]:
```
       n_tokens_title  n_tokens_content  n_unique_tokens  n_non_stop_words  \
count      867.000000             867.0            867.0             867.0
mean        10.850058               0.0              0.0               0.0
std          2.036760               0.0              0.0               0.0
min          5.000000               0.0              0.0               0.0
25%          9.000000               0.0              0.0               0.0
50%         11.000000               0.0              0.0               0.0
75%         12.000000               0.0              0.0               0.0
max         16.000000               0.0              0.0               0.0
```

```
      n_non_stop_unique_tokens  num_hrefs  num_self_hrefs    num_imgs  \
count                    867.0      867.0           867.0  867.000000
mean                       0.0        0.0             0.0    3.747405
std                        0.0        0.0             0.0    8.606464
min                        0.0        0.0             0.0    0.000000
25%                        0.0        0.0             0.0    0.000000
50%                        0.0        0.0             0.0    0.000000
75%                        0.0        0.0             0.0    1.000000
max                        0.0        0.0             0.0  100.000000

      num_videos  average_token_length  …  max_positive_polarity  \
count 867.000000                 867.0  …                  867.0
mean    0.792388                   0.0  …                    0.0
std     1.155682                   0.0  …                    0.0
min     0.000000                   0.0  …                    0.0
25%     0.000000                   0.0  …                    0.0
50%     1.000000                   0.0  …                    0.0
75%     1.000000                   0.0  …                    0.0
max    24.000000                   0.0  …                    0.0

      avg_negative_polarity  min_negative_polarity  max_negative_polarity  \
count                 867.0                  867.0                  867.0
mean                    0.0                    0.0                    0.0
std                     0.0                    0.0                    0.0
min                     0.0                    0.0                    0.0
25%                     0.0                    0.0                    0.0
50%                     0.0                    0.0                    0.0
75%                     0.0                    0.0                    0.0
max                     0.0                    0.0                    0.0

      title_subjectivity  title_sentiment_polarity  abs_title_subjectivity  \
count         867.000000                867.000000              867.000000
mean            0.349305                  0.088811                0.317413
std             0.339876                  0.295841                0.193348
min             0.000000                 -1.000000                0.000000
25%             0.000000                  0.000000                0.133333
50%             0.333333                  0.000000                0.400000
75%             0.600000                  0.250000                0.500000
max             1.000000                  1.000000                0.500000

      abs_title_sentiment_polarity  is_popular  d_mon_tues
count                   867.000000  867.000000  867.000000
mean                      0.194028    0.191465    0.359862
std                       0.240267    0.393681    0.480237
min                       0.000000    0.000000    0.000000
25%                       0.000000    0.000000    0.000000
50%                       0.125000    0.000000    0.000000
```

```
75%                           0.300000     0.000000     1.000000
max                           1.000000     1.000000     1.000000

[8 rows x 53 columns]
```

[7]:
```
# Viewing descriptive statistics for occurrences where `average_token_length`␣
↪is not equal to 0
data[data['average_token_length'] != 0].describe()
```

[7]:
```
       n_tokens_title  n_tokens_content  n_unique_tokens  n_non_stop_words  \
count    28866.000000      28866.000000     28866.000000      28866.000000
mean        10.377018        561.377780         0.571748          1.036063
std          2.110788        466.609145         4.124008          6.127135
min          2.000000         18.000000         0.114964          1.000000
25%          9.000000        258.000000         0.477781          1.000000
50%         10.000000        422.000000         0.543646          1.000000
75%         12.000000        725.000000         0.611792          1.000000
max         23.000000       8474.000000       701.000000       1042.000000

       n_non_stop_unique_tokens     num_hrefs  num_self_hrefs      num_imgs  \
count              28866.000000  28866.000000    28866.000000  28866.000000
mean                   0.716319     11.240456        3.389628      4.547876
std                    3.823021     11.323671        3.854916      8.200756
min                    0.119134      0.000000        0.000000      0.000000
25%                    0.632886      5.000000        1.000000      1.000000
50%                    0.693767      8.000000        3.000000      1.000000
75%                    0.757322     14.000000        4.000000      4.000000
max                  650.000000    304.000000       74.000000    111.000000

         num_videos  average_token_length  …  max_positive_polarity  \
count  28866.000000          28866.000000  …           28866.000000
mean       1.277697              4.688573  …               0.780540
std        4.246004              0.283318  …               0.212662
min        0.000000              3.600000  …               0.000000
25%        0.000000              4.496245  …               0.600000
50%        0.000000              4.676255  …               0.800000
75%        1.000000              4.863729  …               1.000000
max       91.000000              8.041534  …               1.000000

       avg_negative_polarity  min_negative_polarity  max_negative_polarity  \
count           28866.000000           28866.000000           28866.000000
mean               -0.267509              -0.536629              -0.111030
std                 0.122140               0.280179               0.095229
min                -1.000000              -1.000000              -1.000000
25%                -0.331818              -0.714286              -0.125000
50%                -0.256944              -0.500000              -0.100000
75%                -0.193056              -0.300000              -0.050000
```

```
max                  0.000000               0.000000               0.000000

       title_subjectivity  title_sentiment_polarity  abs_title_subjectivity  \
count        28866.000000              28866.000000            28866.000000
mean             0.279852                  0.069116                0.342149
std              0.322743                  0.263360                0.188551
min              0.000000                 -1.000000                0.000000
25%              0.000000                  0.000000                0.166667
50%              0.125000                  0.000000                0.500000
75%              0.500000                  0.136364                0.500000
max              1.000000                  1.000000                0.500000

       abs_title_sentiment_polarity    is_popular     d_mon_tues
count                  28866.000000  28866.000000  28866.000000
mean                       0.154069      0.119552      0.354847
std                        0.224494      0.324443      0.478475
min                        0.000000      0.000000      0.000000
25%                        0.000000      0.000000      0.000000
50%                        0.000000      0.000000      0.000000
75%                        0.250000      0.000000      1.000000
max                        1.000000      1.000000      1.000000

[8 rows x 53 columns]
```

Viewing of the descriptive statistics of the two scenarios it appears as if the original assumption was correct, that these occurrences in the data are slideshows and/or videos with no text in the content. They cannot be dropped however as they are clearly not errors in the data entry. A dummy variable **d_no_words** will be created to represent this scenario.

Besides that, there is an anomolous looking entry given the descriptive statistics of **n_unique_tokens**. This will be examined first.

[8]:
```
data[data['n_unique_tokens'] == 701]
```

[8]:
```
            n_tokens_title  n_tokens_content  n_unique_tokens  \
article_id
32967                    9              1570            701.0

            n_non_stop_words  n_non_stop_unique_tokens  num_hrefs  \
article_id
32967                 1042.0                     650.0         11

            num_self_hrefs  num_imgs  num_videos  average_token_length  …  \
article_id                                                              …
32967                   10        51           0              4.696178  …

            max_positive_polarity  avg_negative_polarity  \
article_id
```

```
32967                        0.0                     0.0

             min_negative_polarity  max_negative_polarity  title_subjectivity  \
article_id
32967                        0.0                     0.0                 0.0

             title_sentiment_polarity  abs_title_subjectivity  \
article_id
32967                           0.0                     0.0

             abs_title_sentiment_polarity  is_popular  d_mon_tues
article_id
32967                               0.0           1           1

[1 rows x 53 columns]
```

Looking at the values of some of the features for this observation, almost all of the columns that are supposed to be rates between 0-1 are far exceeding that. There is clearly some type of an error in the data with this observation which is throwing the descriptive statistics out of whack. A decision to drop this observation will be made below.

Another thing noted from the non-zero average token length is that the standard deviation of `n_tokens_content` is very large. For reference, the values at and above the 99% range will be viewed closer.

[9]:
```python
# Dropping the value via the index (article_id 32967)
data.drop(index=32967, inplace=True)

# Creating a dummy variable for when `average_token_length` is equal 0
data['d_no_words'] = (data['average_token_length'] == 0).astype(int)

# Calculating the 95% of n_tokens_content
content_99 = data['n_tokens_content'].quantile(.99)

# Viewing descriptive statistics at and above 99% quantile value
data[data['n_tokens_content'] >= content_99].describe()
```

[9]:
```
       n_tokens_title  n_tokens_content  n_unique_tokens  n_non_stop_words  \
count      298.000000        298.000000       298.000000      2.980000e+02
mean        10.479866       2976.929530         0.327987      1.000000e+00
std          2.412355        918.705812         0.057942      1.237282e-10
min          5.000000       2258.000000         0.114964      1.000000e+00
25%          9.000000       2430.750000         0.298184      1.000000e+00
50%         11.000000       2671.000000         0.332264      1.000000e+00
75%         12.000000       3138.250000         0.365665      1.000000e+00
max         18.000000       8474.000000         0.462878      1.000000e+00

       n_non_stop_unique_tokens   num_hrefs  num_self_hrefs    num_imgs  \
```

count	298.000000	298.000000	298.000000	298.000000
mean	0.494121	30.875839	8.278523	18.073826
std	0.083742	28.304287	9.485392	21.441988
min	0.129263	0.000000	0.000000	0.000000
25%	0.461614	10.000000	2.000000	1.000000
50%	0.500000	21.000000	5.000000	10.000000
75%	0.549221	43.000000	12.000000	30.750000
max	0.672108	159.000000	74.000000	111.000000

	num_videos	average_token_length	…	avg_negative_polarity	\
count	298.000000	298.000000	…	298.000000	
mean	4.348993	4.579441	…	-0.284523	
std	14.483208	0.242161	…	0.057095	
min	0.000000	3.785066	…	-0.473922	
25%	0.000000	4.433234	…	-0.319522	
50%	0.000000	4.592008	…	-0.273092	
75%	1.000000	4.721814	…	-0.240777	
max	75.000000	5.318665	…	-0.134524	

	min_negative_polarity	max_negative_polarity	title_subjectivity	\
count	298.000000	298.000000	298.000000	
mean	-0.865257	-0.052762	0.304710	
std	0.164475	0.029523	0.333556	
min	-1.000000	-0.400000	0.000000	
25%	-1.000000	-0.050000	0.000000	
50%	-1.000000	-0.050000	0.233333	
75%	-0.800000	-0.050000	0.500000	
max	-0.155556	-0.008333	1.000000	

	title_sentiment_polarity	abs_title_subjectivity	\
count	298.000000	298.000000	
mean	0.128655	0.336460	
std	0.270272	0.189578	
min	-0.600000	0.000000	
25%	0.000000	0.152327	
50%	0.000000	0.500000	
75%	0.250000	0.500000	
max	1.000000	0.500000	

	abs_title_sentiment_polarity	is_popular	d_mon_tues	d_no_words
count	298.000000	298.000000	298.000000	298.0
mean	0.171020	0.187919	0.362416	0.0
std	0.245578	0.391305	0.481507	0.0
min	0.000000	0.000000	0.000000	0.0
25%	0.000000	0.000000	0.000000	0.0
50%	0.005976	0.000000	0.000000	0.0
75%	0.283036	0.000000	1.000000	0.0

```
max                        1.000000    1.000000    1.000000         0.0

[8 rows x 54 columns]
```

[10]:
```python
# Viewing descriptive statistics at 75% quartile and below
content_75 = data['n_tokens_content'].quantile(.75)
data[data['n_tokens_content'] <= content_75].describe()
```

[10]:
```
       n_tokens_title  n_tokens_content  n_unique_tokens  n_non_stop_words  \
count    22300.000000      22300.000000     22300.000000      22300.000000
mean        10.378206        338.894081         0.563366          0.961121
std          2.089012        174.042729         0.140219          0.193311
min          2.000000          0.000000         0.000000          0.000000
25%          9.000000        209.000000         0.520773          1.000000
50%         10.000000        318.000000         0.572816          1.000000
75%         12.000000        466.000000         0.632083          1.000000
max         19.000000        712.000000         1.000000          1.000000

       n_non_stop_unique_tokens     num_hrefs  num_self_hrefs      num_imgs  \
count              22300.000000  22300.000000    22300.000000  22300.000000
mean                   0.695123      8.791704        2.754933      3.370628
std                    0.164975      8.145548        2.432475      5.950367
min                    0.000000      0.000000        0.000000      0.000000
25%                    0.660606      4.000000        1.000000      1.000000
50%                    0.716783      6.000000        2.000000      1.000000
75%                    0.775510     11.000000        4.000000      2.000000
max                    1.000000    118.000000       65.000000    100.000000

         num_videos  average_token_length  …  avg_negative_polarity  \
count  22300.000000          22300.000000  …           22300.000000
mean       1.028879              4.519160  …              -0.253503
std        3.205206              0.952478  …               0.141229
min        0.000000              0.000000  …              -1.000000
25%        0.000000              4.482534  …              -0.331944
50%        0.000000              4.677923  …              -0.245833
75%        1.000000              4.872306  …              -0.166667
max       59.000000              8.041534  …               0.000000

       min_negative_polarity  max_negative_polarity  title_subjectivity  \
count           22300.000000           22300.000000        22300.000000
mean               -0.458067              -0.119923            0.280851
std                 0.281218               0.105838            0.322381
min                -1.000000              -1.000000            0.000000
25%                -0.600000              -0.150000            0.000000
50%                -0.500000              -0.100000            0.144444
75%                -0.250000              -0.050000            0.500000
max                 0.000000               0.000000            1.000000
```

```
       title_sentiment_polarity  abs_title_subjectivity  \
count              22300.000000            22300.000000
mean                   0.067488                0.340745
std                    0.259226                0.189330
min                   -1.000000                0.000000
25%                    0.000000                0.165584
50%                    0.000000                0.500000
75%                    0.136364                0.500000
max                    1.000000                0.500000

       abs_title_sentiment_polarity    is_popular    d_mon_tues    d_no_words
count                  22300.000000  22300.000000  22300.000000  22300.000000
mean                       0.152125      0.119327      0.357354      0.038879
std                        0.220477      0.324181      0.479231      0.193311
min                        0.000000      0.000000      0.000000      0.000000
25%                        0.000000      0.000000      0.000000      0.000000
50%                        0.000000      0.000000      0.000000      0.000000
75%                        0.250000      0.000000      1.000000      0.000000
max                        1.000000      1.000000      1.000000      1.000000

[8 rows x 54 columns]
```

[11]:
```
# Viewing the last 20 observations when sorted by n_tokens_content
data.sort_values('n_tokens_content').tail(20)
```

[11]:
```
            n_tokens_title  n_tokens_content  n_unique_tokens  \
article_id
28698                    8              4155         0.315675
36127                   14              4172         0.312469
21810                    8              4306         0.379726
6441                    12              4331         0.304833
27810                   10              4452         0.293559
20214                   14              4462         0.346752
154                     12              4514         0.312103
22290                    9              4574         0.322843
35561                   11              4661         0.264441
4217                    13              4878         0.288027
25916                    8              4979         0.209687
16145                   16              5553         0.338127
5679                    15              6159         0.242384
16618                   14              6505         0.365745
29343                   11              7002         0.166082
8781                    11              7034         0.165891
10999                   12              7081         0.249398
18536                   10              7413         0.173769
12431                   18              7764         0.226452
```

19276	9	8474	0.188211

	n_non_stop_words	n_non_stop_unique_tokens	num_hrefs	\
article_id				
28698	1.0	0.547187	7	
36127	1.0	0.498939	25	
21810	1.0	0.593688	20	
6441	1.0	0.542174	5	
27810	1.0	0.482619	39	
20214	1.0	0.538931	31	
154	1.0	0.490340	116	
22290	1.0	0.532707	3	
35561	1.0	0.443415	9	
4217	1.0	0.479745	6	
25916	1.0	0.308271	117	
16145	1.0	0.451767	1	
5679	1.0	0.411249	20	
16618	1.0	0.534433	2	
29343	1.0	0.279221	2	
8781	1.0	0.279126	3	
10999	1.0	0.419232	1	
18536	1.0	0.291889	28	
12431	1.0	0.398686	7	
19276	1.0	0.318302	46	

	num_self_hrefs	num_imgs	num_videos	average_token_length	…	\
article_id					…	
28698	0	1	0	4.765102	…	
36127	3	1	0	4.553931	…	
21810	10	1	1	4.666280	…	
6441	0	1	0	4.580236	…	
27810	0	1	0	4.746181	…	
20214	1	13	0	4.634469	…	
154	3	50	1	4.315463	…	
22290	2	5	0	4.430695	…	
35561	1	7	13	4.519631	…	
4217	0	1	0	4.748667	…	
25916	4	1	0	4.874473	…	
16145	1	6	1	4.699982	…	
5679	13	7	0	4.395681	…	
16618	1	101	2	4.807379	…	
29343	2	100	1	4.241931	…	
8781	3	100	1	4.241825	…	
10999	1	1	0	4.512781	…	
18536	4	108	0	4.253069	…	
12431	3	1	1	3.935858	…	
19276	17	111	2	4.281921	…	

	avg_negative_polarity	min_negative_polarity \
article_id		
28698	-0.219591	-1.0
36127	-0.235841	-1.0
21810	-0.353855	-1.0
6441	-0.269213	-1.0
27810	-0.282290	-0.9
20214	-0.223446	-1.0
154	-0.310585	-1.0
22290	-0.258193	-1.0
35561	-0.404658	-1.0
4217	-0.266902	-1.0
25916	-0.193521	-0.8
16145	-0.392198	-1.0
5679	-0.317200	-1.0
16618	-0.282952	-1.0
29343	-0.389866	-1.0
8781	-0.388173	-1.0
10999	-0.252184	-0.8
18536	-0.373817	-1.0
12431	-0.284537	-1.0
19276	-0.359945	-1.0

	max_negative_polarity	title_subjectivity \
article_id		
28698	-0.012500	0.062500
36127	-0.050000	0.266667
21810	-0.050000	0.000000
6441	-0.050000	0.600000
27810	-0.050000	0.000000
20214	-0.033333	0.000000
154	-0.050000	0.622222
22290	-0.050000	0.000000
35561	-0.075000	0.850000
4217	-0.050000	0.454545
25916	-0.050000	0.000000
16145	-0.050000	0.500000
5679	-0.050000	0.000000
16618	-0.050000	0.000000
29343	-0.050000	1.000000
8781	-0.050000	0.600000
10999	-0.025000	0.000000
18536	-0.025000	0.000000
12431	-0.025000	0.416667
19276	-0.050000	0.535714

article_id	title_sentiment_polarity	abs_title_subjectivity
28698	0.000000	0.437500
36127	0.066667	0.233333
21810	0.000000	0.500000
6441	0.700000	0.100000
27810	0.000000	0.500000
20214	0.000000	0.500000
154	0.044444	0.122222
22290	0.000000	0.500000
35561	-0.300000	0.350000
4217	0.136364	0.045455
25916	0.000000	0.500000
16145	0.250000	0.000000
5679	0.000000	0.500000
16618	0.000000	0.500000
29343	0.850000	0.500000
8781	0.475000	0.100000
10999	0.000000	0.500000
18536	0.000000	0.500000
12431	0.375000	0.083333
19276	0.285714	0.035714

article_id	abs_title_sentiment_polarity	is_popular	d_mon_tues	d_no_words
28698	0.000000	0	0	0
36127	0.066667	0	0	0
21810	0.000000	0	0	0
6441	0.700000	1	1	0
27810	0.000000	0	0	0
20214	0.000000	0	0	0
154	0.044444	0	0	0
22290	0.000000	0	1	0
35561	0.300000	0	0	0
4217	0.136364	0	0	0
25916	0.000000	0	1	0
16145	0.250000	0	1	0
5679	0.000000	0	0	0
16618	0.000000	0	0	0
29343	0.850000	1	1	0
8781	0.475000	0	0	0
10999	0.000000	0	1	0
18536	0.000000	0	0	0
12431	0.375000	0	1	0
19276	0.285714	1	0	0

[20 rows x 54 columns]

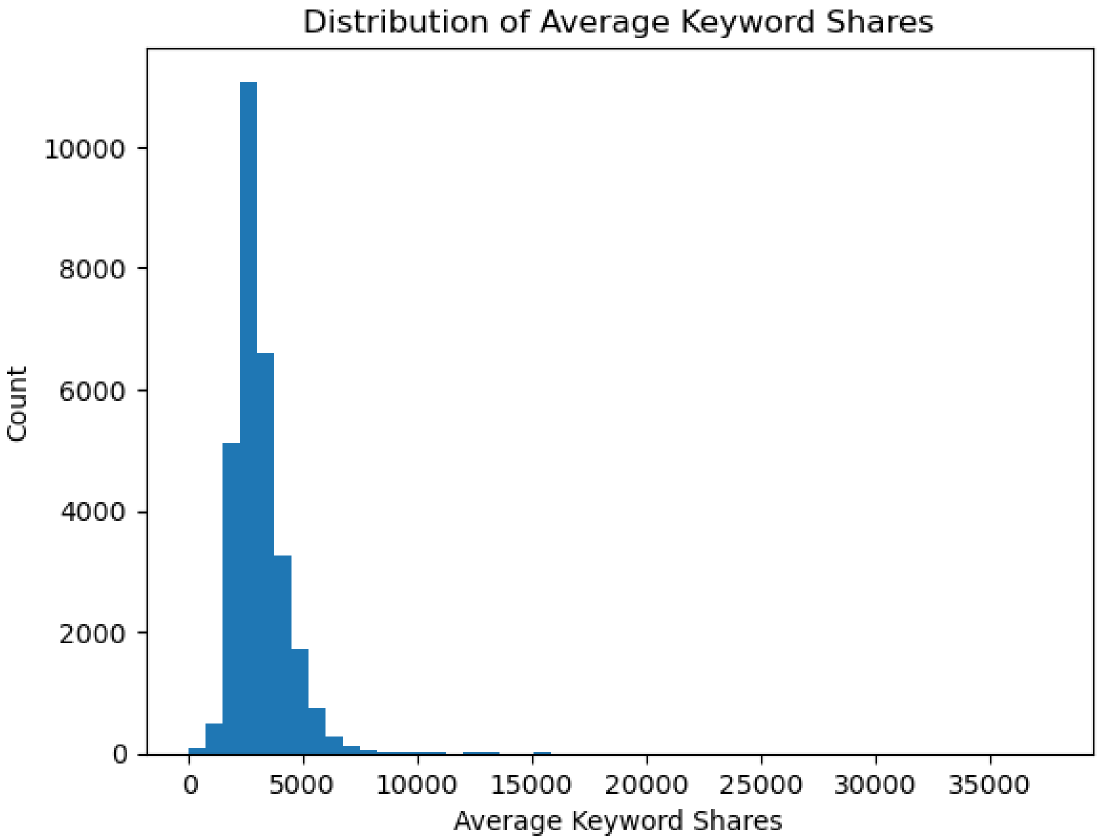
[12]:
```python
# Viewing overall distribution of articles denoted as popular vs not popular
data['is_popular'].value_counts()
```

[12]:
```
is_popular
0    26116
1     3616
Name: count, dtype: int64
```

From reviewing the descriptive statistics and especially the mean values of `is_popular` for the different quantiles it appears that content length is significant for determining popularity and therefore will not be curtailed/processed to adjust for potential extreme values. When sorting the dataframe by `n_tokens_content` and viewing the 20 largest observations the dropoff is not significantly dramatic enough to warrant potentially dropping those data points. It might also hamper the prediction power of the models.

Now the distribution of `kw_avg_avg` will be checked.

[13]:
```python
# Plottig the distribution of kw_avg_avg
plt.hist(data['kw_avg_avg'], bins=50)

plt.xlabel('Average Keyword Shares')
plt.ylabel('Count')
plt.title('Distribution of Average Keyword Shares')

plt.show()
```



Similar to `n_tokens_content` we have a distribution that is somewhat normal but has a very long right tail.

1.2 Feature Engineering

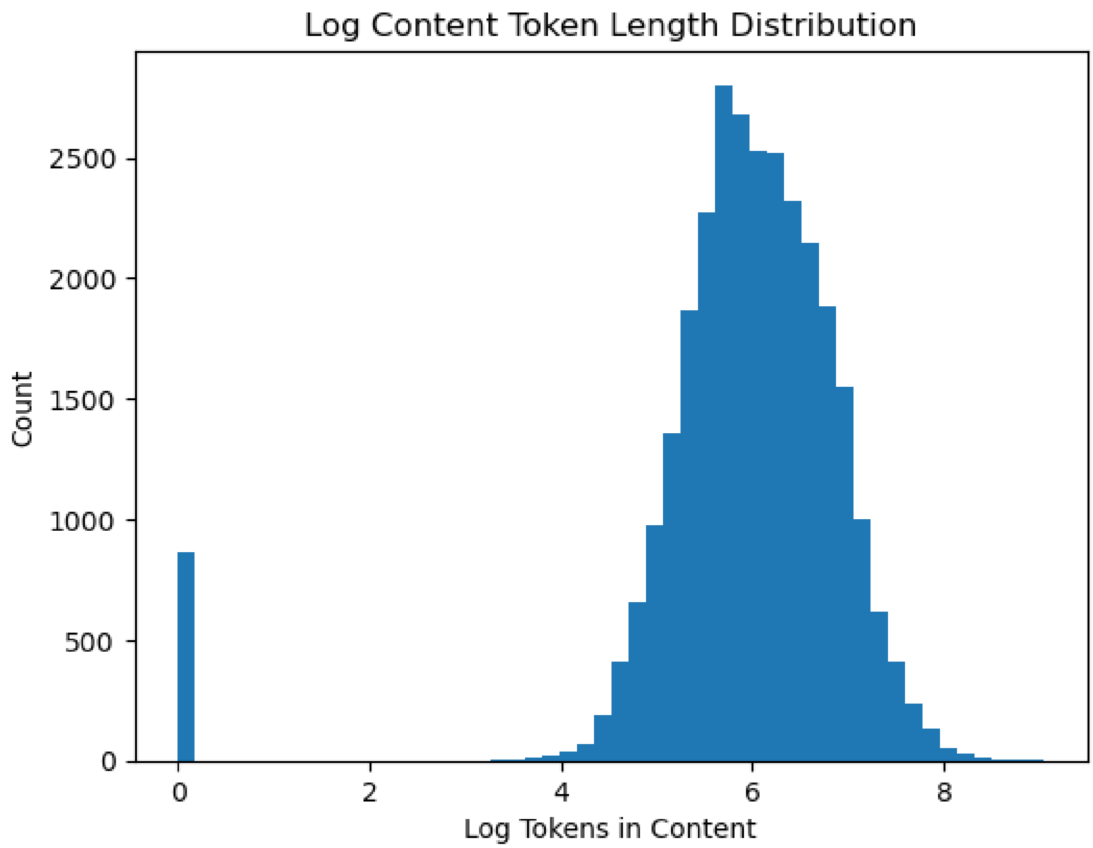
Some feature engineering will be performed on the data to account for any non-linearity in the data as well as potentially capture additional information that is not being captured by the base data points. All feature engineered variables will begin with the `f_` tag.

One such thing would be examine the distribution of log values for `n_tokens_content` and `kw_avg_avg` to see if it creates a more normal distribution along with taking quadratics of certain variables to capture potential non-linearity in the data, as mentioned above.

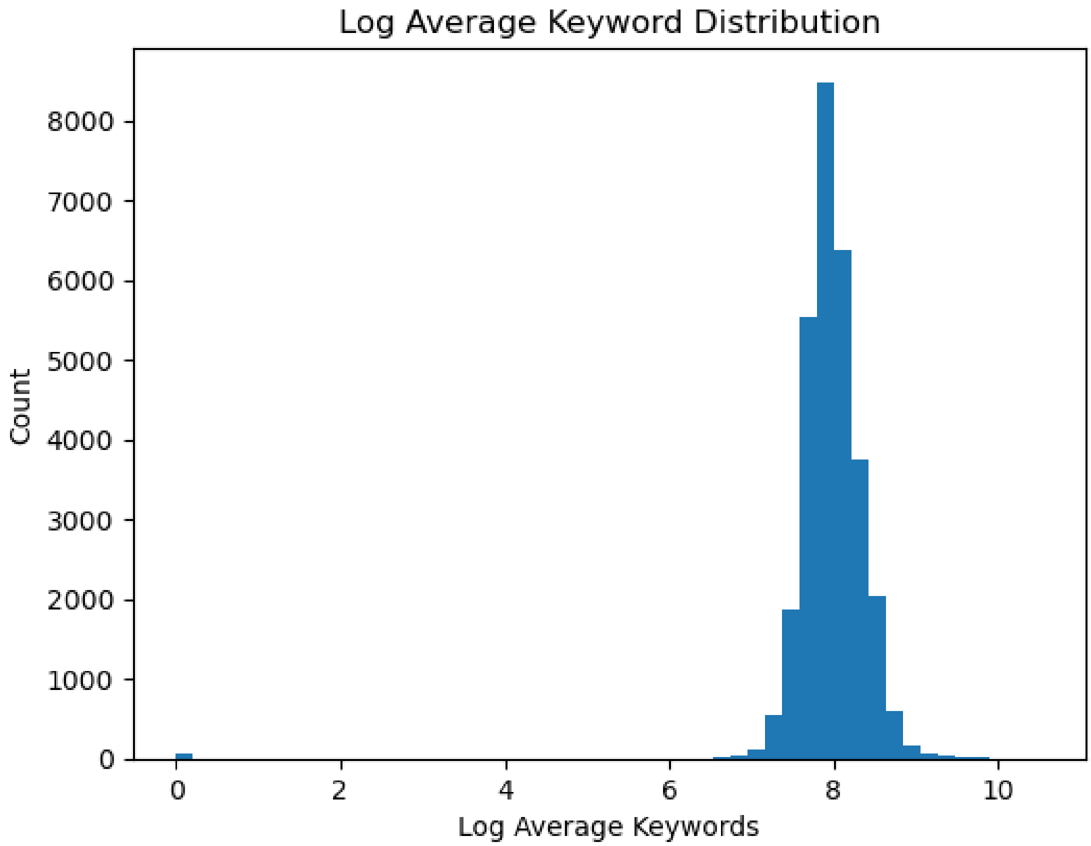
[14]:
```
# Checking the distribution plot for the log values of n_tokens_content,␣
↪setting the values to 0 where
plt.hist(np.where(data['n_tokens_content'] == 0, 0, np.
↪log(data['n_tokens_content'])), bins=50)

plt.xlabel('Log Tokens in Content')
plt.ylabel('Count')
plt.title('Log Content Token Length Distribution')
```

```
plt.show()
```

```
C:\Users\Xrona\anaconda3\Lib\site-packages\pandas\core\arraylike.py:396:
RuntimeWarning: divide by zero encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```



[15]:
```
# Performing the same task for kw_avg_avg
plt.hist(np.where(data['kw_avg_avg'] == 0, 0, np.log(data['kw_avg_avg'])),␣
 ↪bins=50)

plt.xlabel('Log Average Keywords')
plt.ylabel('Count')
plt.title('Log Average Keyword Distribution')

plt.show()
```



For occurrences where the number of tokens was equal to 0 the value of 0 was imputted for the log. The warnings generated are not actual issues. The log transformations of `n_tokens_content` and `kw_avg_avg` create more normal distributions and therefore will be used.

[16]:
```
# Creating the log values as specified
data['f_log_tokens_content'] = np.where(data['n_tokens_content'] == 0, 0, np.
 ↪log(data['n_tokens_content']))
data['f_log_kw_avg_avg'] = np.where(data['kw_avg_avg'] == 0, 0, np.
 ↪log(data['kw_avg_avg']))
```

```
C:\Users\Xrona\anaconda3\Lib\site-packages\pandas\core\arraylike.py:396:
RuntimeWarning: divide by zero encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

Now that log values have been taken, the values of all the features with a standard deviation above 10 (consdiered to be large, also to avoid using rate/ratio features) will be winsorized at the top end in order to mitigate the effect of the extreme values on the data without completely discarding them. The 99% upper bounds will be used as the content at the lower bounds for all columns is significant given that they are likely only videos and/or image slideshows.

[17]:

```
# Importing required method
from scipy.stats.mstats import winsorize

# Defining a function for winsorizing
def winsorize_columns(data, columns):
    data_copy = data.copy()
    for column in columns:
        if column in data_copy.columns:
            data_copy[column] = winsorize(data_copy[column], limits=(0.00, 0.
 ↪01))
    return data_copy

# Setting columns to winsorize to a variable
to_winsorize = [col for col in data.columns if data[col].std() > 10]

# Running function on columns
data = winsorize_columns(data,to_winsorize)
```

Lastly, additional features will be engineered post-winsorization to try to account for any non-linearity in the data.

[18]:

```
# Creating quadratic variables
data['f_n_tokens_title_sq'] = np.power(data['n_tokens_title'], 2)
data['f_num_imgs_sq'] = np.power(data['num_imgs'], 2)
data['f_num_keywords'] = np.power(data['num_keywords'], 2)

# Creating ratio variables
data['f_global_pos_neg_word_ratio'] = np.
 ↪where((data['global_rate_positive_words'] > 0) &␣
 ↪(data['global_rate_negative_words'] == 0), 1,  # Set ratio to 1 when pos␣
 ↪words > 0 and neg words = 0
                                          np.
 ↪where((data['global_rate_positive_words'] == 0) &␣
 ↪(data['global_rate_negative_words'] == 0), 0,  # Set ratio to 0 when pos␣
 ↪words = 0 and neg words = 0
                                                         ␣
 ↪data['global_rate_positive_words'] / data['global_rate_negative_words']))  #␣
 ↪Calculate ratio normally
data['f_token_img_ratio'] = np.where((data['n_tokens_content'] > 0) &␣
 ↪(data['num_imgs'] == 0), 1,  # Set ratio to 1 when n_tokens_content > 0 and␣
 ↪num_imgs = 0
                              np.where((data['n_tokens_content'] == 0) &␣
 ↪(data['num_imgs'] == 0), 0,  # Set ratio to 0 when n_tokens_content = 0 and␣
 ↪num_imgs = 0
                                   data['n_tokens_content'] /␣
 ↪data['num_imgs']))  # Calculate ratio normally
```

```
data['f_token_video_ratio'] = np.where((data['n_tokens_content'] > 0) &␣
 ↪(data['num_videos'] == 0), 1,  # Set ratio to 1 when n_tokens_content > 0␣
 ↪and num_videos = 0
                                   np.where((data['n_tokens_content'] == 0) &␣
 ↪(data['num_videos'] == 0), 0,  # Set ratio to 0 when n_tokens_content = 0␣
 ↪and num_videos = 0
                                        data['n_tokens_content'] /␣
 ↪data['num_videos']))  # Calculate ratio normally

# Creating additional feature engineered variables
data['f_avg_polarity'] = data['avg_positive_polarity'] +␣
 ↪data['avg_negative_polarity'] # Showing the overall average polarity of the␣
 ↪article
```

1.3 Training, Validation, and Test Splits

The training and test splits for creating models will be created using `is_popular` as the target variable. A 80/20% split will be made of the data to create a training and test split for OLS and machine learing models like Random Forest and Gradient Boosting models. A pseudo-random seed will be generated using the due date for the Kaggle competition as the seed number (20240419).

For the train_test_split the `stratify` parameter will be used in order to preserve the the distribution of `is_popular` given that it number of popular articles is inbalanced in the dataset.

[19]:
```
# Importing required function
from sklearn.model_selection import train_test_split

# Setting the pseudo random seed
prng = np.random.RandomState(20240419)

# Creating X and y variables for splitting
X = data.drop(columns=['is_popular'])
y = data['is_popular']

# Creating initial training/test split of data, using stratify to preserve␣
 ↪distribution of is_popular
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2,␣
 ↪stratify=y, random_state=prng)
```

1.4 Benchmark and Base Feature OLS Models

The benchmark and OLS models will be created below.

1.4.1 Benchmark Model

A simple benchmark model simply using the mean value of `is_popular` will be used.

[20]:
```
# Importing required function
from sklearn.metrics import roc_auc_score

# Creating benchmark model
benchmark = np.mean(y_train)

# Calculating the AUC scores for the benchmark model
train_auc = roc_auc_score(y_train, np.repeat(benchmark, len(y_train)))
test_auc = roc_auc_score(y_test, np.repeat(benchmark, len(y_test)))
benchmark_pred = ['Benchmark', train_auc, test_auc]

# Storing and displaying results in a dataframe
results = pd.DataFrame([benchmark_pred], columns = ['Model', 'Train AUC', 'Test␣
↪AUC'])
results
```

[20]:

	Model	Train AUC	Test AUC
0	Benchmark	0.5	0.5

The results of the benchmark model is .5 forboth the training and test sets. This means that it doesn't do any better than predicting the outcome randomly. This makes it a perfect benchmark model.

1.4.2 Single Feature Logit

This model will only include a single feature and will be used as the benchmark going forward. The feature selected will be `d_weekend` as the day of the week seems to be a significant predictor for popularity. ll predictions are made for probabilities. This means using `predict_proba()` instead of `predict()`. Grabbing the `[:,1]` element of the probability is to denote the prediction of the probability that the article is popular.

[21]:
```
# Importing required functions
from sklearn.linear_model import LogisticRegression

# Creating a logistic regression model and fitting it to the data
logit_single = LogisticRegression().fit(X_train[['kw_avg_avg']], y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, logit_single.
↪predict_proba(X_train[['kw_avg_avg']])[:,1])
test_auc = roc_auc_score(y_test, logit_single.
↪predict_proba(X_test[['kw_avg_avg']])[:,1])
logit_single_pred = ['Single Feature Logit', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = logit_single_pred
results
```

[21]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424

The single feature logit model is doing a much better job than the benchmark. It appears that using a single feature `kw_avg_avg` is a decent predictor and is better than a random prediction of popularity. `kw_avg_avg` is the average shares of the average keyword in the article and it looks like this is a decent predictor to use.

1.4.3 Logit Using All Base Features

[22]:
```
# Creating list of base features for model selection from the dataset
base_features = [col for col in X_train.columns if col not in exclude_cols and␣
↪not col.startswith('f_')]

# Creating and fitting the model, setting max iterations to 3000 to allow␣
↪convergence
logit_base = LogisticRegression(max_iter=3000).fit(X_train[base_features],␣
↪y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, logit_base.
↪predict_proba(X_train[base_features])[:,1])
test_auc = roc_auc_score(y_test, logit_base.
↪predict_proba(X_test[base_features])[:,1])
logit_base_pred = ['Logit Base Features', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = logit_base_pred
results
```

[22]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154

Using all the base features does a worse job than only 1 feature. It seems like some of the features in the dataset are adding noise to the model and making a worse prediction.

1.4.4 LASSO All Base Features

For the LASSO version of the model a standard scaler will need to be applied so that the LASSO doesn't have a bias towards features with larger standard deviations.

[23]:
```
# Importing required method
from sklearn.preprocessing import StandardScaler

# Initializing a StandardScaler
```

```
scaler = StandardScaler()

# Applying the standard scaler
X_train_base_scaled = scaler.fit_transform(X_train[base_features])
X_test_base_scaled = scaler.transform(X_test[base_features])

# Creating the LASSO model
lasso_base = LogisticRegression(penalty='l1', solver='liblinear',␣
 ↪max_iter=3000).fit(X_train_base_scaled, y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, lasso_base.
 ↪predict_proba(X_train_base_scaled)[:,1])
test_auc = roc_auc_score(y_test, lasso_base.predict_proba(X_test_base_scaled)[:
 ↪,1])
lasso_base_pred = ['LASSO Base Features', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = lasso_base_pred
results
```

[23]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694

The LASSO on the base features seems to have corrected for some of the noise in the base features because it produced a better AUC score on the test set than the single feature logit. The L1 penalty has reduced the coefficients of features that are less meaningful and more noisy and produced a better result.

1.4.5 Logit with Interaction Terms on Base Features

First order interactions only will be used (no quadratics) on only base features.

[24]:

```
# Importing required method
from sklearn.preprocessing import PolynomialFeatures

# Creating interaction terms in training data
interactions = PolynomialFeatures(degree=1, interaction_only=True)
X_train_base_interactions = interactions.fit_transform(X_train[base_features])

# Fitting Logit model to training data with interaction terms
logit_base_interactions = LogisticRegression(max_iter=3000).
 ↪fit(X_train_base_interactions, y_train)

# Creating predictions and calculating AUC score
```

```
train_auc = roc_auc_score(y_train, logit_base_interactions.
 ↪predict_proba(X_train_base_interactions)[:,1])
test_auc = roc_auc_score(y_test, logit_base_interactions.
 ↪predict_proba(interactions.transform(X_test[base_features]))[:,1])
logit_base_interactions_pred = ['Logit Base Interactions', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = logit_base_interactions_pred
results
```

[24]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694
4	Logit Base Interactions	0.654632	0.652614

First order interaction terms on the base features gave virtually the same score as the Logit base features model. Adding interaction terms does not seem to be affecting the model at all.

1.4.6 LASSO Version of Previous Model

[25]:
```
# Applying the standard scaler to the data to the interactions
X_train_base_int_scaled = scaler.fit_transform(X_train_base_interactions)
X_test_base_int_scaled = scaler.transform(interactions.
 ↪transform(X_test[base_features]))

# Tweaking the previous model to turn it into a LASSO
lasso_base_interactions = LogisticRegression(penalty='l1', solver='liblinear',␣
 ↪max_iter=3000).fit(X_train_base_int_scaled, y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, lasso_base_interactions.
 ↪predict_proba(X_train_base_int_scaled)[:,1])
test_auc = roc_auc_score(y_test, lasso_base_interactions.
 ↪predict_proba(X_test_base_int_scaled)[:,1])
lasso_base_interactions_pred = ['LASSO Base Interactions', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = lasso_base_interactions_pred
results
```

[25]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694

```
4  Logit Base Interactions   0.654632  0.652614
5  LASSO Base Interactions   0.697954  0.688693
```

Adding interaction terms very marginally decreased the AUC score on the test set from the LASSO model without interaction terms. This further shows that adding interaction terms to the model increases complexity with no actual gain. Going forward the interaction terms will not be used.

1.5 Base Feature Machine Learing Models

Below Machine Learing models using exclusively base features will be created. Since there are no categorical variables in the training/test data being used there will be no need for a pipeline

1.5.1 Random Forest Base Features

Creating a Random Forest model with a minimum sample split of 30 to prevent overfitting.

[26]:
```
# Importing required method
from sklearn.ensemble import RandomForestClassifier

# Creating and fitting the RF to training data using prng defined earlier.␣
↪Parameters set to prevent overfitting
rf_base = RandomForestClassifier(min_samples_split=30, random_state=prng).
↪fit(X_train[base_features], y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, rf_base.
↪predict_proba(X_train[base_features])[:,1])
test_auc = roc_auc_score(y_test, rf_base.predict_proba(X_test[base_features])[:
↪,1])
rf_base_pred = ['RF Base Features', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = rf_base_pred
results
```

[26]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694
4	Logit Base Interactions	0.654632	0.652614
5	LASSO Base Interactions	0.697954	0.688693
6	RF Base Features	0.990398	0.701710

From the scores we can see the RF model performs almost perfectly on the training set. This is not usually desirable as it likely means that the model is overfitting to the training set. From the test AUC scores the RF model has performed the best so far but is a clear dropoff from the training AUC score, meaning that the RF model is indeed overfitting despite some effort to prevent it. Despite this, so far the RF model is doing the best job of predicting article popularity.

1.5.2 Gradient Boosting Machine Base Features

As with the RF model, `min_samples_split` set to 30 to avoid overfitting.

[27]:
```
# Importing required method
from sklearn.ensemble import GradientBoostingClassifier

# Creating and fitting GBM to training
gbm_base = GradientBoostingClassifier(min_samples_split=30, random_state=prng).
 ↪fit(X_train[base_features], y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, gbm_base.
 ↪predict_proba(X_train[base_features])[:,1])
test_auc = roc_auc_score(y_test, gbm_base.predict_proba(X_test[base_features])[:
 ↪,1])
gbm_base_pred = ['GBM Base Features', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = gbm_base_pred
results
```

[27]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694
4	Logit Base Interactions	0.654632	0.652614
5	LASSO Base Interactions	0.697954	0.688693
6	RF Base Features	0.990398	0.701710
7	GBM Base Features	0.760346	0.702428

From the scores the GBM model has a more appropriate train AUC score than the RF model and also performs very slightly better on the Test AUC. The GBM model is doing a better job of regularization on the training set which is resulting in a (marginally) better test AUC score.

1.5.3 XGBoost Base Features

A different boosting model will be attempted.

[28]:
```
# Importing required method
from xgboost import XGBClassifier

# Creating and fitting XGB to training set
xgb_base = XGBClassifier().fit(X_train[base_features], y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, xgb_base.
 ↪predict_proba(X_train[base_features])[:,1])
```

```
test_auc = roc_auc_score(y_test, xgb_base.predict_proba(X_test[base_features])[:
 ↪,1])
xgb_base_pred = ['XGB Base Features', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = xgb_base_pred
results
```

[28]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694
4	Logit Base Interactions	0.654632	0.652614
5	LASSO Base Interactions	0.697954	0.688693
6	RF Base Features	0.990398	0.701710
7	GBM Base Features	0.760346	0.702428
8	XGB Base Features	0.989411	0.667010

From these results it appears that the XGBoost model performs similarly to the RF model except with a worse test AUC score which means it does a worse job of predicting popularity of articles than either of the two machine learning models. It is doing worse than the linear models as well.

1.6 Base Feature Fully Connected Neural Network Models

Below some neural network models will be created using only base features. A sigmoid layer will be used as the output layer for all models given that the classification task is for a binary target variable. Prior to creating any neural network models, however, a new split of the data will need to be done to create a validation set. Along with this, the data will need to be scaled using `StandardScaler()` in order to regularize the data to make the neural network models function better.

[29]:
```
# Creating new training and validation splits from X_train
nnX_train, nnX_val, nny_train, nny_val = train_test_split(X_train, y_train,␣
 ↪test_size=.2, stratify=y_train, random_state=prng)

# Applying StandardScaler base features of these new splits along with X_test
nnX_train_base_scaled = scaler.fit_transform(nnX_train[base_features])
nnX_val_base_scaled = scaler.transform(nnX_val[base_features])
nnX_test_base_scaled = scaler.transform(X_test[base_features])
```

1.6.1 Single RelU Hidden Layer

[30]:
```
# Importing required methods
from keras.utils import set_random_seed
from keras.models import Sequential
from keras.layers import Input, Dense
```

```python
# Setting the random seed for all neural network models for reproducibility,␣
↪using same seed number as above
set_random_seed(20240419)

# Creating the model
relu_hidden = Sequential([
    Input(shape=nnX_train_base_scaled.shape[1:]),
    Dense(100, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compiling the model
relu_hidden.compile(loss='binary_crossentropy', optimizer='adam',␣
↪metrics=['accuracy'])

# Fitting the model
relu_hidden.fit(nnX_train_base_scaled, nny_train,␣
↪validation_data=(nnX_val_base_scaled, nny_val), epochs=20, batch_size=128)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(nny_train, relu_hidden.predict(nnX_train_base_scaled))
test_auc = roc_auc_score(y_test, relu_hidden.predict(nnX_test_base_scaled))
relu_hidden_pred = ['RelU Hidden Base Features', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = relu_hidden_pred
results
```

```
Epoch 1/20
149/149              1s 2ms/step -
accuracy: 0.8346 - loss: 0.4470 - val_accuracy: 0.8777 - val_loss: 0.3494
Epoch 2/20
149/149              0s 1ms/step -
accuracy: 0.8760 - loss: 0.3507 - val_accuracy: 0.8783 - val_loss: 0.3471
Epoch 3/20
149/149              0s 1ms/step -
accuracy: 0.8759 - loss: 0.3459 - val_accuracy: 0.8779 - val_loss: 0.3459
Epoch 4/20
149/149              0s 1ms/step -
accuracy: 0.8758 - loss: 0.3428 - val_accuracy: 0.8777 - val_loss: 0.3453
Epoch 5/20
149/149              0s 1ms/step -
accuracy: 0.8757 - loss: 0.3406 - val_accuracy: 0.8781 - val_loss: 0.3449
Epoch 6/20
149/149              0s 1ms/step -
accuracy: 0.8758 - loss: 0.3388 - val_accuracy: 0.8789 - val_loss: 0.3447
Epoch 7/20
```

```
149/149              0s 1ms/step -
accuracy: 0.8760 - loss: 0.3372 - val_accuracy: 0.8787 - val_loss: 0.3447
Epoch 8/20
149/149              0s 1ms/step -
accuracy: 0.8762 - loss: 0.3359 - val_accuracy: 0.8785 - val_loss: 0.3447
Epoch 9/20
149/149              0s 988us/step -
accuracy: 0.8768 - loss: 0.3346 - val_accuracy: 0.8787 - val_loss: 0.3448
Epoch 10/20
149/149              0s 991us/step -
accuracy: 0.8769 - loss: 0.3335 - val_accuracy: 0.8787 - val_loss: 0.3451
Epoch 11/20
149/149              0s 1ms/step -
accuracy: 0.8768 - loss: 0.3325 - val_accuracy: 0.8789 - val_loss: 0.3453
Epoch 12/20
149/149              0s 1ms/step -
accuracy: 0.8766 - loss: 0.3316 - val_accuracy: 0.8787 - val_loss: 0.3456
Epoch 13/20
149/149              0s 970us/step -
accuracy: 0.8771 - loss: 0.3307 - val_accuracy: 0.8787 - val_loss: 0.3459
Epoch 14/20
149/149              0s 1ms/step -
accuracy: 0.8769 - loss: 0.3299 - val_accuracy: 0.8779 - val_loss: 0.3462
Epoch 15/20
149/149              0s 961us/step -
accuracy: 0.8773 - loss: 0.3290 - val_accuracy: 0.8777 - val_loss: 0.3466
Epoch 16/20
149/149              0s 996us/step -
accuracy: 0.8774 - loss: 0.3282 - val_accuracy: 0.8772 - val_loss: 0.3470
Epoch 17/20
149/149              0s 1ms/step -
accuracy: 0.8774 - loss: 0.3275 - val_accuracy: 0.8770 - val_loss: 0.3473
Epoch 18/20
149/149              0s 994us/step -
accuracy: 0.8774 - loss: 0.3268 - val_accuracy: 0.8770 - val_loss: 0.3477
Epoch 19/20
149/149              0s 971us/step -
accuracy: 0.8779 - loss: 0.3261 - val_accuracy: 0.8772 - val_loss: 0.3481
Epoch 20/20
149/149              0s 985us/step -
accuracy: 0.8781 - loss: 0.3254 - val_accuracy: 0.8774 - val_loss: 0.3485
595/595              0s 537us/step
186/186              0s 577us/step
```

[30]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424

```
2        Logit Base Features   0.660114  0.659154
3        LASSO Base Features   0.697954  0.688694
4    Logit Base Interactions   0.654632  0.652614
5    LASSO Base Interactions   0.697954  0.688693
6           RF Base Features   0.990398  0.701710
7          GBM Base Features   0.760346  0.702428
8          XGB Base Features   0.989411  0.667010
9  RelU Hidden Base Features   0.769597  0.689875
```

This initial neural network model performed very well, only doing worse than the GBM model. The preprocessing done with a standard scaler made the data easier for the neural network model to handle and it did a good job of predicting popularity.

1.6.2 Two Hidden RelU Layers, 256 Nodes Each

[31]:
```
# Creating the model
two_relu_256 = Sequential([
    Input(shape=nnX_train_base_scaled.shape[1:]),
    Dense(256, activation='relu'),
    Dense(256, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compiling the model
two_relu_256.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])

# Fitting the model
two_relu_256.fit(nnX_train_base_scaled, nny_train,␣
 ↪validation_data=(nnX_val_base_scaled, nny_val), epochs=20, batch_size=128)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(nny_train, two_relu_256.
 ↪predict(nnX_train_base_scaled))
test_auc = roc_auc_score(y_test, two_relu_256.predict(nnX_test_base_scaled))
two_relu_256_pred = ['Two Relu 256 Hidden Base Features', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = two_relu_256_pred
results
```

```
Epoch 1/20
149/149              1s 2ms/step -
accuracy: 0.8609 - loss: 0.3907 - val_accuracy: 0.8783 - val_loss: 0.3479
Epoch 2/20
149/149              0s 1ms/step -
accuracy: 0.8759 - loss: 0.3436 - val_accuracy: 0.8785 - val_loss: 0.3474
Epoch 3/20
```

```
149/149            0s 1ms/step -
accuracy: 0.8763 - loss: 0.3386 - val_accuracy: 0.8783 - val_loss: 0.3477
Epoch 4/20
149/149            0s 2ms/step -
accuracy: 0.8767 - loss: 0.3346 - val_accuracy: 0.8787 - val_loss: 0.3480
Epoch 5/20
149/149            0s 1ms/step -
accuracy: 0.8767 - loss: 0.3304 - val_accuracy: 0.8787 - val_loss: 0.3487
Epoch 6/20
149/149            0s 1ms/step -
accuracy: 0.8774 - loss: 0.3264 - val_accuracy: 0.8793 - val_loss: 0.3504
Epoch 7/20
149/149            0s 1ms/step -
accuracy: 0.8782 - loss: 0.3223 - val_accuracy: 0.8787 - val_loss: 0.3521
Epoch 8/20
149/149            0s 1ms/step -
accuracy: 0.8796 - loss: 0.3176 - val_accuracy: 0.8785 - val_loss: 0.3544
Epoch 9/20
149/149            0s 1ms/step -
accuracy: 0.8807 - loss: 0.3122 - val_accuracy: 0.8783 - val_loss: 0.3577
Epoch 10/20
149/149            0s 1ms/step -
accuracy: 0.8820 - loss: 0.3063 - val_accuracy: 0.8779 - val_loss: 0.3606
Epoch 11/20
149/149            0s 1ms/step -
accuracy: 0.8848 - loss: 0.2994 - val_accuracy: 0.8779 - val_loss: 0.3644
Epoch 12/20
149/149            0s 1ms/step -
accuracy: 0.8876 - loss: 0.2918 - val_accuracy: 0.8774 - val_loss: 0.3697
Epoch 13/20
149/149            0s 1ms/step -
accuracy: 0.8906 - loss: 0.2836 - val_accuracy: 0.8777 - val_loss: 0.3761
Epoch 14/20
149/149            0s 1ms/step -
accuracy: 0.8942 - loss: 0.2748 - val_accuracy: 0.8760 - val_loss: 0.3830
Epoch 15/20
149/149            0s 1ms/step -
accuracy: 0.8976 - loss: 0.2656 - val_accuracy: 0.8756 - val_loss: 0.3901
Epoch 16/20
149/149            0s 1ms/step -
accuracy: 0.9022 - loss: 0.2562 - val_accuracy: 0.8734 - val_loss: 0.3964
Epoch 17/20
149/149            0s 1ms/step -
accuracy: 0.9064 - loss: 0.2460 - val_accuracy: 0.8716 - val_loss: 0.4039
Epoch 18/20
149/149            0s 1ms/step -
accuracy: 0.9110 - loss: 0.2362 - val_accuracy: 0.8707 - val_loss: 0.4136
Epoch 19/20
```

```
149/149              0s 1ms/step -
accuracy: 0.9151 - loss: 0.2259 - val_accuracy: 0.8688 - val_loss: 0.4238
Epoch 20/20
149/149              0s 2ms/step -
accuracy: 0.9177 - loss: 0.2158 - val_accuracy: 0.8695 - val_loss: 0.4351
595/595              0s 697us/step
186/186              0s 713us/step
```

[31]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694
4	Logit Base Interactions	0.654632	0.652614
5	LASSO Base Interactions	0.697954	0.688693
6	RF Base Features	0.990398	0.701710
7	GBM Base Features	0.760346	0.702428
8	XGB Base Features	0.989411	0.667010
9	RelU Hidden Base Features	0.769597	0.689875
10	Two Relu 256 Hidden Base Features	0.934741	0.653588

In an attempt to make the model much more complex in order to better capture the data, it appears to have backfired by providing a significantly worse AUC score than the initial neural network model. Adding more complexity did not help in this instance and caused an overfit. A version of the initial model with fewer RelU nodes will be created next.

1.6.3 Simpler Hidden Layer, Fewer RelU Nodes

[32]:

```
# Creating the model
relu_base_50 = Sequential([
    Input(shape=nnX_train_base_scaled.shape[1:]),
    Dense(50, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compiling the model
relu_base_50.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])

# Fitting the model
relu_base_50.fit(nnX_train_base_scaled, nny_train,␣
 ↪validation_data=(nnX_val_base_scaled, nny_val), epochs=20, batch_size=128)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(nny_train, relu_base_50.
 ↪predict(nnX_train_base_scaled))
test_auc = roc_auc_score(y_test, relu_base_50.predict(nnX_test_base_scaled))
relu_base_50_pred = ['RelU 50 Base Features', train_auc, test_auc]
```

```python
# Adding to results
results.loc[len(results)] = relu_base_50_pred
results
```

```
Epoch 1/20
149/149            1s 2ms/step -
accuracy: 0.6757 - loss: 0.6130 - val_accuracy: 0.8753 - val_loss: 0.3611
Epoch 2/20
149/149            0s 1000us/step -
accuracy: 0.8750 - loss: 0.3579 - val_accuracy: 0.8762 - val_loss: 0.3550
Epoch 3/20
149/149            0s 979us/step -
accuracy: 0.8755 - loss: 0.3512 - val_accuracy: 0.8764 - val_loss: 0.3524
Epoch 4/20
149/149            0s 1ms/step -
accuracy: 0.8761 - loss: 0.3473 - val_accuracy: 0.8766 - val_loss: 0.3507
Epoch 5/20
149/149            0s 981us/step -
accuracy: 0.8766 - loss: 0.3447 - val_accuracy: 0.8768 - val_loss: 0.3495
Epoch 6/20
149/149            0s 1ms/step -
accuracy: 0.8771 - loss: 0.3426 - val_accuracy: 0.8772 - val_loss: 0.3487
Epoch 7/20
149/149            0s 996us/step -
accuracy: 0.8770 - loss: 0.3411 - val_accuracy: 0.8774 - val_loss: 0.3481
Epoch 8/20
149/149            0s 1ms/step -
accuracy: 0.8766 - loss: 0.3398 - val_accuracy: 0.8779 - val_loss: 0.3478
Epoch 9/20
149/149            0s 995us/step -
accuracy: 0.8764 - loss: 0.3387 - val_accuracy: 0.8777 - val_loss: 0.3477
Epoch 10/20
149/149            0s 970us/step -
accuracy: 0.8764 - loss: 0.3377 - val_accuracy: 0.8774 - val_loss: 0.3476
Epoch 11/20
149/149            0s 974us/step -
accuracy: 0.8767 - loss: 0.3369 - val_accuracy: 0.8777 - val_loss: 0.3477
Epoch 12/20
149/149            0s 926us/step -
accuracy: 0.8769 - loss: 0.3360 - val_accuracy: 0.8774 - val_loss: 0.3477
Epoch 13/20
149/149            0s 941us/step -
accuracy: 0.8772 - loss: 0.3352 - val_accuracy: 0.8774 - val_loss: 0.3478
Epoch 14/20
149/149            0s 976us/step -
accuracy: 0.8770 - loss: 0.3345 - val_accuracy: 0.8770 - val_loss: 0.3479
Epoch 15/20
```

```
149/149              0s 963us/step -
accuracy: 0.8771 - loss: 0.3339 - val_accuracy: 0.8768 - val_loss: 0.3480
Epoch 16/20
149/149              0s 970us/step -
accuracy: 0.8770 - loss: 0.3333 - val_accuracy: 0.8766 - val_loss: 0.3481
Epoch 17/20
149/149              0s 930us/step -
accuracy: 0.8769 - loss: 0.3327 - val_accuracy: 0.8762 - val_loss: 0.3482
Epoch 18/20
149/149              0s 949us/step -
accuracy: 0.8770 - loss: 0.3320 - val_accuracy: 0.8764 - val_loss: 0.3483
Epoch 19/20
149/149              0s 1ms/step -
accuracy: 0.8773 - loss: 0.3314 - val_accuracy: 0.8768 - val_loss: 0.3485
Epoch 20/20
149/149              0s 972us/step -
accuracy: 0.8773 - loss: 0.3309 - val_accuracy: 0.8770 - val_loss: 0.3487
595/595              0s 561us/step
186/186              0s 567us/step
```

[32]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694
4	Logit Base Interactions	0.654632	0.652614
5	LASSO Base Interactions	0.697954	0.688693
6	RF Base Features	0.990398	0.701710
7	GBM Base Features	0.760346	0.702428
8	XGB Base Features	0.989411	0.667010
9	RelU Hidden Base Features	0.769597	0.689875
10	Two Relu 256 Hidden Base Features	0.934741	0.653588
11	RelU 50 Base Features	0.754303	0.687832

Making the model even simpler produced almost the same result but slightly worse AUC score on the test set. It may be that being too simple may not be the best option.

1.7 Feature Engineering Trained Models

Below will be models created above but trained on feature engineered data instead. Feature Engineered data will be all the features from the data but will be run off of a LASSO shortlisted variable set.

Only the better performing models will be used for the purposes of this exercise.

1.7.1 Feature Engineered LASSO

For this LASSO model cross validation (5 folds) will be used to find the best c-value tuning parameter.

[33]:
```
%%time

# Importing required method
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import LogisticRegressionCV

# Scaling the data for the LASSO
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Creating the number of folds to use with each fold being stratified to ensure␣
↪representative popular cases
k = StratifiedKFold(n_splits=5, shuffle=True, random_state=prng)

# Defining the c values for regularization testing
c = np.linspace(.01, 1, num=100)

# Creating LASSO model for FE
lasso_fe = LogisticRegressionCV(Cs=c, cv=k, scoring='roc_auc', penalty='l1',␣
↪refit=True, solver='liblinear', random_state=prng)

# Fitting the model
lasso_fe.fit(X_train_scaled, y_train)

# Creating predictions and calculating AUC score from best estimator
train_auc = roc_auc_score(y_train, lasso_fe.predict_proba(X_train_scaled)[:,1])
test_auc = roc_auc_score(y_test, lasso_fe.predict_proba(X_test_scaled)[:,1])
lasso_fe_pred = ['LASSO FE', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = lasso_fe_pred
results
```

```
CPU times: total: 12min 40s
Wall time: 12min 37s
```

[33]:

	Model	Train AUC	Test AUC
0	Benchmark	0.500000	0.500000
1	Single Feature Logit	0.664677	0.666424
2	Logit Base Features	0.660114	0.659154
3	LASSO Base Features	0.697954	0.688694
4	Logit Base Interactions	0.654632	0.652614
5	LASSO Base Interactions	0.697954	0.688693
6	RF Base Features	0.990398	0.701710
7	GBM Base Features	0.760346	0.702428
8	XGB Base Features	0.989411	0.667010
9	RelU Hidden Base Features	0.769597	0.689875

```
10  Two Relu 256 Hidden Base Features   0.934741  0.653588
11              RelU 50 Base Features   0.754303  0.687832
12                           LASSO FE   0.705755  0.693671
```

Adding the feature engineered variables only slightly improved the AUC scores from the base features LASSO model meaning that there isn't much in the way of non-linearity that is either present in the data or captured by the FE terms. The shortlisted LASSO features will grab only the features that are truly meaningful from all features in the data after feature engineering for future models.

[34]:
```python
# Getting the list of LASSO shortlisted base features
lasso_fe_coef = lasso_fe.coef_.flatten() # Get 1D array
lasso_fe_results = pd.DataFrame({'Feature': X_train.columns, 'Coefficient':␣
 ↪lasso_fe_coef})
lasso_fe_shortlist = lasso_fe_results[lasso_fe_results['Coefficient'] != 0]

# Setting names of shortlisted features to a variable
shortlist = lasso_fe_shortlist['Feature'].tolist()

# Disabling scientific notation
pd.set_option('display.float_format', lambda x: '%.7f' % x)

# Displaying shortlist dataframe
lasso_fe_shortlist
```

[34]:

	Feature	Coefficient
1	n_tokens_content	0.0560052
2	n_unique_tokens	0.0334502
5	num_hrefs	0.0876893
6	num_self_hrefs	-0.0733450
7	num_imgs	0.0229865
8	num_videos	0.0511954
9	average_token_length	-0.1181666
11	d_genre_lifestyle	0.0001054
12	d_genre_entertainment	-0.0652205
13	d_genre_bus	-0.1049861
14	d_genre_socmed	0.0424464
15	d_genre_tech	0.0498925
17	kw_min_min	0.0638567
18	kw_max_min	-0.0029080
20	kw_min_max	-0.0577423
22	kw_avg_max	-0.0688707
23	kw_min_avg	-0.0909262
24	kw_max_avg	-0.1578039
25	kw_avg_avg	0.6044885
26	self_reference_min_shares	0.0923127
27	self_reference_max_shares	0.0222521
28	self_reference_avg_sharess	0.0623748

```
29                      d_weekend    0.0723745
30                         LDA_00    0.0615257
31                         LDA_01   -0.0121815
32                         LDA_02   -0.1564867
34                         LDA_04    0.0252446
35             global_subjectivity    0.1013466
36       global_sentiment_polarity   -0.0063198
37      global_rate_positive_words   -0.0240200
42          min_positive_polarity   -0.0181382
43          max_positive_polarity    0.0052315
44          avg_negative_polarity   -0.0045366
45          min_negative_polarity   -0.0083550
47             title_subjectivity    0.0316666
48        title_sentiment_polarity    0.0568141
49          abs_title_subjectivity    0.0450969
50   abs_title_sentiment_polarity    0.0236093
51                     d_mon_tues    0.0440732
53           f_log_tokens_content   -0.0378266
54               f_log_kw_avg_avg   -0.0103197
55             f_n_tokens_title_sq    0.0164148
56                  f_num_imgs_sq    0.0077856
58    f_global_pos_neg_word_ratio   -0.0364230
59              f_token_img_ratio   -0.0644949
60            f_token_video_ratio    0.0846839
61                 f_avg_polarity   -0.0196460
```

[35]:

```
# Showing columns not included in LASSO shortlist
[col for col in X_train.columns if col not in shortlist]
```

[35]:

```
['n_tokens_title',
 'n_non_stop_words',
 'n_non_stop_unique_tokens',
 'num_keywords',
 'd_genre_world',
 'kw_avg_min',
 'kw_max_max',
 'LDA_03',
 'global_rate_negative_words',
 'rate_positive_words',
 'rate_negative_words',
 'avg_positive_polarity',
 'max_negative_polarity',
 'd_no_words',
 'f_num_keywords']
```

This shortlist will be used going forward for all models.

1.7.2 Feature Engineered RF

[36]:
```
# Creating and fitting the RF to training data using prng defined earlier.␣
↪Parameters set to prevent overfitting
rf_fe = RandomForestClassifier(min_samples_split=30, random_state=prng).
↪fit(X_train[shortlist], y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, rf_fe.predict_proba(X_train[shortlist])[:,1])
test_auc = roc_auc_score(y_test, rf_fe.predict_proba(X_test[shortlist])[:,1])
rf_fe_pred = ['RF FE', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = rf_fe_pred
results
```

[36]:

	Model	Train AUC	Test AUC
0	Benchmark	0.5000000	0.5000000
1	Single Feature Logit	0.6646775	0.6664240
2	Logit Base Features	0.6601144	0.6591540
3	LASSO Base Features	0.6979545	0.6886940
4	Logit Base Interactions	0.6546322	0.6526143
5	LASSO Base Interactions	0.6979541	0.6886932
6	RF Base Features	0.9903982	0.7017100
7	GBM Base Features	0.7603463	0.7024283
8	XGB Base Features	0.9894109	0.6670103
9	RelU Hidden Base Features	0.7695974	0.6898747
10	Two Relu 256 Hidden Base Features	0.9347406	0.6535879
11	RelU 50 Base Features	0.7543033	0.6878324
12	LASSO FE	0.7057548	0.6936707
13	RF FE	0.9909127	0.6998723

From the results the shortlisted RF model performed worse than the base features model. It appears that the shortlist may not be working as intended and that some of the patterns in the data are not being captured.

1.7.3 Feature Engineered Gradient Boosting Machine

[37]:
```
# Creating and fitting GBM to training
gbm_fe = GradientBoostingClassifier(min_samples_split=30, random_state=prng).
↪fit(X_train[shortlist], y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, gbm_fe.predict_proba(X_train[shortlist])[:
↪,1])
test_auc = roc_auc_score(y_test, gbm_fe.predict_proba(X_test[shortlist])[:,1])
gbm_fe_pred = ['GBM FE', train_auc, test_auc]
```

```
# Adding to results
results.loc[len(results)] = gbm_fe_pred
results
```

[37]:

	Model	Train AUC	Test AUC
0	Benchmark	0.5000000	0.5000000
1	Single Feature Logit	0.6646775	0.6664240
2	Logit Base Features	0.6601144	0.6591540
3	LASSO Base Features	0.6979545	0.6886940
4	Logit Base Interactions	0.6546322	0.6526143
5	LASSO Base Interactions	0.6979541	0.6886932
6	RF Base Features	0.9903982	0.7017100
7	GBM Base Features	0.7603463	0.7024283
8	XGB Base Features	0.9894109	0.6670103
9	RelU Hidden Base Features	0.7695974	0.6898747
10	Two Relu 256 Hidden Base Features	0.9347406	0.6535879
11	RelU 50 Base Features	0.7543033	0.6878324
12	LASSO FE	0.7057548	0.6936707
13	RF FE	0.9909127	0.6998723
14	GBM FE	0.7715692	0.7093004

For the GBM model it looks like the shortlist does actually help the AUC score. It looks like with the shortlist then the RF model is overfitting whereas the GBM model is doing a better job of not overfitting.

1.7.4 Feature Engineered Single Hidden RelU Layer

[38]:

```
# Applying StandardScaler base features of these new splits along with X_test
nnX_train_shortlist_scaled = scaler.fit_transform(nnX_train[shortlist])
nnX_val_shortlist_scaled = scaler.transform(nnX_val[shortlist])
X_test_shortlist_scaled = scaler.transform(X_test[shortlist])

# Creating the model
relu_fe = Sequential([
    Input(shape=nnX_train_shortlist_scaled.shape[1:]),
    Dense(100, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compiling the model
relu_fe.compile(loss='binary_crossentropy', optimizer='adam',␣
↪metrics=['accuracy'])

# Fitting the model
relu_fe.fit(nnX_train_shortlist_scaled, nny_train,␣
↪validation_data=(nnX_val_shortlist_scaled, nny_val), epochs=20,␣
↪batch_size=128)
```

```
# Creating predictions and calculating AUC score
train_auc = roc_auc_score(nny_train, relu_fe.
 ↪predict(nnX_train_shortlist_scaled))
test_auc = roc_auc_score(y_test, relu_fe.predict(X_test_shortlist_scaled))
relu_fe_pred = ['RelU FE', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = relu_fe_pred
results
```

```
Epoch 1/20
149/149              1s 2ms/step -
accuracy: 0.6824 - loss: 0.6010 - val_accuracy: 0.8766 - val_loss: 0.3513
Epoch 2/20
149/149              0s 1ms/step -
accuracy: 0.8742 - loss: 0.3534 - val_accuracy: 0.8781 - val_loss: 0.3480
Epoch 3/20
149/149              0s 992us/step -
accuracy: 0.8751 - loss: 0.3457 - val_accuracy: 0.8779 - val_loss: 0.3468
Epoch 4/20
149/149              0s 956us/step -
accuracy: 0.8759 - loss: 0.3405 - val_accuracy: 0.8774 - val_loss: 0.3464
Epoch 5/20
149/149              0s 939us/step -
accuracy: 0.8768 - loss: 0.3368 - val_accuracy: 0.8777 - val_loss: 0.3464
Epoch 6/20
149/149              0s 947us/step -
accuracy: 0.8768 - loss: 0.3339 - val_accuracy: 0.8774 - val_loss: 0.3467
Epoch 7/20
149/149              0s 1ms/step -
accuracy: 0.8765 - loss: 0.3314 - val_accuracy: 0.8777 - val_loss: 0.3472
Epoch 8/20
149/149              0s 970us/step -
accuracy: 0.8762 - loss: 0.3292 - val_accuracy: 0.8766 - val_loss: 0.3477
Epoch 9/20
149/149              0s 902us/step -
accuracy: 0.8762 - loss: 0.3272 - val_accuracy: 0.8768 - val_loss: 0.3483
Epoch 10/20
149/149              0s 939us/step -
accuracy: 0.8765 - loss: 0.3254 - val_accuracy: 0.8770 - val_loss: 0.3488
Epoch 11/20
149/149              0s 966us/step -
accuracy: 0.8770 - loss: 0.3238 - val_accuracy: 0.8772 - val_loss: 0.3494
Epoch 12/20
149/149              0s 1ms/step -
accuracy: 0.8774 - loss: 0.3222 - val_accuracy: 0.8766 - val_loss: 0.3500
Epoch 13/20
```

```
149/149              0s 1ms/step -
accuracy: 0.8774 - loss: 0.3206 - val_accuracy: 0.8766 - val_loss: 0.3506
Epoch 14/20
149/149              0s 1ms/step -
accuracy: 0.8781 - loss: 0.3191 - val_accuracy: 0.8762 - val_loss: 0.3513
Epoch 15/20
149/149              0s 1ms/step -
accuracy: 0.8782 - loss: 0.3177 - val_accuracy: 0.8760 - val_loss: 0.3520
Epoch 16/20
149/149              0s 974us/step -
accuracy: 0.8781 - loss: 0.3163 - val_accuracy: 0.8760 - val_loss: 0.3527
Epoch 17/20
149/149              0s 956us/step -
accuracy: 0.8784 - loss: 0.3150 - val_accuracy: 0.8756 - val_loss: 0.3533
Epoch 18/20
149/149              0s 1ms/step -
accuracy: 0.8787 - loss: 0.3136 - val_accuracy: 0.8756 - val_loss: 0.3540
Epoch 19/20
149/149              0s 989us/step -
accuracy: 0.8791 - loss: 0.3123 - val_accuracy: 0.8756 - val_loss: 0.3547
Epoch 20/20
149/149              0s 1ms/step -
accuracy: 0.8792 - loss: 0.3111 - val_accuracy: 0.8753 - val_loss: 0.3554
595/595              0s 616us/step
186/186              0s 523us/step
```

[38]:

	Model	Train AUC	Test AUC
0	Benchmark	0.5000000	0.5000000
1	Single Feature Logit	0.6646775	0.6664240
2	Logit Base Features	0.6601144	0.6591540
3	LASSO Base Features	0.6979545	0.6886940
4	Logit Base Interactions	0.6546322	0.6526143
5	LASSO Base Interactions	0.6979541	0.6886932
6	RF Base Features	0.9903982	0.7017100
7	GBM Base Features	0.7603463	0.7024283
8	XGB Base Features	0.9894109	0.6670103
9	RelU Hidden Base Features	0.7695974	0.6898747
10	Two Relu 256 Hidden Base Features	0.9347406	0.6535879
11	RelU 50 Base Features	0.7543033	0.6878324
12	LASSO FE	0.7057548	0.6936707
13	RF FE	0.9909127	0.6998723
14	GBM FE	0.7715692	0.7093004
15	RelU FE	0.8002753	0.6955664

For the neural network model adding the feature engineered data caused it to perform only slightly better than the base features version, and worse than the GBM model. Intuitively this makes sense since the data may not be complex enough for a neural network to outperform other model types, including a GBM model that is designed for situations where data may be more simplistic in nature.

1.7.5 GBM Model on All Features (No shortlist)

For comparison's sake, the GBM model will be rerun without a shortlist

[39]:
```
# Creating and fitting GBM to training
gbm_all = GradientBoostingClassifier(min_samples_split=30, random_state=prng).
 ↪fit(X_train, y_train)

# Creating predictions and calculating AUC score
train_auc = roc_auc_score(y_train, gbm_all.predict_proba(X_train)[:,1])
test_auc = roc_auc_score(y_test, gbm_all.predict_proba(X_test)[:,1])
gbm_all_pred = ['GBM All Features', train_auc, test_auc]

# Adding to results
results.loc[len(results)] = gbm_all_pred
results
```

[39]:

	Model	Train AUC	Test AUC
0	Benchmark	0.5000000	0.5000000
1	Single Feature Logit	0.6646775	0.6664240
2	Logit Base Features	0.6601144	0.6591540
3	LASSO Base Features	0.6979545	0.6886940
4	Logit Base Interactions	0.6546322	0.6526143
5	LASSO Base Interactions	0.6979541	0.6886932
6	RF Base Features	0.9903982	0.7017100
7	GBM Base Features	0.7603463	0.7024283
8	XGB Base Features	0.9894109	0.6670103
9	RelU Hidden Base Features	0.7695974	0.6898747
10	Two Relu 256 Hidden Base Features	0.9347406	0.6535879
11	RelU 50 Base Features	0.7543033	0.6878324
12	LASSO FE	0.7057548	0.6936707
13	RF FE	0.9909127	0.6998723
14	GBM FE	0.7715692	0.7093004
15	RelU FE	0.8002753	0.6955664
16	GBM All Features	0.7717604	0.7126532

From these results it's clear that the shortlist is not as beneficial as simply running the model on all of the features. Some of the patterns in the data is lost from the shortlisted features and not being captured. Going forward the GBM model on all features will be used for making a prediction for submission.

1.8 Analysis and Conclusion

From looking at the perfomance of all the models the GBM model appears to perform the best for this dataset. This is likely due to GBM's unique method of robustness to overfitting with regularization techniques like shrinkage being applied to prevent memorization of the training data that other models like the Random Forest and Neural Network models don't apply.

When examining the feature engineered variables using the LASSO shortlist a positive effect is shown on the prediction power, albeit not greatly. This means that while capturing some additional patterns in the data, the shorlisting does not significantly alter the prediction power. That being said, the GBM model takes the most advantage of the feature engineering to produce the best score from the set overall, and when tested on all features (not just the shortlisted ones) the GBM model produced the best prediction yet.

The overall conclusion is that the feature engineered GBM model created on all features is the best model for making a prediction to submit towards the Kaggle competition.

1.8.1 Generating Kaggle submission

The Kaggle submission is to be generated using the model on the data from `test.csv` downloaded from the competition page and generating a submission file that has the `article_id` as the index and the prediction probabilities genreated for each observation as the `score` column.

The same feature engineering will need to be done to the kaggle test data to allow the model to predict correctly.

[42]:
```
# Loading the test.csv data from Kaggle
test_data = pd.read_csv('Data/test.csv', index_col='article_id')

# Dropping timedelta
test_data.drop(columns=['timedelta'], inplace=True)

# Adding feature engineered columns to test data in order to allow FE GBM model␣
↪to run
test_data['d_mon_tues'] = test_data['weekday_is_monday'] +␣
↪test_data['weekday_is_tuesday']
test_data.rename(columns={'is_weekend': 'd_weekend'}, inplace=True)
test_data.drop(columns=weekdays, inplace=True)
[test_data.rename(columns={genre: genre.replace('data_channel_is_',␣
↪'d_genre_')}, inplace=True) for genre in genres]
test_data['d_no_words'] = (test_data['average_token_length'] == 0).astype(int)

test_data['f_log_tokens_content'] = np.where(test_data['n_tokens_content'] ==␣
↪0, 0, np.log(test_data['n_tokens_content']))
test_data['f_log_kw_avg_avg'] = np.where(test_data['kw_avg_avg'] == 0, 0, np.
↪log(test_data['kw_avg_avg']))
test_data['f_n_tokens_title_sq'] = np.power(test_data['n_tokens_title'], 2)
test_data['f_num_imgs_sq'] = np.power(test_data['num_imgs'], 2)
test_data['f_num_keywords'] = np.power(test_data['num_keywords'], 2)
test_data['f_global_pos_neg_word_ratio'] = np.
↪where((test_data['global_rate_positive_words'] > 0) &␣
↪(test_data['global_rate_negative_words'] == 0), 1,
                                              np.
↪where((test_data['global_rate_positive_words'] == 0) &␣
↪(test_data['global_rate_negative_words'] == 0), 0,
```

```
␣
↪test_data['global_rate_positive_words'] /␣
↪test_data['global_rate_negative_words']))
test_data['f_token_img_ratio'] = np.where((test_data['n_tokens_content'] > 0) &␣
↪(test_data['num_imgs'] == 0), 1,
                              np.where((test_data['n_tokens_content'] == 0) &␣
↪(test_data['num_imgs'] == 0), 0,
                                   test_data['n_tokens_content'] /␣
↪test_data['num_imgs']))
test_data['f_token_video_ratio'] = np.where((test_data['n_tokens_content'] > 0)␣
↪& (test_data['num_videos'] == 0), 1,
                                  np.where((test_data['n_tokens_content'] == 0)␣
↪& (test_data['num_videos'] == 0), 0,
                                       test_data['n_tokens_content'] /␣
↪test_data['num_videos']))
test_data['f_avg_polarity'] = test_data['avg_positive_polarity'] +␣
↪test_data['avg_negative_polarity']

# Creating predictions using the test data from Kaggle
gbe_all_kaggle_pred = gbm_all.predict_proba(test_data)[:,1]

# Creating submission file per specifications using date created in filename
pd.DataFrame(gbe_all_kaggle_pred, columns=['score'], index=test_data.index).
↪to_csv('Submissions/submission_nf_20240419-4.csv')
```

```
C:\Users\Xrona\anaconda3\Lib\site-packages\pandas\core\arraylike.py:396:
RuntimeWarning: divide by zero encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
C:\Users\Xrona\anaconda3\Lib\site-packages\pandas\core\arraylike.py:396:
RuntimeWarning: divide by zero encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```