arbash-malik-assignment-3

April 19, 2024

1 Importing helper libraries

[37]:
```
import warnings
warnings.filterwarnings('ignore')

import pandas as pd
import numpy as np
import os
import sys
import matplotlib.pyplot as plt
import re
import ast
import patsy
import datetime
import math
import statsmodels.api as sm
import sklearn.metrics as metrics
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.formula.api as smf
import time
import plotly
import xgboost as xgb
import tensorflow as tf
import keras

from pathlib                          import Path
from plotnine                         import *
from patsy                            import dmatrices
from collections                      import defaultdict
from plotnine.data                    import mpg
from mizani.formatters                import percent_format
from statsmodels.tools.eval_measures import mse,rmse
from plotly.express import *

from sklearn.model_selection import train_test_split, GridSearchCV, KFold,␣
↪RandomizedSearchCV, RepeatedKFold
```

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble        import␣
 ↪HistGradientBoostingClassifier,RandomForestClassifier,GradientBoostingClassifier
from sklearn.linear_model    import LogisticRegression,␣
 ↪LogisticRegressionCV,Lasso
from sklearn.metrics         import brier_score_loss, roc_curve, auc,␣
 ↪confusion_matrix, roc_auc_score, mean_squared_error,␣
 ↪accuracy_score,make_scorer
from sklearn.preprocessing   import StandardScaler
from sklearn.pipeline        import Pipeline
from sklearn.tree            import DecisionTreeClassifier
from scipy.stats.mstats      import winsorize
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from xgboost                 import XGBClassifier
from keras.metrics           import AUC
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks  import EarlyStopping


prng = np.random.RandomState(20240415)
keras.utils.set_random_seed(20240405)
```

2 Importing Dataset

[38]:
```
main_df = pd.read_csv('train.csv', index_col='article_id')
test_df = pd.read_csv('test.csv', index_col='article_id')
```

[39]:
```
main_df.shape
```

[39]:
```
(29733, 60)
```

2.1 Basic EDA

[43]:
```
main_df.describe(percentiles = [0.05,0.1,0.75,0.90,0.95,0.96,0.97,0.98,0.99])
```

[43]:
```
          timedelta  n_tokens_title  n_tokens_content  n_unique_tokens  \
count  29733.000000    29733.000000      29733.000000     29733.000000
mean     355.645646       10.390812        545.008274         0.555076
std      214.288261        2.110135        469.358037         4.064572
min        8.000000        2.000000          0.000000         0.000000
5%        43.000000        7.000000        104.000000         0.351315
10%       72.000000        8.000000        151.000000         0.407609
50%      342.000000       10.000000        409.000000         0.539894
75%      545.000000       12.000000        712.000000         0.609375
```

90%	662.000000	13.000000	1091.000000	0.677778
95%	697.000000	14.000000	1396.000000	0.722109
96%	704.000000	14.000000	1514.000000	0.735043
97%	710.000000	14.000000	1664.040000	0.752941
98%	716.360000	15.000000	1862.080000	0.772865
99%	724.000000	15.000000	2256.720000	0.804348
max	731.000000	23.000000	8474.000000	701.000000

	n_non_stop_words	n_non_stop_unique_tokens	num_hrefs	\
count	29733.000000	29733.000000	29733.000000	
mean	1.005852	0.695432	10.912690	
std	6.039655	3.768796	11.316508	
min	0.000000	0.000000	0.000000	
5%	1.000000	0.479450	1.000000	
10%	1.000000	0.555556	2.000000	
50%	1.000000	0.690566	8.000000	
75%	1.000000	0.755208	14.000000	
90%	1.000000	0.819444	23.000000	
95%	1.000000	0.857143	30.000000	
96%	1.000000	0.869565	33.000000	
97%	1.000000	0.882353	36.000000	
98%	1.000000	0.900000	43.000000	
99%	1.000000	0.923077	56.000000	
max	1042.000000	650.000000	304.000000	

	num_self_hrefs	num_imgs	num_videos	…	min_positive_polarity	\
count	29733.000000	29733.000000	29733.000000	…	29733.000000	
mean	3.290788	4.524535	1.263546	…	0.095593	
std	3.840874	8.213823	4.189080	…	0.071503	
min	0.000000	0.000000	0.000000	…	0.000000	
5%	0.000000	0.000000	0.000000	…	0.033333	
10%	0.000000	0.000000	0.000000	…	0.033333	
50%	2.000000	1.000000	0.000000	…	0.100000	
75%	4.000000	4.000000	1.000000	…	0.100000	
90%	6.000000	14.000000	2.000000	…	0.160000	
95%	9.000000	20.000000	6.000000	…	0.200000	
96%	10.000000	22.000000	10.000000	…	0.250000	
97%	11.000000	25.000000	11.000000	…	0.250000	
98%	13.000000	30.000000	16.000000	…	0.300000	
99%	20.000000	36.000000	21.000000	…	0.400000	
max	74.000000	111.000000	91.000000	…	1.000000	

	max_positive_polarity	avg_negative_polarity	min_negative_polarity	\
count	29733.000000	29733.000000	29733.000000	
mean	0.757780	-0.259709	-0.520981	
std	0.247293	0.128488	0.290454	
min	0.000000	-1.000000	-1.000000	

```
5%                 0.375000              -0.470000              -1.000000
10%                0.500000              -0.410000              -1.000000
50%                0.800000              -0.252827              -0.500000
75%                1.000000              -0.186494              -0.300000
90%                1.000000              -0.122917              -0.150000
95%                1.000000               0.000000               0.000000
96%                1.000000               0.000000               0.000000
97%                1.000000               0.000000               0.000000
98%                1.000000               0.000000               0.000000
99%                1.000000               0.000000               0.000000
max                1.000000               0.000000               0.000000

       max_negative_polarity  title_subjectivity  title_sentiment_polarity  \
count           29733.000000        29733.000000              29733.000000
mean               -0.107793            0.281878                  0.069691
std                 0.095672            0.323461                  0.264379
min                -1.000000            0.000000                 -1.000000
5%                 -0.250000            0.000000                 -0.337292
10%                -0.187500            0.000000                 -0.145833
50%                -0.100000            0.144444                  0.000000
75%                -0.050000            0.500000                  0.136364
90%                -0.050000            0.800000                  0.433333
95%                 0.000000            1.000000                  0.500000
96%                 0.000000            1.000000                  0.550000
97%                 0.000000            1.000000                  0.616667
98%                 0.000000            1.000000                  0.800000
99%                 0.000000            1.000000                  1.000000
max                 0.000000            1.000000                  1.000000

       abs_title_subjectivity  abs_title_sentiment_polarity    is_popular
count            29733.000000                  29733.000000  29733.000000
mean                 0.341427                      0.155234      0.121649
std                  0.188735                      0.225066      0.326886
min                  0.000000                      0.000000      0.000000
5%                   0.000000                      0.000000      0.000000
10%                  0.045455                      0.000000      0.000000
50%                  0.500000                      0.000000      0.000000
75%                  0.500000                      0.250000      0.000000
90%                  0.500000                      0.500000      1.000000
95%                  0.500000                      0.600000      1.000000
96%                  0.500000                      0.666667      1.000000
97%                  0.500000                      0.750000      1.000000
98%                  0.500000                      0.875000      1.000000
99%                  0.500000                      1.000000      1.000000
max                  0.500000                      1.000000      1.000000

[15 rows x 60 columns]
```

[41]:

```python
# Checking for null values

null_counts = main_df.isnull().sum()

for col, null_count in null_counts.items():
    if null_count > 0:
        print(f"Column '{col}' has {null_count} null values.")
    else:
        print(f"Column '{col}' has no null values.")
```

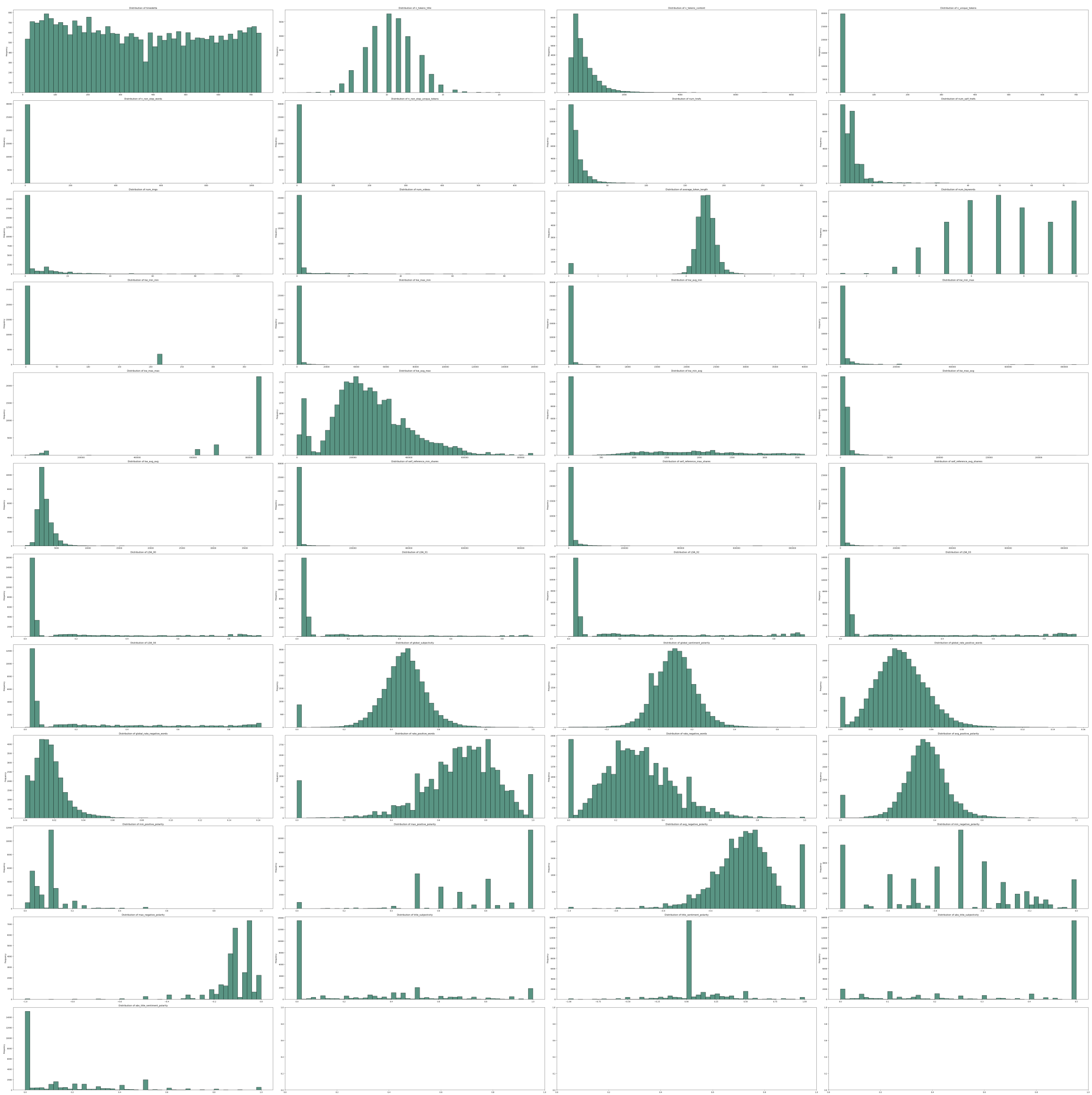
```
Column 'timedelta' has no null values.
Column 'n_tokens_title' has no null values.
Column 'n_tokens_content' has no null values.
Column 'n_unique_tokens' has no null values.
Column 'n_non_stop_words' has no null values.
Column 'n_non_stop_unique_tokens' has no null values.
Column 'num_hrefs' has no null values.
Column 'num_self_hrefs' has no null values.
Column 'num_imgs' has no null values.
Column 'num_videos' has no null values.
Column 'average_token_length' has no null values.
Column 'num_keywords' has no null values.
Column 'data_channel_is_lifestyle' has no null values.
Column 'data_channel_is_entertainment' has no null values.
Column 'data_channel_is_bus' has no null values.
Column 'data_channel_is_socmed' has no null values.
Column 'data_channel_is_tech' has no null values.
Column 'data_channel_is_world' has no null values.
Column 'kw_min_min' has no null values.
Column 'kw_max_min' has no null values.
Column 'kw_avg_min' has no null values.
Column 'kw_min_max' has no null values.
Column 'kw_max_max' has no null values.
Column 'kw_avg_max' has no null values.
Column 'kw_min_avg' has no null values.
Column 'kw_max_avg' has no null values.
Column 'kw_avg_avg' has no null values.
Column 'self_reference_min_shares' has no null values.
Column 'self_reference_max_shares' has no null values.
Column 'self_reference_avg_sharess' has no null values.
Column 'weekday_is_monday' has no null values.
Column 'weekday_is_tuesday' has no null values.
Column 'weekday_is_wednesday' has no null values.
Column 'weekday_is_thursday' has no null values.
Column 'weekday_is_friday' has no null values.
Column 'weekday_is_saturday' has no null values.
Column 'weekday_is_sunday' has no null values.
```

```
Column 'is_weekend' has no null values.
Column 'LDA_00' has no null values.
Column 'LDA_01' has no null values.
Column 'LDA_02' has no null values.
Column 'LDA_03' has no null values.
Column 'LDA_04' has no null values.
Column 'global_subjectivity' has no null values.
Column 'global_sentiment_polarity' has no null values.
Column 'global_rate_positive_words' has no null values.
Column 'global_rate_negative_words' has no null values.
Column 'rate_positive_words' has no null values.
Column 'rate_negative_words' has no null values.
Column 'avg_positive_polarity' has no null values.
Column 'min_positive_polarity' has no null values.
Column 'max_positive_polarity' has no null values.
Column 'avg_negative_polarity' has no null values.
Column 'min_negative_polarity' has no null values.
Column 'max_negative_polarity' has no null values.
Column 'title_subjectivity' has no null values.
Column 'title_sentiment_polarity' has no null values.
Column 'abs_title_subjectivity' has no null values.
Column 'abs_title_sentiment_polarity' has no null values.
Column 'is_popular' has no null values.
```

[42]:

```
# Identify binary variables by checking the number of unique values
binary_threshold = 2  # Binary variables have exactly 2 unique values

# Filter out binary variables from the dataframe
non_binary_columns = [col for col in main_df.columns if main_df[col].nunique()␣
↪> binary_threshold]
filtered_df = main_df[non_binary_columns]

# Set up the dimensions of the figure and subplots
num_cols = len(filtered_df.columns)
num_rows = 12
num_cols_per_row = 4

fig, axes = plt.subplots(nrows=num_rows, ncols=num_cols_per_row, figsize=(80,␣
↪80))
axes = axes.flatten()  # Flatten the 2D array of axes for easier iteration

# Plot histograms for each numerical variable (excluding binary variables)
for i, column in enumerate(filtered_df.columns):
    ax = axes[i]
    if i < num_cols:
        sns.histplot(filtered_df[column], ax=ax, kde=False, bins=50, color =␣
↪'#227159')  # Use sns.histplot for histograms without KDE
```

```
        ax.set_title(f"Distribution of {column}", fontsize=12)
        ax.set_xlabel("")
        ax.set_ylabel("Frequency")
    else:
        ax.set_visible(False)

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```



A lot of features have skewed distribution, which can be confirmed by the describe table. There are huge outliers after 90th percentile. Instead of dropping I will be winsorizing them. It is a very strict winsorize but I believe it would be better for the models.

[7]:
```python
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=(0.01, 0.
↪1))  # Set limits to 1st and 95th percentiles
    return data_copy
```

[8]:
```python
columns_to_winsorize = main_df.drop(['timedelta', 'is_popular'], axis=1)
main_df = winsorize_columns(main_df,columns_to_winsorize)
```

2.2 Variable Selection

I plan to use Lasso shortlisting through a Logistic regression (LogisticRegressionCV) to shortlist my variables. This will help me save plenty of time and give a sense of direction.

[9]:
```python
# Split into train and test sets

X = main_df.drop(['timedelta', 'is_popular'], axis=1)
y = main_df['is_popular']


# Split the data into training and test sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=prng)
```

[10]:
```python
%%time

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

# Define the range of regularization strengths (inverse of regularization␣
↪parameter C)
Cs = np.arange(0.01, 1, 0.01)  # Example list of Cs (inverse of regularization␣
↪strength)

# Define the KFold cross-validation splitter
kfold = KFold(n_splits=5, shuffle=True, random_state=prng)

logreg_cv = LogisticRegressionCV(Cs=Cs, cv=kfold, scoring='roc_auc',␣
↪penalty='l1', refit = True, solver='liblinear', random_state=prng)

# Fit the model to your data (X_train, y_train)
logreg_cv.fit(X_train_scaled, y_train)

# Get the best regularization strength (C) selected by cross-validation
best_C = logreg_cv.C_[0]
```

```
print("Best C (inverse of regularization strength):", best_C)
```

```
Best C (inverse of regularization strength): 0.11
CPU times: total: 7min 7s
Wall time: 7min 32s
```

[11]:
```
# Make predictions using the best model on validation data (X_val)
y_pred_proba = logreg_cv.predict_proba(X_val)[:, 1]

# Evaluate the model performance using ROC AUC score
lassologit_roc_auc = roc_auc_score(y_val, y_pred_proba)
print("Logit Lasso - ROC AUC:", lassologit_roc_auc)
```

```
Logit Lasso - ROC AUC: 0.49588466645202806
```

[12]:
```
# Get the best coefficients from the model
best_coefficients = logreg_cv.coef_.flatten()  # Flatten to 1D array

# Get feature names corresponding to non-zero coefficients
# Filter feature names based on coefficients greater than zero
feature_names = np.array(X_train.columns)
selected_features = feature_names[best_coefficients != 0]

# Filter best coefficients to include only non zero
non_zero_coefficients = best_coefficients[best_coefficients != 0]

# Create a DataFrame with selected features and corresponding positive␣
↪coefficients
feature_coefficients_df = pd.DataFrame({
    'Feature': selected_features,
    'Coefficient': non_zero_coefficients
})
```

[13]:
```
selected_features =␣
↪feature_coefficients_df[abs(feature_coefficients_df['Coefficient']) >= 0.03].
↪reset_index(drop=True)
selected_features = selected_features['Feature']
selected_features
```

[13]:
```
0                        num_hrefs
1                   num_self_hrefs
2                         num_imgs
3                       num_videos
4             average_token_length
5                     num_keywords
6    data_channel_is_entertainment
7                data_channel_is_bus
```

```
8              data_channel_is_tech
9                        kw_min_min
10                       kw_avg_max
11                       kw_min_avg
12                       kw_avg_avg
13          self_reference_min_shares
14         self_reference_avg_sharess
15                  weekday_is_monday
16                       is_weekend
17                           LDA_00
18                           LDA_02
19                           LDA_03
20               global_subjectivity
21               rate_positive_words
22               rate_negative_words
23              min_positive_polarity
24                title_subjectivity
25            abs_title_subjectivity
Name: Feature, dtype: object
```

3 All Models

Test Validation Split with our shortlisted features Since I have already made a simple logistic model, I am planning to use tree, ensemble methods & neural network: 1. Decision Tree 2. Random Forest 3. Gradient Boosting 4. HistGradient Boosting 5. ExtremeGradient Boosting 6. Neural Network

[14]:
```
# Split into train and test sets
X = main_df[selected_features]
y = main_df['is_popular']

# Split the data into training and test sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=prng, stratify=y)
```

I am using a 20% data split for my validation set to fine tune my models. I also used ***stratify*** to make sure that the balances are proportionate. My plan is to use ***kfold crossvalidation*** as well for every model so that my models are generalized better. I am using 5 folds with shuffle being true to ensure randomness for better generalization.

3.1 1. Decision Tree

[15]:
```
# Define the number of splits and random state for KFold
k = KFold(n_splits=5, shuffle=True, random_state=prng)

# Define the lists of values for max_features and min_samples_split
max_features = [15,20,25]
min_samples_split = [100,200,300,400,500,600,700,800,900,1000]
```

```
# Define the parameter grid for GridSearchCV
param_grid = {
    'max_features': max_features,
    'criterion': ['gini'],
    'min_samples_split': min_samples_split
}

# Initialize the Decision Tree classifier with specified parameters
decision_tree = DecisionTreeClassifier(random_state=prng,␣
 ↪class_weight='balanced')

# Create GridSearchCV with refit='roc_auc' and specified scoring metrics
decision_tree_grid = GridSearchCV(decision_tree, param_grid, cv=k,␣
 ↪refit='roc_auc', scoring='roc_auc', n_jobs=-1)

# Fit the GridSearchCV to the data
decision_tree_grid.fit(X_train, y_train)

# Print the best parameters found by GridSearchCV
print("Best Parameters:", decision_tree_grid.best_params_)
```

```
Best Parameters: {'criterion': 'gini', 'max_features': 25, 'min_samples_split':
900}
```

[16]:
```
# Get the best Decision Tree model from GridSearchCV
best_decision_tree = decision_tree_grid.best_estimator_

# Evaluate the best model on the test set (assuming X_test and y_test are␣
 ↪defined)
y_pred = best_decision_tree.predict(X_val)
dt_roc_auc = roc_auc_score(y_val, y_pred)

print("Decision Tree - ROC AUC Score:", dt_roc_auc)
```

```
Decision Tree - ROC AUC Score: 0.6377625397410398
```

The Decision Tree model performs better on validation as compared to the logit model. But even then the score is a bit low.

3.2 2. Random forest

Bieng limited on computing power, I gave a small grid. My max_features numbers are based on the recommended square root of the features. The number of estimators are also low to prevent any overfitting.

[17]:
```
%%time
```

```python
k = KFold(n_splits=5, shuffle=True, random_state=prng)

# Define the lists of values for max_features and min_samples_split
max_features = [3,4,5]
min_samples = [100,200,300]

# Define the parameter grid for GridSearchCV
param_grid = {
    'max_features': max_features,
    'criterion': ['gini'],
    'min_samples_split': min_samples
}

# Initialize the Random Forest classifier with specified parameters
random_forest = RandomForestClassifier(random_state=prng, n_estimators=100,␣
↪oob_score=True,class_weight='balanced',bootstrap=True)

# Create GridSearchCV with refit='roc_auc' and specified scoring metrics
random_forest_grid = GridSearchCV(random_forest, param_grid, cv=k,␣
↪refit='roc_auc', scoring='roc_auc', n_jobs=-1)

# Fit the GridSearchCV to the data
random_forest_grid.fit(X_train, y_train)

# Print the best parameters found by GridSearchCV
print("Best Parameters:", random_forest_grid.best_params_)
```

```
Best Parameters: {'criterion': 'gini', 'max_features': 3, 'min_samples_split':
100}
CPU times: total: 4.95 s
Wall time: 44.9 s
```

[18]:

```python
# Get the best model from the grid search
best_model_rf = random_forest_grid.best_estimator_

# Make predictions using the best model on validation data
y_pred_rf = best_model_rf.predict_proba(X_val)[:, 1]

# Evaluate using ROC AUC score
roc_auc_rf = roc_auc_score(y_val, y_pred_rf)
print("Random Forest - ROC AUC:", roc_auc_rf)
```

```
Random Forest - ROC AUC: 0.7290743964974932
```

The random forest model performs better and is competetive to the scores on the leaderboard, so directionally it is a good model

3.3 3. Gradient Boosting

Since from looking at the data, one can deduce that the realationships should be linear and not complex so that is why I gave a big grid for the gradient boosting model. The number of estimators are also low range to prevent overfitting. I also gave two learning rates to have a better gridsearch.

[19]:
```
%%time

# Define the number of splits and random state for KFold
k = KFold(n_splits=5, shuffle=True, random_state=prng)

# Define the lists of values for max_features and min_samples_split
max_features = [3,4,5,6]
min_samples = [150,300,450]
learning_rate = [0.01, 0.1]
n_estimators = [50,100,150]

# Define the parameter grid for GridSearchCV
param_grid = {
    'max_features': max_features,
    'min_samples_split': min_samples,
    'learning_rate': learning_rate,
    'n_estimators': n_estimators
}

# Initialize the Gradient Boosting classifier with specified parameters
gradient_boost = GradientBoostingClassifier(random_state=prng)

# Create GridSearchCV with refit='roc_auc' and specified scoring metrics
gradient_boost_grid = GridSearchCV(gradient_boost, param_grid, cv=k,
 ↪refit='roc_auc', scoring='roc_auc', n_jobs=-1)

# Fit the GridSearchCV to the data
gradient_boost_grid.fit(X_train, y_train)

# Print the best parameters found by GridSearchCV
print("Best Parameters:", gradient_boost_grid.best_params_)
```

```
Best Parameters: {'learning_rate': 0.1, 'max_features': 4, 'min_samples_split':
150, 'n_estimators': 150}
CPU times: total: 15.1 s
Wall time: 2min 55s
```

[20]:
```
# Get the best model from the grid search
best_model_gb = gradient_boost_grid.best_estimator_

# Make predictions using the best model on validation data
y_pred_gb = best_model_gb.predict_proba(X_val)[:, 1]
```

```
# Evaluate using ROC AUC score
roc_auc_gb = roc_auc_score(y_val, y_pred_gb)
print("Gradient Boosting - ROC AUC:", roc_auc_gb)
```

```
Gradient Boosting - ROC AUC: 0.7348049432452411
```

The Gradient Boosting model performs better than the random forest but not by a big margin.

3.4 4. HistGradient Boosting

I used this model to increase my computation speed on a much larger grid.

[21]:
```
%%time

# Define the number of splits and random state for KFold
k = KFold(n_splits=5, shuffle=True, random_state=prng)

# Define the lists of values for max_bins, learning_rate, and max_iter
max_bins = [32,64,128]
learning_rate = [0.001, 0.01, 0.03, 0.05, 0.1]
max_iter = [50, 100, 150, 200, 250, 300]

# Define the parameter grid for GridSearchCV
param_grid = {
    'max_bins': max_bins,
    'learning_rate': learning_rate,
    'max_iter': max_iter,
    'loss': ['log_loss']
}

# Initialize the HistGradientBoostingClassifier with specified parameters
hist_gradient_boost = HistGradientBoostingClassifier(random_state=prng)

# Create GridSearchCV with refit='roc_auc' and specified scoring metrics
hist_gradient_boost_grid = GridSearchCV(hist_gradient_boost, param_grid, cv=k,␣
 ↪refit='roc_auc', scoring='roc_auc', n_jobs=-1)

# Fit the GridSearchCV to the data
hist_gradient_boost_grid.fit(X_train, y_train)

# Print the best parameters found by GridSearchCV
print("Best Parameters:", hist_gradient_boost_grid.best_params_)
```

```
Best Parameters: {'learning_rate': 0.03, 'loss': 'log_loss', 'max_bins': 64,
'max_iter': 200}
CPU times: total: 18.3 s
Wall time: 2min 13s
```

[22]:
```python
# Get the best model from the grid search
best_model_gb = hist_gradient_boost_grid.best_estimator_

# Make predictions using the best model on validation data
y_pred_gb = best_model_gb.predict_proba(X_val)[:, 1]

# Evaluate using ROC AUC score
roc_auc_hist_gb = roc_auc_score(y_val, y_pred_gb)
print("HistGradient - Best ROC AUC:", roc_auc_hist_gb)
```

```
HistGradient - Best ROC AUC: 0.7328307322941885
```

The model performs well as compared to the Random Forest but only slightly worse than the simple GBM.

3.5 5. XGradient Boosting

Using XG Boost would be a nice way to check if the features are complex. XGBoost usually performs better in terms of flexibility and speed. Although it would depend on the dataset.

[23]:
```python
# Define the number of splits and random state for KFold
k = KFold(n_splits=5, shuffle=True, random_state=prng)

# Define the lists of values for max_depth, learning_rate, and n_estimators
max_depth = [3, 4, 5 ,6]
learning_rate = [0.001, 0.01, 0.03, 0.05, 0.1]
n_estimators = [50, 100, 150, 200, 250,300]

# Define the parameter grid for GridSearchCV
param_grid = {
    'max_depth': max_depth,
    'learning_rate': learning_rate,
    'n_estimators': n_estimators
}

# Initialize the XGBClassifier with specified parameters
xgb_classifier = XGBClassifier(objective = 'binary:logistic', random_state=prng)

# Create GridSearchCV with refit='roc_auc' and specified scoring metrics
xgb_classifier_grid = GridSearchCV(xgb_classifier, param_grid, cv=k,␣
 ↪refit='roc_auc', scoring='roc_auc', n_jobs=-1)

# Fit the GridSearchCV to the data
xgb_classifier_grid.fit(X_train, y_train)

# Print the best parameters found by GridSearchCV
print("Best Parameters:", xgb_classifier_grid.best_params_)
```

```
Best Parameters: {'learning_rate': 0.05, 'max_depth': 3, 'n_estimators': 200}
```

[24]:
```python
# Get the best model from the grid search
best_model_gb = xgb_classifier_grid.best_estimator_

# Make predictions using the best model on validation data
y_pred_gb = best_model_gb.predict_proba(X_val)[:, 1]

# Evaluate using ROC AUC score
roc_auc_xgb = roc_auc_score(y_val, y_pred_gb)
print("Gradient Boosting Best ROC AUC:", roc_auc_xgb)
```

```
Gradient Boosting Best ROC AUC: 0.7299205814635717
```

It does not perform better than the simple GBM as well as the HistGBM.

3.6 6. Neural Network

Lastly, I created a simple Neural Network to check if it performs well in a binary classification model.

[25]:
```python
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Initialize a neural network model
model_nn = Sequential([
    Dense(26, activation='relu', input_shape=(X_train_scaled.shape[1],)),
    Dense(13, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the model
model_nn.compile(optimizer='adam', loss='binary_crossentropy', metrics=[AUC()])

# Train the model
model_nn.fit(X_train_scaled, y_train, validation_data=(X_val_scaled, y_val),␣
↪epochs=15, batch_size=32)
```

```
Epoch 1/15
744/744              3s 2ms/step -
auc: 0.5904 - loss: 0.4086 - val_auc: 0.7040 - val_loss: 0.3427
Epoch 2/15
744/744              1s 2ms/step -
auc: 0.6909 - loss: 0.3507 - val_auc: 0.7144 - val_loss: 0.3393
Epoch 3/15
744/744              1s 2ms/step -
auc: 0.7054 - loss: 0.3462 - val_auc: 0.7178 - val_loss: 0.3381
Epoch 4/15
744/744              1s 2ms/step -
auc: 0.7124 - loss: 0.3438 - val_auc: 0.7188 - val_loss: 0.3378
```

```
Epoch 5/15
744/744              1s 2ms/step -
auc: 0.7183 - loss: 0.3418 - val_auc: 0.7183 - val_loss: 0.3378
Epoch 6/15
744/744              2s 2ms/step -
auc: 0.7225 - loss: 0.3404 - val_auc: 0.7182 - val_loss: 0.3380
Epoch 7/15
744/744              2s 2ms/step -
auc: 0.7258 - loss: 0.3392 - val_auc: 0.7175 - val_loss: 0.3382
Epoch 8/15
744/744              1s 2ms/step -
auc: 0.7292 - loss: 0.3381 - val_auc: 0.7175 - val_loss: 0.3382
Epoch 9/15
744/744              1s 2ms/step -
auc: 0.7319 - loss: 0.3371 - val_auc: 0.7167 - val_loss: 0.3386
Epoch 10/15
744/744              1s 2ms/step -
auc: 0.7345 - loss: 0.3363 - val_auc: 0.7149 - val_loss: 0.3391
Epoch 11/15
744/744              1s 2ms/step -
auc: 0.7367 - loss: 0.3354 - val_auc: 0.7140 - val_loss: 0.3394
Epoch 12/15
744/744              1s 2ms/step -
auc: 0.7388 - loss: 0.3346 - val_auc: 0.7132 - val_loss: 0.3397
Epoch 13/15
744/744              1s 2ms/step -
auc: 0.7408 - loss: 0.3338 - val_auc: 0.7128 - val_loss: 0.3399
Epoch 14/15
744/744              1s 2ms/step -
auc: 0.7425 - loss: 0.3331 - val_auc: 0.7117 - val_loss: 0.3404
Epoch 15/15
744/744              1s 2ms/step -
auc: 0.7445 - loss: 0.3324 - val_auc: 0.7111 - val_loss: 0.3406
```

[25]: <keras.src.callbacks.history.History at 0x1b7c6a16990>

[26]:
```
# Make predictions on the test set
y_pred_nn = model_nn.predict(X_val_scaled).flatten()

# Evaluate using ROC AUC score
roc_auc_nn = roc_auc_score(y_val, y_pred_nn)
print("Neural Network ROC AUC:", roc_auc_nn)
```

```
186/186              0s 1ms/step
Neural Network ROC AUC: 0.7109446982646324
```

The performance is commendable as the validation AUC score is good but comparing to the ensemble methods it is worse.

4 Comparison

[27]:
```
auc_scores_df = pd.DataFrame({
    'Model': ['Lasso Logit','Decision Tree','RandomForest', 'GradientBoosting',␣
 ↪'HistGradientBoosting', 'XGBoost', 'NeuralNetwork'],
    'ROC_AUC_Score': [lassologit_roc_auc, dt_roc_auc ,roc_auc_rf, roc_auc_gb,␣
 ↪roc_auc_hist_gb, roc_auc_xgb, roc_auc_nn]
})

auc_scores_df
```

[27]:

	Model	ROC_AUC_Score
0	Lasso Logit	0.495885
1	Decision Tree	0.637763
2	RandomForest	0.729074
3	GradientBoosting	0.734805
4	HistGradientBoosting	0.732831
5	XGBoost	0.729921
6	NeuralNetwork	0.710945

Training our Best Model on all data to ensure better generalization

[31]:
```
# Split into train and test sets
X_train = main_df[selected_features]
y_train = main_df['is_popular']
```

[32]:
```
%%time

# Define the number of splits and random state for KFold
k = KFold(n_splits=5, shuffle=True, random_state=prng)

# Define the lists of values for max_features and min_samples_split
max_features = [3,4,5,6]
min_samples = [150,300,450]
learning_rate = [0.01, 0.1]
n_estimators = [50,100,150]

# Define the parameter grid for GridSearchCV
param_grid = {
    'max_features': max_features,
    'min_samples_split': min_samples,
    'learning_rate': learning_rate,
    'n_estimators': n_estimators
}

# Initialize the Gradient Boosting classifier with specified parameters
gradient_boost = GradientBoostingClassifier(random_state=prng)
```

```python
# Create GridSearchCV with refit='roc_auc' and specified scoring metrics
gradient_boost_grid = GridSearchCV(gradient_boost, param_grid, cv=k,␣
↪refit='roc_auc', scoring='roc_auc', n_jobs=-1)

# Fit the GridSearchCV to the data
gradient_boost_grid.fit(X_train, y_train)

# Print the best parameters found by GridSearchCV
print("Best Parameters:", gradient_boost_grid.best_params_)
```

```
Best Parameters: {'learning_rate': 0.1, 'max_features': 3, 'min_samples_split':
300, 'n_estimators': 150}
CPU times: total: 17.1 s
Wall time: 3min 49s
```

[33]:
```python
# Get the best model from the grid search
best_model_gb = gradient_boost_grid.best_estimator_

# Make predictions using the best model on validation data
y_pred_gb = best_model_gb.predict_proba(X_train)[:, 1]

# Evaluate using ROC AUC score
roc_auc_gb = roc_auc_score(y_train, y_pred_gb)
print("Gradient Boosting - Train ROC AUC:", roc_auc_gb)
```

```
Gradient Boosting - Train ROC AUC: 0.756570873074185
```

4.1 Generating file for submission

[44]:
```python
test_df = pd.read_csv('test.csv', index_col='article_id')
X_test = test_df[selected_features]

test_df['score'] = best_model_gb.predict_proba(X_test)[:, 1]
test_df.reset_index(inplace=True)

submission = test_df[['article_id','score']]
submission.to_csv('arbash_test.csv', sep =',', index = False, encoding =␣
↪'utf-8')
```

[ ]:
```python
```