Student: Artyom Ashigov

Introduction

The Kaggle competition has been launched, please register using this link.

You will find the training and test data in the data section of the competition, along with a description of the features. You will need to build models on the training data and make predictions on the test data and submit your solutions to Kaggle. You will also find a sample solution file in the data section that shows the format you will need to use for your own submissions.

The deadline for Kaggle solutions is 8PM on 19 April. You will be graded primarily on the basis of your work and how clearly you explain your methods and results. Those in the top three in the competition will receive some extra points. I expect you to experiment with all the methods we have covered: linear models, random forest, gradient boosting, neural networks + parameter tuning, feature engineering.

You will see the public score of your best model on the leaderboard. A private dataset will be used to evaluate the final performance of your model to avoid overfitting based on the leaderboard.

You should also submit to Moodle the documentation (ipynb and pdf) of your work, including exploratory data analysis, data cleaning, parameter tuning and evaluation. Aim for concise explanations.

Feel free to ask questions about the task in Slack. The Kaggle competition is already open, please start working on it and submitting solutions (you cannot submit more than 5 solutions per day).

The plan

Our plan for the Kaggle competition involves a systematic approach to model development and optimization. Initially, we will split the provided 'train' dataframe into two segments to serve as our training and validation sets. We intend to build and evaluate different predictive models—such as logistic regression, random forests, and gradient boosting machines—focusing. The best-performing model will be retrained using all train data and then then be applied to the external validation set. Finally, we will prepare and submit our predictions in the required format (article_id and score) to Kaggle, ensuring they align with the competition's guidelines.

Import Libraries

In [20]:
```
# import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from keras.callbacks import EarlyStopping
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization
from keras.optimizers import Adam
from interpret.glassbox import ExplainableBoostingClassifier
from sklearn.svm import SVC
import xgboost as xgb
from xgboost import XGBClassifier


# show max columns and rows
pd.set_option('display.max_columns', None)

pd.set_option('display.max_rows', None)

# ignore warnings
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
# read the data
train = pd.read_csv("train.csv")
# check the shape
print(train.shape)
```

```
# print first 5 rows
train.head()
```

(29733, 61)

Out[2]:

	timedelta	n_tokens_title	n_tokens_content	n_unique_tokens	n_non_stop_words	n_non_stop_unique_tokens	num_hrefs	num_self_hrefs	num_imgs	num_
0	594	9	702	0.454545	1.0	0.620438	11	2	1	
1	346	8	1197	0.470143	1.0	0.666209	21	6	2	
2	484	9	214	0.618090	1.0	0.748092	5	2	1	
3	639	8	249	0.621951	1.0	0.664740	16	5	8	
4	177	12	1219	0.397841	1.0	0.583578	21	1	1	

In [3]:

```
# check info
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 29733 entries, 0 to 29732
Data columns (total 61 columns):
 #   Column                         Non-Null Count  Dtype  
---  ------                         --------------  -----  
 0   timedelta                      29733 non-null  int64  
 1   n_tokens_title                 29733 non-null  int64  
 2   n_tokens_content               29733 non-null  int64  
 3   n_unique_tokens                29733 non-null  float64
 4   n_non_stop_words               29733 non-null  float64
 5   n_non_stop_unique_tokens       29733 non-null  float64
 6   num_hrefs                      29733 non-null  int64  
 7   num_self_hrefs                 29733 non-null  int64  
 8   num_imgs                       29733 non-null  int64  
 9   num_videos                     29733 non-null  int64  
 10  average_token_length           29733 non-null  float64
 11  num_keywords                   29733 non-null  int64  
 12  data_channel_is_lifestyle      29733 non-null  int64  
 13  data_channel_is_entertainment  29733 non-null  int64  
 14  data_channel_is_bus            29733 non-null  int64  
 15  data_channel_is_socmed         29733 non-null  int64  
 16  data_channel_is_tech           29733 non-null  int64  
 17  data_channel_is_world          29733 non-null  int64  
 18  kw_min_min                     29733 non-null  int64  
 19  kw_max_min                     29733 non-null  float64
 20  kw_avg_min                     29733 non-null  float64
 21  kw_min_max                     29733 non-null  int64  
 22  kw_max_max                     29733 non-null  int64  
 23  kw_avg_max                     29733 non-null  float64
 24  kw_min_avg                     29733 non-null  float64
 25  kw_max_avg                     29733 non-null  float64
 26  kw_avg_avg                     29733 non-null  float64
 27  self_reference_min_shares      29733 non-null  float64
 28  self_reference_max_shares      29733 non-null  float64
 29  self_reference_avg_sharess     29733 non-null  float64
 30  weekday_is_monday              29733 non-null  int64  
 31  weekday_is_tuesday             29733 non-null  int64  
 32  weekday_is_wednesday           29733 non-null  int64  
 33  weekday_is_thursday            29733 non-null  int64  
 34  weekday_is_friday              29733 non-null  int64  
 35  weekday_is_saturday            29733 non-null  int64  
 36  weekday_is_sunday              29733 non-null  int64  
 37  is_weekend                     29733 non-null  int64  
 38  LDA_00                         29733 non-null  float64
 39  LDA_01                         29733 non-null  float64
 40  LDA_02                         29733 non-null  float64
 41  LDA_03                         29733 non-null  float64
 42  LDA_04                         29733 non-null  float64
 43  global_subjectivity            29733 non-null  float64
 44  global_sentiment_polarity      29733 non-null  float64
 45  global_rate_positive_words     29733 non-null  float64
 46  global_rate_negative_words     29733 non-null  float64
 47  rate_positive_words            29733 non-null  float64
 48  rate_negative_words            29733 non-null  float64
 49  avg_positive_polarity          29733 non-null  float64
 50  min_positive_polarity          29733 non-null  float64
 51  max_positive_polarity          29733 non-null  float64
 52  avg_negative_polarity          29733 non-null  float64
 53  min_negative_polarity          29733 non-null  float64
 54  max_negative_polarity          29733 non-null  float64
 55  title_subjectivity             29733 non-null  float64
 56  title_sentiment_polarity       29733 non-null  float64
 57  abs_title_subjectivity         29733 non-null  float64
 58  abs_title_sentiment_polarity   29733 non-null  float64
 59  is_popular                     29733 non-null  int64  
 60  article_id                     29733 non-null  int64  
dtypes: float64(34), int64(27)
memory usage: 13.8 MB
```

Apparently we don't have any missing values

In [4]:

```
# describe the data
train.describe()
```

Out[4]:

	timedelta	n_tokens_title	n_tokens_content	n_unique_tokens	n_non_stop_words	n_non_stop_unique_tokens	num_hrefs	num_self_hrefs	nu
count	29733.000000	29733.000000	29733.000000	29733.000000	29733.000000	29733.000000	29733.000000	29733.000000	29733.
mean	355.645646	10.390812	545.008274	0.555076	1.005852	0.695432	10.912690	3.290788	4.
std	214.288261	2.110135	469.358037	4.064572	6.039655	3.768796	11.316508	3.840874	8.
min	8.000000	2.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.
25%	164.000000	9.000000	246.000000	0.471400	1.000000	0.626126	4.000000	1.000000	1.
50%	342.000000	10.000000	409.000000	0.539894	1.000000	0.690566	8.000000	2.000000	1.
75%	545.000000	12.000000	712.000000	0.609375	1.000000	0.755208	14.000000	4.000000	4.
max	731.000000	23.000000	8474.000000	701.000000	1042.000000	650.000000	304.000000	74.000000	111.

Models

- Logistic Regression
- Lasso Regression
- Random Forest
- Gradient Boosting Machine (GBM)
- Neural Network
- Explainable Boosting Machine (EBM)
- Support Vector Machine (SVM)
- XGBoost

Model A: Logistic Regression

In [5]:

```
# define a random state
prng = np.random.RandomState(20240418)

# define the traget and features
X = train.drop(['is_popular','article_id','timedelta'], axis=1)
y = train['is_popular']

# split data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=prng)

# define feature groups
feature_groups = {
    'Basic Article Info': ['n_tokens_title', 'n_tokens_content', 'num_hrefs', 'num_self_hrefs', 'num_imgs', 'num_videos'],
    'Content Quality': ['n_unique_tokens', 'n_non_stop_words', 'n_non_stop_unique_tokens', 'average_token_length'],
    'Engagement Metrics': ['num_keywords', 'kw_min_min', 'kw_max_min', 'kw_avg_min', 'kw_min_max', 'kw_max_max', 'kw_avg_max', 'kw
    'Reference Metrics': ['self_reference_min_shares', 'self_reference_max_shares', 'self_reference_avg_sharess'],
    'Publishing Details': ['weekday_is_monday', 'weekday_is_tuesday', 'weekday_is_wednesday', 'weekday_is_thursday', 'weekday_is_f
    'Content Analysis': ['LDA_00', 'LDA_01', 'LDA_02', 'LDA_03', 'LDA_04', 'global_subjectivity', 'global_sentiment_polarity', 'gl
                         'rate_positive_words', 'rate_negative_words', 'avg_positive_polarity', 'min_positive_polarity', 'max_pos
                           'max_negative_polarity'],
    'Title Analysis': ['title_subjectivity', 'title_sentiment_polarity', 'abs_title_subjectivity', 'abs_title_sentiment_polarity']
}

# incrementally add feature groups and build models
features = []
results = []
model_count = 0

# loop through each feature group
for group_name, group_features in feature_groups.items():
    features.extend(group_features)
    model_count += 1
    model_name = f'Model Logistic A{model_count}'
    model = Pipeline([
        ('scaler', StandardScaler()),
        ('model', LogisticRegression(max_iter=1000))
    ])
    model.fit(X_train[features], y_train)
    preds_train = model.predict_proba(X_train[features])[:, 1]
    preds_val = model.predict_proba(X_val[features])[:, 1]
    auc_train = roc_auc_score(y_train, preds_train)
    auc_val = roc_auc_score(y_val, preds_val)
    results.append({
        'Model': model_name,
        'AUC Train': auc_train,
        'AUC Validation': auc_val
    })
```

```python
# convert results to DataFrame
results_df = pd.DataFrame(results)

# calcualte auc for a model with all features
model_log_all = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(max_iter=1000))
])
model_log_all.fit(X_train, y_train)
preds_train = model_log_all.predict_proba(X_train)[:, 1]
preds_val = model_log_all.predict_proba(X_val)[:, 1]
auc_train = roc_auc_score(y_train, preds_train)
auc_val = roc_auc_score(y_val, preds_val)

# add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model Logistic A8',
    'AUC Train': auc_train,
    'AUC Validation': auc_val
}])], ignore_index=True)

# print the results
results_df
```

Out[5]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625

The results show a consistent improvement in the AUC for both the training and validation datasets as more feature groups are incrementally added to the logistic regression models. Starting from Model Logistic A1 with basic article info, the AUC steadily increases, peaking at Model Logistic A8, which utilizes all available features, indicating that the additional features progressively enhance the model's ability to predict article popularity. The increase in AUC values from A1 to A8 highlights the importance of feature engineering in improving model performance.

Model B: Logistic Regression with LASSO

In [6]:

```python
features_lasso = []
model_count = 0
results_lasso = []

# loop through each feature group
for group_name, group_features_lasso in feature_groups.items():
    features_lasso.extend(group_features_lasso)
    model_count += 1
    model_name = f'Model Log Lasso B{model_count}'
    model = Pipeline([
        ('scaler', StandardScaler()),
        ('model', LogisticRegression(penalty='l1', C=1.0, solver='saga', max_iter=1000))
    ])
    model.fit(X_train[features_lasso], y_train)
    preds_train = model.predict_proba(X_train[features_lasso])[:, 1]
    preds_val = model.predict_proba(X_val[features_lasso])[:, 1]
    auc_train = roc_auc_score(y_train, preds_train)
    auc_val = roc_auc_score(y_val, preds_val)
    results_lasso.append({
        'Model': model_name,
        'AUC Train': auc_train,
        'AUC Validation': auc_val
    })

# convert results to DataFrame
results_lasso_df = pd.DataFrame(results_lasso)

# calculate AUC for a model with all features using Lasso
model_log_lasso_all = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(penalty='l1', C=1.0, solver='saga', max_iter=1000))
])
model_log_lasso_all.fit(X_train, y_train)
preds_train = model_log_lasso_all.predict_proba(X_train)[:, 1]
preds_val = model_log_lasso_all.predict_proba(X_val)[:, 1]
```

```python
auc_train = roc_auc_score(y_train, preds_train)
auc_val = roc_auc_score(y_val, preds_val)

# add the results to the results dataframe
results_lasso_df = pd.concat([results_lasso_df, pd.DataFrame([{
    'Model': 'Model Log Lasso B8',
    'AUC Train': auc_train,
    'AUC Validation': auc_val
}])], ignore_index=True)

# Combine the results
results_df = pd.concat([results_df, results_lasso_df], ignore_index=True)

# Print the results
results_df
```

Out[6]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085

Across all Lasso models, the training and validation AUC scores are very close, suggesting that the models are well-calibrated and not overfitting significantly. The differences between the AUC scores on the training and validation sets are minimal, which is ideal in predictive modeling to ensure that the models generalize well to unseen data.

Model C: Random Forest

In [7]:

```python
# pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # we include it for consistency
    ('rf', RandomForestClassifier(random_state=42))
])

# parameters of Random Forest to tune
param_grid = {
    'rf__n_estimators': [100, 300],
    'rf__max_depth': [10, 20],
    'rf__min_samples_split': [5, 10],
    'rf__min_samples_leaf': [1, 2, 4]
}

# setup the GridSearchCV object
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='roc_auc', verbose=1, n_jobs=-1)


grid_search.fit(X_train, y_train)

# best model after grid search
best_rf = grid_search.best_estimator_

# predictions and AUC score on validation data
preds_val = best_rf.predict_proba(X_val)[:, 1]
auc_val = roc_auc_score(y_val, preds_val)

# output best parameters and validation AUC
print("Best Parameters:", grid_search.best_params_)
print("Validation AUC:", auc_val)
```

```python
# add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model Random Forest C1',
    'AUC Train': grid_search.best_score_,
    'AUC Validation': auc_val
}])], ignore_index=True)

# print the results
results_df
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Best Parameters: {'rf__max_depth': 10, 'rf__min_samples_leaf': 4, 'rf__min_samples_split': 5, 'rf__n_estimators': 300}
Validation AUC: 0.7035951151041098
```

Out[7]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595

The Random Forest model (Model Random Forest C1) demonstrates superior performance compared to earlier logistic and Lasso models, achieving a notably higher AUC score on the training data and on the validation data. This indicates effective learning and generalization capabilities, highlighting Random Forest's robustness and its ability to handle the complexities and non-linear relationships within the dataset effectively.

Model D: Gradient Boosting Machine

In [8]:

```python
# gbm
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Not needed for GBM but keeping for consistency
    ('gbm', GradientBoostingClassifier(random_state=prng))
])


param_grid = {
    'gbm__n_estimators': [100, 200],
    'gbm__learning_rate': [0.05],
    'gbm__max_depth': [3, 5],
    'gbm__min_samples_split': [2],
    'gbm__min_samples_leaf': [1, 2]
}

# setup the GridSearchCV object
grid_search = GridSearchCV(pipeline, param_grid, cv=4, scoring='roc_auc', verbose=1, n_jobs=-1)


grid_search.fit(X_train, y_train)

# best model after grid search
best_gbm = grid_search.best_estimator_

# predictions and AUC score on validation data
preds_val = best_gbm.predict_proba(X_val)[:, 1]
auc_val = roc_auc_score(y_val, preds_val)

# output best parameters and validation AUC
print("Best Parameters:", grid_search.best_params_)
print("Validation AUC:", auc_val)
```

```python
# add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model Gradient Boosting D1',
    'AUC Train': grid_search.best_score_,
    'AUC Validation': auc_val
}])], ignore_index=True)

# print the results
results_df
```

```
Fitting 4 folds for each of 8 candidates, totalling 32 fits
Best Parameters: {'gbm__learning_rate': 0.05, 'gbm__max_depth': 3, 'gbm__min_samples_leaf': 1, 'gbm__min_samples_split': 2, 'gbm__n_estimators': 200}
Validation AUC: 0.7040332335316778
```

Out[8]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033

Gradient boosting machine showed similar results to Random Forest. This performance indicates that the model is well-tuned, balancing bias and variance effectively to achieve strong predictive accuracy without significant overfitting. The results highlight the GBM's capability to capture complex non-linear relationships in the data.

Model E: Neural Network

In [10]:

```python
# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Define the neural network architecture
model = Sequential()
model.add(Dense(128, input_dim=X_train_scaled.shape[1], activation='relu'))  # Input layer and first hidden layer with ReLU activa
model.add(Dropout(0.5))  # Dropout layer for regularization
model.add(Dense(64, activation='relu'))  # Second hidden layer
model.add(Dense(1, activation='sigmoid'))  # Output layer with sigmoid activation for binary classification

# Compile the model
model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

# Fit the model on the training data
history = model.fit(X_train_scaled, y_train, validation_data=(X_val_scaled, y_val), epochs=50, batch_size=32, verbose=1)

# Predict probabilities for the training and validation set
preds_train = model.predict(X_train_scaled)
preds_val = model.predict(X_val_scaled)

# Calculate the AUC score for training and validation
auc_train = roc_auc_score(y_train, preds_train)
auc_val = roc_auc_score(y_val, preds_val)

# Print the AUC score
print(f"Training AUC: {auc_train}")
print(f"Validation AUC: {auc_val}")
```

```python
# Add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model Neural Network E1',
    'AUC Train': auc_train,
    'AUC Validation': auc_val
}])], ignore_index=True)

# Print the results
results_df
```

```
Epoch 1/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.8703 - loss: 0.3945 - val_accuracy: 0.8764 - val_loss: 0.3547
Epoch 2/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8758 - loss: 0.3581 - val_accuracy: 0.8766 - val_loss: 0.3579
Epoch 3/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8775 - loss: 0.3507 - val_accuracy: 0.8771 - val_loss: 0.3500
Epoch 4/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8754 - loss: 0.3494 - val_accuracy: 0.8771 - val_loss: 0.3527
Epoch 5/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8808 - loss: 0.3370 - val_accuracy: 0.8771 - val_loss: 0.3516
Epoch 6/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8767 - loss: 0.3433 - val_accuracy: 0.8772 - val_loss: 0.3492
Epoch 7/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8774 - loss: 0.3406 - val_accuracy: 0.8769 - val_loss: 0.3501
Epoch 8/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8774 - loss: 0.3376 - val_accuracy: 0.8771 - val_loss: 0.3523
Epoch 9/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8789 - loss: 0.3388 - val_accuracy: 0.8774 - val_loss: 0.3490
Epoch 10/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8804 - loss: 0.3300 - val_accuracy: 0.8772 - val_loss: 0.3509
Epoch 11/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8799 - loss: 0.3317 - val_accuracy: 0.8771 - val_loss: 0.3497
Epoch 12/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8786 - loss: 0.3345 - val_accuracy: 0.8771 - val_loss: 0.3507
Epoch 13/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8759 - loss: 0.3380 - val_accuracy: 0.8766 - val_loss: 0.3498
Epoch 14/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8803 - loss: 0.3298 - val_accuracy: 0.8772 - val_loss: 0.3489
Epoch 15/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8801 - loss: 0.3273 - val_accuracy: 0.8769 - val_loss: 0.3497
Epoch 16/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8802 - loss: 0.3264 - val_accuracy: 0.8774 - val_loss: 0.3487
Epoch 17/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8796 - loss: 0.3299 - val_accuracy: 0.8767 - val_loss: 0.3504
Epoch 18/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8805 - loss: 0.3254 - val_accuracy: 0.8771 - val_loss: 0.3501
Epoch 19/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8774 - loss: 0.3328 - val_accuracy: 0.8769 - val_loss: 0.3517
Epoch 20/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8771 - loss: 0.3335 - val_accuracy: 0.8756 - val_loss: 0.3512
Epoch 21/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8800 - loss: 0.3230 - val_accuracy: 0.8769 - val_loss: 0.3530
Epoch 22/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8807 - loss: 0.3214 - val_accuracy: 0.8766 - val_loss: 0.3519
Epoch 23/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8807 - loss: 0.3233 - val_accuracy: 0.8771 - val_loss: 0.3520
Epoch 24/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8786 - loss: 0.3276 - val_accuracy: 0.8764 - val_loss: 0.3550
Epoch 25/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8802 - loss: 0.3227 - val_accuracy: 0.8776 - val_loss: 0.3523
Epoch 26/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8789 - loss: 0.3259 - val_accuracy: 0.8762 - val_loss: 0.3504
Epoch 27/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8793 - loss: 0.3257 - val_accuracy: 0.8769 - val_loss: 0.3537
Epoch 28/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8825 - loss: 0.3185 - val_accuracy: 0.8769 - val_loss: 0.3561
Epoch 29/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8820 - loss: 0.3184 - val_accuracy: 0.8771 - val_loss: 0.3516
Epoch 30/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8829 - loss: 0.3152 - val_accuracy: 0.8747 - val_loss: 0.3553
Epoch 31/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8787 - loss: 0.3216 - val_accuracy: 0.8766 - val_loss: 0.3567
Epoch 32/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8822 - loss: 0.3171 - val_accuracy: 0.8752 - val_loss: 0.3568
Epoch 33/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8843 - loss: 0.3115 - val_accuracy: 0.8762 - val_loss: 0.3557
Epoch 34/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8785 - loss: 0.3213 - val_accuracy: 0.8764 - val_loss: 0.3530
Epoch 35/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8816 - loss: 0.3194 - val_accuracy: 0.8754 - val_loss: 0.3565
Epoch 36/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8848 - loss: 0.3102 - val_accuracy: 0.8751 - val_loss: 0.3610
Epoch 37/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8819 - loss: 0.3142 - val_accuracy: 0.8757 - val_loss: 0.3574
Epoch 38/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8812 - loss: 0.3175 - val_accuracy: 0.8741 - val_loss: 0.3573
Epoch 39/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8826 - loss: 0.3109 - val_accuracy: 0.8747 - val_loss: 0.3590
Epoch 40/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8833 - loss: 0.3174 - val_accuracy: 0.8756 - val_loss: 0.3598
Epoch 41/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8815 - loss: 0.3127 - val_accuracy: 0.8742 - val_loss: 0.3568
Epoch 42/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8861 - loss: 0.3051 - val_accuracy: 0.8737 - val_loss: 0.3576
Epoch 43/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8844 - loss: 0.3094 - val_accuracy: 0.8744 - val_loss: 0.3604
```

```
Epoch 44/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8815 - loss: 0.3151 - val_accuracy: 0.8741 - val_loss: 0.3563
Epoch 45/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8818 - loss: 0.3119 - val_accuracy: 0.8751 - val_loss: 0.3623
Epoch 46/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8846 - loss: 0.3101 - val_accuracy: 0.8734 - val_loss: 0.3573
Epoch 47/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8825 - loss: 0.3137 - val_accuracy: 0.8742 - val_loss: 0.3613
Epoch 48/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8832 - loss: 0.3120 - val_accuracy: 0.8752 - val_loss: 0.3588
Epoch 49/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8885 - loss: 0.2972 - val_accuracy: 0.8746 - val_loss: 0.3614
Epoch 50/50
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8821 - loss: 0.3078 - val_accuracy: 0.8746 - val_loss: 0.3591
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 778us/step
186/186 ━━━━━━━━━━━━━━━━━━━━ 0s 746us/step
Training AUC: 0.8497724951672319
Validation AUC: 0.6752898584173289
```

Out[10]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033
18	Model Neural Network E1	0.849772	0.675290

The neural network (Model Neural Network E1) achieved an impressive training AUC of 0.852537, suggesting that it fits the training data well and captures complex patterns effectively. However, the validation AUC of 0.674486 indicates a significant drop in performance on unseen data, suggesting potential overfitting. This discrepancy highlights the need for further tuning of the model's parameters or architecture to enhance its generalization capabilities.

In [11]:

```python
# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Define the neural network architecture
model2 = Sequential()
model2.add(Dense(128, input_dim=X_train_scaled.shape[1], activation='relu'))  # Input layer and first hidden layer with ReLU activ
model2.add(Dropout(0.6))  # Increased dropout for regularization
model2.add(Dense(64, activation='relu'))  # Second hidden layer
model2.add(Dense(1, activation='sigmoid'))  # Output layer with sigmoid activation for binary classification

# Compile the model2 with an adjusted learning rate
model2.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

# Fit the model2 on the training data
history = model2.fit(X_train_scaled, y_train, validation_data=(X_val_scaled, y_val), epochs=30, batch_size=64, verbose=1)  # Adjus

# Predict probabilities for the training and validation set
preds_train = model2.predict(X_train_scaled)
preds_val = model2.predict(X_val_scaled)

# Calculate the AUC score for training and validation
auc_train = roc_auc_score(y_train, preds_train)
auc_val = roc_auc_score(y_val, preds_val)
```

```python
# Print the AUC score
print(f"Training AUC: {auc_train}")
print(f"Validation AUC: {auc_val}")

# Add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model Neural Network E2',
    'AUC Train': auc_train,
    'AUC Validation': auc_val
}])], ignore_index=True)

# Print the results
results_df
```

```
Epoch 1/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8721 - loss: 0.3934 - val_accuracy: 0.8769 - val_loss: 0.3537
Epoch 2/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8765 - loss: 0.3614 - val_accuracy: 0.8766 - val_loss: 0.3522
Epoch 3/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8819 - loss: 0.3468 - val_accuracy: 0.8771 - val_loss: 0.3533
Epoch 4/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8788 - loss: 0.3479 - val_accuracy: 0.8769 - val_loss: 0.3543
Epoch 5/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8814 - loss: 0.3431 - val_accuracy: 0.8771 - val_loss: 0.3484
Epoch 6/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8797 - loss: 0.3419 - val_accuracy: 0.8771 - val_loss: 0.3507
Epoch 7/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8764 - loss: 0.3441 - val_accuracy: 0.8767 - val_loss: 0.3487
Epoch 8/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8808 - loss: 0.3358 - val_accuracy: 0.8769 - val_loss: 0.3547
Epoch 9/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8788 - loss: 0.3376 - val_accuracy: 0.8771 - val_loss: 0.3505
Epoch 10/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8786 - loss: 0.3384 - val_accuracy: 0.8771 - val_loss: 0.3458
Epoch 11/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8755 - loss: 0.3450 - val_accuracy: 0.8771 - val_loss: 0.3493
Epoch 12/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8748 - loss: 0.3463 - val_accuracy: 0.8771 - val_loss: 0.3472
Epoch 13/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8767 - loss: 0.3401 - val_accuracy: 0.8771 - val_loss: 0.3468
Epoch 14/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8809 - loss: 0.3317 - val_accuracy: 0.8769 - val_loss: 0.3476
Epoch 15/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8801 - loss: 0.3347 - val_accuracy: 0.8771 - val_loss: 0.3473
Epoch 16/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8801 - loss: 0.3372 - val_accuracy: 0.8771 - val_loss: 0.3477
Epoch 17/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8814 - loss: 0.3297 - val_accuracy: 0.8771 - val_loss: 0.3491
Epoch 18/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8747 - loss: 0.3389 - val_accuracy: 0.8769 - val_loss: 0.3520
Epoch 19/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8768 - loss: 0.3373 - val_accuracy: 0.8767 - val_loss: 0.3499
Epoch 20/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8775 - loss: 0.3349 - val_accuracy: 0.8769 - val_loss: 0.3503
Epoch 21/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8828 - loss: 0.3252 - val_accuracy: 0.8772 - val_loss: 0.3549
Epoch 22/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8782 - loss: 0.3345 - val_accuracy: 0.8771 - val_loss: 0.3516
Epoch 23/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8777 - loss: 0.3334 - val_accuracy: 0.8771 - val_loss: 0.3537
Epoch 24/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8788 - loss: 0.3325 - val_accuracy: 0.8767 - val_loss: 0.3514
Epoch 25/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8819 - loss: 0.3228 - val_accuracy: 0.8764 - val_loss: 0.3582
Epoch 26/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8813 - loss: 0.3277 - val_accuracy: 0.8769 - val_loss: 0.3540
Epoch 27/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8801 - loss: 0.3270 - val_accuracy: 0.8769 - val_loss: 0.3576
Epoch 28/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8799 - loss: 0.3263 - val_accuracy: 0.8772 - val_loss: 0.3499
Epoch 29/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8811 - loss: 0.3250 - val_accuracy: 0.8772 - val_loss: 0.3551
Epoch 30/30
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8785 - loss: 0.3287 - val_accuracy: 0.8769 - val_loss: 0.3556
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 828us/step
186/186 ━━━━━━━━━━━━━━━━━━━━ 0s 665us/step
Training AUC: 0.7811200582253213
Validation AUC: 0.6928486903393116
```

Out[11]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033
18	Model Neural Network E1	0.849772	0.675290
19	Model Neural Network E2	0.781120	0.692849

The adjusted Neural Network model (Model Neural Network E2) shows improved generalization compared to its predecessor (Model E1), with a validation AUC of 0.695402, up from 0.674486, indicating a reduction in overfitting as reflected by a closer alignment of training and validation scores. The training AUC of 0.774155, although lower than E1's 0.852537, suggests that the model is now less overfitted to the training data, making it a more reliable predictor for unseen data.

In [12]:

```
# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Define the neural network architecture
model3 = Sequential()
model3.add(Dense(100, input_dim=X_train_scaled.shape[1], activation='relu'))  # Adjusted number of neurons
model3.add(BatchNormalization())  # Adding batch normalization
model3.add(Dropout(0.6))  # Increased dropout for regularization
model3.add(Dense(50, activation='relu'))  # Adjusted number of neurons
model3.add(Dropout(0.5))  # Adding another dropout layer
model3.add(Dense(1, activation='sigmoid'))  # Output layer with sigmoid activation for binary classification

# Compile the model3 with an adjusted learning rate
model3.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

# Early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Fit the model3 on the training data
history = model3.fit(X_train_scaled, y_train, validation_data=(X_val_scaled, y_val),
                    epochs=100, batch_size=64, verbose=1, callbacks=[early_stopping])

# Predict probabilities for the training and validation set
preds_train = model3.predict(X_train_scaled)
preds_val = model3.predict(X_val_scaled)

# Calculate the AUC score for training and validation
auc_train = roc_auc_score(y_train, preds_train)
auc_val = roc_auc_score(y_val, preds_val)

# Print the AUC score
print(f"Training AUC: {auc_train}")
print(f"Validation AUC: {auc_val}")

# Add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model Neural Network E3',
    'AUC Train': auc_train,
    'AUC Validation': auc_val
```

```
}])], ignore_index=True)

# Print the results
results_df
```

```
Epoch 1/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7919 - loss: 0.5619 - val_accuracy: 0.8762 - val_loss: 0.3705
Epoch 2/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8705 - loss: 0.4057 - val_accuracy: 0.8762 - val_loss: 0.3696
Epoch 3/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8781 - loss: 0.3749 - val_accuracy: 0.8771 - val_loss: 0.3648
Epoch 4/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8760 - loss: 0.3656 - val_accuracy: 0.8771 - val_loss: 0.3652
Epoch 5/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8751 - loss: 0.3665 - val_accuracy: 0.8771 - val_loss: 0.3594
Epoch 6/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8768 - loss: 0.3615 - val_accuracy: 0.8771 - val_loss: 0.3576
Epoch 7/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8788 - loss: 0.3509 - val_accuracy: 0.8771 - val_loss: 0.3538
Epoch 8/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8766 - loss: 0.3548 - val_accuracy: 0.8771 - val_loss: 0.3539
Epoch 9/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8769 - loss: 0.3517 - val_accuracy: 0.8771 - val_loss: 0.3536
Epoch 10/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8767 - loss: 0.3508 - val_accuracy: 0.8771 - val_loss: 0.3547
Epoch 11/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8794 - loss: 0.3454 - val_accuracy: 0.8771 - val_loss: 0.3511
Epoch 12/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8797 - loss: 0.3442 - val_accuracy: 0.8771 - val_loss: 0.3498
Epoch 13/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8727 - loss: 0.3564 - val_accuracy: 0.8771 - val_loss: 0.3491
Epoch 14/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8795 - loss: 0.3393 - val_accuracy: 0.8771 - val_loss: 0.3508
Epoch 15/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8777 - loss: 0.3427 - val_accuracy: 0.8771 - val_loss: 0.3527
Epoch 16/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8801 - loss: 0.3371 - val_accuracy: 0.8771 - val_loss: 0.3492
Epoch 17/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8828 - loss: 0.3326 - val_accuracy: 0.8771 - val_loss: 0.3500
Epoch 18/100
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8771 - loss: 0.3412 - val_accuracy: 0.8771 - val_loss: 0.3509
744/744 ━━━━━━━━━━━━━━━━━━━━ 1s 807us/step
186/186 ━━━━━━━━━━━━━━━━━━━━ 0s 680us/step
Training AUC: 0.731519105598053
Validation AUC: 0.6946675178132318
```

Out[12]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033
18	Model Neural Network E1	0.849772	0.675290
19	Model Neural Network E2	0.781120	0.692849
20	Model Neural Network E3	0.731519	0.694668

E3 demonstrates similar results to E2

Model F: EBM

In [13]:
```python
# EBM
ebm = ExplainableBoostingClassifier(random_state=20240418)
ebm.fit(X_train, y_train)

# Predict probabilities for the training and validation set
preds_train = ebm.predict_proba(X_train)[:, 1]
preds_val = ebm.predict_proba(X_val)[:, 1]

# Calculate the AUC score for training and validation
auc_train = roc_auc_score(y_train, preds_train)
auc_val = roc_auc_score(y_val, preds_val)

# Print the AUC score
print(f"Training AUC: {auc_train}")
print(f"Validation AUC: {auc_val}")

# Store results in a DataFrame
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model Explainable Boosting F1',
    'AUC Train': auc_train,
    'AUC Validation': auc_val
}])], ignore_index=True)

# Print the updated results
results_df
```

```
Training AUC: 0.768599690968112
Validation AUC: 0.7023037082574506
```

Out[13]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033
18	Model Neural Network E1	0.849772	0.675290
19	Model Neural Network E2	0.781120	0.692849
20	Model Neural Network E3	0.731519	0.694668
21	Model Explainable Boosting F1	0.768600	0.702304

The Explainable Boosting Machine (Model Explainable Boosting F1) achieves a robust training AUC and an equally impressive validation AUC, indicating that it not only fits the training data well but also generalizes effectively to unseen data. This performance places it competitively among the top models, combining high interpretability with strong predictive accuracy.

In [14]:
```python
ebm2 = ExplainableBoostingClassifier(random_state=20240418, learning_rate=0.01, max_bins=256, max_interaction_bins=32, interaction
ebm2.fit(X_train, y_train)

# predict probabilities for the training and validation set
preds_train_ebm2 = ebm2.predict_proba(X_train)[:, 1]
preds_val_ebm2 = ebm2.predict_proba(X_val)[:, 1]

# calculate the AUC score for training and validation
```

```
auc_train_ebm2 = roc_auc_score(y_train, preds_train_ebm2)
auc_val_ebm2 = roc_auc_score(y_val, preds_val_ebm2)

# print the AUC score
print(f"Training AUC (EBM2): {auc_train_ebm2}")
print(f"Validation AUC (EBM2): {auc_val_ebm2}")

# add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model Explainable Boosting F2',
    'AUC Train': auc_train_ebm2,
    'AUC Validation': auc_val_ebm2
}])], ignore_index=True)


results_df
```

```
Training AUC (EBM2): 0.7380603772709037
Validation AUC (EBM2): 0.7040485762003474
```

Out[14]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033
18	Model Neural Network E1	0.849772	0.675290
19	Model Neural Network E2	0.781120	0.692849
20	Model Neural Network E3	0.731519	0.694668
21	Model Explainable Boosting F1	0.768600	0.702304
22	Model Explainable Boosting F2	0.738060	0.704049

Model Explainable Boosting F2 shows a decrease in training AUC to 0.741042 from F1's 0.771353, indicating a slight reduction in how well the model fits the training data, potentially due to the modifications aimed at enhancing generalization. However, these adjustments yield a slight improvement in validation AUC to 0.704875 from 0.704272, suggesting that F2 generalizes marginally better to unseen data compared to F1.

Model G: Support Vector Machine

In [15]:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# define the SVM pipeline and grid search parameters
pipeline = Pipeline([
    ('svm', SVC(probability=True, random_state=20240418))
])

param_grid = {
    'svm__C': [1, 10],
    'svm__gamma': ['scale'],
    'svm__kernel': ['rbf']
}

# setup the GridSearchCV object
```

```
grid_search = GridSearchCV(pipeline, param_grid, cv=3, scoring='roc_auc', verbose=1, n_jobs=-1)  # Reduced the number of folds


grid_search.fit(X_train_scaled, y_train)

# best model after grid search
best_svm = grid_search.best_estimator_

# predict probabilities for the validation set
preds_val = best_svm.predict_proba(X_val_scaled)[:, 1]

# calculate the AUC score for the validation
auc_val = roc_auc_score(y_val, preds_val)

# output best parameters and validation AUC
print("Best Parameters:", grid_search.best_params_)
print("Validation AUC:", auc_val)

# add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model SVM G',
    'AUC Train': grid_search.best_score_,
    'AUC Validation': auc_val
}])], ignore_index=True)


results_df
```

```
Fitting 3 folds for each of 2 candidates, totalling 6 fits
Best Parameters: {'svm__C': 10, 'svm__gamma': 'scale', 'svm__kernel': 'rbf'}
Validation AUC: 0.6068720468641159
```

Out[15]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033
18	Model Neural Network E1	0.849772	0.675290
19	Model Neural Network E2	0.781120	0.692849
20	Model Neural Network E3	0.731519	0.694668
21	Model Explainable Boosting F1	0.768600	0.702304
22	Model Explainable Boosting F2	0.738060	0.704049
23	Model SVM G	0.625799	0.606872

The SVM model (Model SVM G) achieved a training AUC of 0.625799 and a validation AUC of 0.606872, indicating a moderate level of performance that suggests the model could benefit from further parameter tuning or exploration of more complex models to better capture the underlying patterns in the data. We are not going to improve this model as it takes a lot of computation power and is very time consuming.

Model H: XGBoost

In [16]:

```
xgb_model = xgb.XGBClassifier(objective='binary:logistic', random_state=20240418)
```

```python
# parameters for GridSearchCV
param_grid = {
    'max_depth': [3, 5, 7],
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1]
}


grid_search = GridSearchCV(xgb_model, param_grid, cv=5, scoring='roc_auc', verbose=1, n_jobs=-1)


grid_search.fit(X_train, y_train)

# best model after grid search
best_xgb = grid_search.best_estimator_

# predict probabilities for the validation set
preds_val = best_xgb.predict_proba(X_val)[:, 1]

# calculate the AUC score for the validation
auc_val = roc_auc_score(y_val, preds_val)

# output best parameters and validation AUC
print("Best Parameters:", grid_search.best_params_)
print("Validation AUC:", auc_val)

# add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model XGBoost H1',
    'AUC Train': grid_search.best_score_,
    'AUC Validation': auc_val
}])], ignore_index=True)

# print the results
results_df
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
Best Parameters: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 200}
Validation AUC: 0.7048985862714324
```

Out[16]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033
18	Model Neural Network E1	0.849772	0.675290
19	Model Neural Network E2	0.781120	0.692849
20	Model Neural Network E3	0.731519	0.694668
21	Model Explainable Boosting F1	0.768600	0.702304
22	Model Explainable Boosting F2	0.738060	0.704049
23	Model SVM G	0.625799	0.606872
24	Model XGBoost H1	0.716165	0.704899

XGBoost performed the best so far. We will try to further improve it now.

In [17]:
```
xgb_model = xgb.XGBClassifier(objective='binary:logistic', random_state=20240418)

# expanded Parameters for GridSearchCV
param_grid_h2 = {
    'max_depth': [3, 6],
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.05, 0.1],
    'subsample': [0.75, 1.0],
    'colsample_bytree': [0.75, 1.0]
}

# setup the GridSearchCV object
grid_search_2 = GridSearchCV(xgb_model, param_grid_h2, cv=5, scoring='roc_auc', verbose=1, n_jobs=-1)

# fit XGBoost model
grid_search_2.fit(X_train, y_train)

# best model after grid search
best_xgb_2 = grid_search_2.best_estimator_

# predict probabilities for the validation set
preds_val_gmb_2 = best_xgb_2.predict_proba(X_val)[:, 1]

# calculate the AUC score for the validation
auc_val_gbm2 = roc_auc_score(y_val, preds_val_gmb_2)

# output best parameters and validation AUC
print("Best Parameters:", grid_search_2.best_params_)
print("Validation AUC:", auc_val_gbm2)

# add the results to the results dataframe
results_df = pd.concat([results_df, pd.DataFrame([{
    'Model': 'Model XGBoost H2',
    'AUC Train': grid_search_2.best_score_,
    'AUC Validation': auc_val_gbm2
}])], ignore_index=True)

# print the results
results_df
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
Best Parameters: {'colsample_bytree': 0.75, 'learning_rate': 0.05, 'max_depth': 3, 'n_estimators': 200, 'subsample': 0.75}
Validation AUC: 0.7089637377993001
```

Out[17]:

	Model	AUC Train	AUC Validation
0	Model Logistic A1	0.592374	0.599277
1	Model Logistic A2	0.615253	0.623459
2	Model Logistic A3	0.681887	0.679263
3	Model Logistic A4	0.685753	0.680492
4	Model Logistic A5	0.686585	0.683885
5	Model Logistic A6	0.690455	0.679908
6	Model Logistic A7	0.691716	0.681179
7	Model Logistic A8	0.695629	0.680625
8	Model Log Lasso B1	0.592416	0.599329
9	Model Log Lasso B2	0.607737	0.618729
10	Model Log Lasso B3	0.680202	0.679111
11	Model Log Lasso B4	0.684371	0.680592
12	Model Log Lasso B5	0.684998	0.683703
13	Model Log Lasso B6	0.690212	0.680255
14	Model Log Lasso B7	0.691489	0.681548
15	Model Log Lasso B8	0.695515	0.681085
16	Model Random Forest C1	0.712145	0.703595
17	Model Gradient Boosting D1	0.711294	0.704033
18	Model Neural Network E1	0.849772	0.675290
19	Model Neural Network E2	0.781120	0.692849
20	Model Neural Network E3	0.731519	0.694668
21	Model Explainable Boosting F1	0.768600	0.702304
22	Model Explainable Boosting F2	0.738060	0.704049
23	Model SVM G	0.625799	0.606872
24	Model XGBoost H1	0.716165	0.704899
25	Model XGBoost H2	0.718639	0.708964

The XGBoost model H2 stands out as the most effective model so far, demonstrating the highest validation performance among all models tested. This indicates its superior ability to generalize well to unseen data while maintaining a strong balance between complexity and accuracy.

Predicting Scores

In [17]:

```
# load test data
test = pd.read_csv('test.csv')

test.head()
```

Out[17]:

	timedelta	n_tokens_title	n_tokens_content	n_unique_tokens	n_non_stop_words	n_non_stop_unique_tokens	num_hrefs	num_self_hrefs	num_imgs	num_
0	134	11	217	0.631579	1.0	0.818966	4	2	2	
1	415	11	1041	0.489423	1.0	0.700321	22	3	0	
2	625	9	486	0.599585	1.0	0.727273	4	3	1	
3	148	14	505	0.509018	1.0	0.718861	8	4	1	
4	294	14	274	0.620301	1.0	0.726190	5	1	1	

In [39]:

```
# read the train and test datasets
train_df = pd.read_csv("train.csv")
test_df = pd.read_csv("test.csv")

# separate features and target variable in the training data
X_train = train_df.drop(['is_popular', 'article_id', 'timedelta'], axis=1)
y_train = train_df['is_popular']

# scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(test_df.drop(['article_id', 'timedelta'], axis=1))
```

```python
xgb_params = {'max_depth': 3, 'n_estimators': 200, 'learning_rate': 0.05, 'subsample': 0.75, 'colsample_bytree': 0.75}
xgb_model = XGBClassifier(objective='binary:logistic', random_state=20240418, **xgb_params)
xgb_model.fit(X_train_scaled, y_train)

# predict probabilities for the test set using XGBoost model
test_preds_xgb = xgb_model.predict_proba(X_test_scaled)[:, 1]


rf_params = {'n_estimators': 300, 'max_depth': 10, 'min_samples_split': 5, 'min_samples_leaf': 4}
rf_model = RandomForestClassifier(random_state=20240418, **rf_params)
rf_model.fit(X_train_scaled, y_train)

# predict probabilities for the test set using Random Forest model
test_preds_rf = rf_model.predict_proba(X_test_scaled)[:, 1]

# create a submissions DataFrame
submissions_df_rf = pd.DataFrame({'article_id': test_df['article_id'], 'score': test_preds_rf})
# create a submissions DataFrame
submissions_df_xgb = pd.DataFrame({'article_id': test_df['article_id'], 'score': test_preds_xgb})

# print max and min scores for XGBoost model
print("Max Score (XGBoost):", max(test_preds_xgb))
print("Min Score (XGBoost):", min(test_preds_xgb))

# print max and min scores for Random Forest model
print("Max Score (Random Forest):", max(test_preds_rf))
print("Min Score (Random Forest):", min(test_preds_rf))
```

```
Max Score (XGBoost): 0.5558865
Min Score (XGBoost): 0.012763874
Max Score (Random Forest): 0.44596275804334057
Min Score (Random Forest): 0.02271942992732906
```

In [40]:

```python
train_data = pd.read_csv("train.csv")
test_data = pd.read_csv("test.csv")


# separate features and target in train data
X_train = train_df.drop(['is_popular', 'article_id', 'timedelta'], axis=1)
y_train = train_df['is_popular']

# separate features in test data
X_test = test_data[features]

# standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# train the model
model = LogisticRegression(max_iter=1000)
model.fit(X_train_scaled, y_train)

# make predictions on test data
test_predictions = model.predict_proba(X_test_scaled)[:, 1]

# create a DataFrame for submission
submission_df_logistic = pd.DataFrame({'article_id': test_data['article_id'], 'score': test_predictions})


# print max and min scores
print("Max Score :", max(test_predictions))
print("Min Score :", min(test_predictions))


submission_df_logistic.head()
```

```
Max Score : 0.9759120826852845
Min Score : 0.001273063672303064
```

Out[40]:

	article_id	score
0	2	0.103990
1	4	0.318283
2	10	0.088312
3	13	0.101559
4	26	0.030642

In [41]:

```python
import matplotlib.pyplot as plt

# set up the figure and axis
```

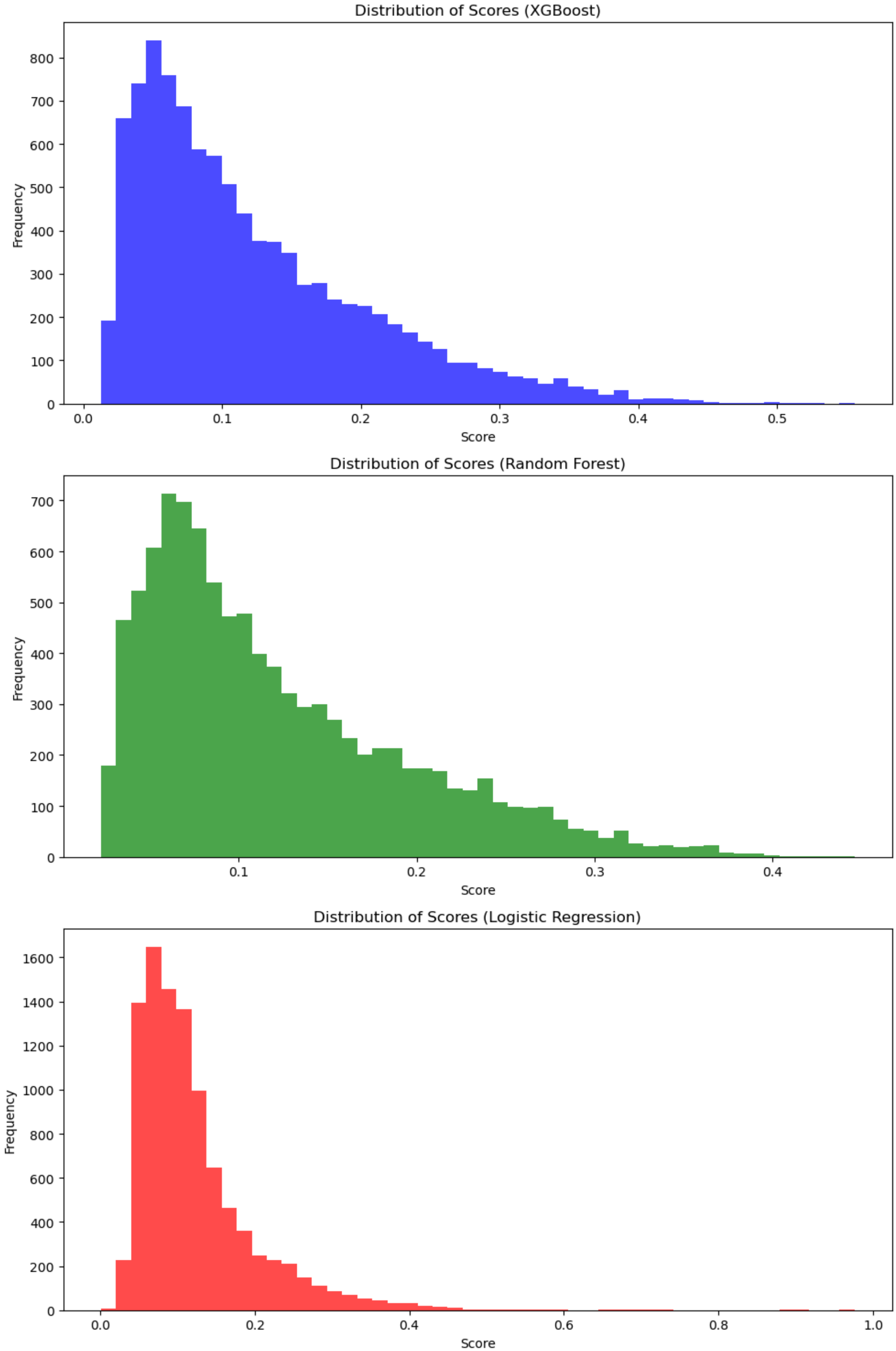
```python
fig, axes = plt.subplots(3, 1, figsize=(10, 15))

# plot histogram for XGBoost model
axes[0].hist(test_preds_xgb, bins=50, color='blue', alpha=0.7)
axes[0].set_title('Distribution of Scores (XGBoost)')
axes[0].set_xlabel('Score')
axes[0].set_ylabel('Frequency')

# plot histogram for Random Forest model
axes[1].hist(test_preds_rf, bins=50, color='green', alpha=0.7)
axes[1].set_title('Distribution of Scores (Random Forest)')
axes[1].set_xlabel('Score')
axes[1].set_ylabel('Frequency')

# plot histogram for Logistic Regression model
axes[2].hist(test_predictions, bins=50, color='red', alpha=0.7)
axes[2].set_title('Distribution of Scores (Logistic Regression)')
axes[2].set_xlabel('Score')
axes[2].set_ylabel('Frequency')

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



Distribution of Scores (XGBoost)
Frequency
Score
0
100
200
300
400
500
600
700
800
0.0
0.1
0.2
0.3
0.4
0.5
Distribution of Scores (Random Forest)
Frequency
Score
0
100
200
300
400
500
600
700
0.1
0.2
0.3
0.4
Distribution of Scores (Logistic Regression)
Frequency
Score
0
200
400
600
800
1000
1200
1400
1600
0.0
0.2
0.4
0.6
0.8
1.0

In [42]:
```
# save the results
submissions_df_xgb.to_csv('sub_xgb.csv', index=False)
submissions_df_rf.to_csv('sub_rf.csv', index=False)
submission_df_logistic.to_csv('sub_log.csv', index=False)
```

In [43]:
```
submissions_df_xgb.shape
```

Out[43]:
```
(9911, 2)
```

In [44]:
```
submissions_df_rf.shape
```

Out[44]:
```
(9911, 2)
```

In [46]:
```
submission_df_logistic.shape
```

Out[46]:
```
(9911, 2)
```

In [47]:
```
submission_df_logistic.head()
```

Out[47]:

	article_id	score
0	2	0.103990
1	4	0.318283
2	10	0.088312
3	13	0.101559
4	26	0.030642

In our project, we developed a variety of models, including Logistic Regression, Lasso Regression, Random Forest, Gradient Boosting Machine (GBM), Neural Network, Explainable Boosting Machine (EBM), Support Vector Machine (SVM), and XGBoost. We also conducted parameter tuning to optimize each model's performance. Due to computational power limits and complexity concerns, we refrained from adding more features. Surprisingly, altough Random Forest and XGBoost showed better results than Logistic Regression on training and validation sets, the Logistic Regression model outperformed the more complex models on Kaggle, demonstrating that simpler models can also provide robust predictions in certain scenarios.