brandenburg-kaggle

April 13, 2024

#

Kaggle Competition Assignment

Ian Brandenburg (2304791)

GitHub Repo

0.1 # Introduction

This kaggle competition is aimed at developing models for binary classification. Machine Learning Tools are encorporated, with the primary metric being the AUC score. The RMSLE score is also used in this project to assist in determining the best models, while the AUC is the primary metric.

The objective to determine of the Mashable article is classified as popular or not. The articles themselves are not used, but statistics and variables related to these articles are used. The Kaggle competition provides a training (29,733 rows) and test set (9,911 rows). A total of 61 columns are included in the training set, one of them being `is_popular`, which is the target variable. `timedelta` and `article_id` were dropped from the models since they do not particularly relate to the predictions. The training set was then split into a training set and validation set at 20% going to the validation set.

The dataset provided seemed to already be cleaned after analyzing the exploratory anylsis. Additionally, many binary variables had already been developed. However, the dataset underwent an enhancement process where feature engineering techniques were applied to the variables. This included generating interaction features, as well as applying polynomial transformations such as squaring and cubing, to uncover nonlinear relationships and improve the model's predictive capabilities.

Many models were tested to determine the best models to predict if an article `is_popular`. Additionally, models that were not specifically covered during the course were also included to test out new methods. These models included: - Logistic Regression - Lasso - Stacking Model (Included Decision Tree, Random Forest, and XGB) - Deicision Tree Classifier - Random Forest - Gradient Boosted Random Forest - Light Gradient Boosting - Cat Boosting - Explainable Boosting Machine - Neural Network Models

These models experiement with different parameters and settings to attempt to find the best predictive models. Furthermore, GridSearchCV was used with certain models to determine the best parameters for those specific models. In the end, the most effective model based on the AUC metric was consistently the Explainable Boosting Machine.

0.2 ### Import Libraries

[1]:
```
# General utilities
import numpy as np
import pandas as pd
import time
import os
import warnings
from itertools import combinations
import matplotlib.pyplot as plt

# Sklearn model selection, preprocessing, metrics, and ensemble methods
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import roc_auc_score
from sklearn.linear_model import LogisticRegression, LassoCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, StackingClassifier
from sklearn.inspection import permutation_importance

# Sklearn pipeline utilities
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline, make_pipeline

# XGBoost
import xgboost as xgb

# Cat Boost Classifier
from catboost import CatBoostClassifier

# Light GBM
import lightgbm as lgb

# InterpretML for explainable boosting
from interpret.glassbox import ExplainableBoostingClassifier

# TensorFlow and Keras for neural networks
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Conv1D, MaxPooling1D,
 ↪Flatten, BatchNormalization
from tensorflow.keras.metrics import AUC
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam

# Suppress warnings
warnings.filterwarnings('ignore')
```

0.3 # Data Wrangling

0.4 ## Data Import

The data was directly imported from GitHub after being downloaded into the GitHub Repo.

[2]:
```
train_data = pd.read_csv("https://raw.githubusercontent.com/Iandrewburg/
↪Data_Science/main/Data_Science_2/Assignments/Take_Home_Final/train.csv")
train_data.head()
```

[2]:
```
   timedelta  n_tokens_title  n_tokens_content  n_unique_tokens  \
0        594               9               702         0.454545
1        346               8              1197         0.470143
2        484               9               214         0.618090
3        639               8               249         0.621951
4        177              12              1219         0.397841

   n_non_stop_words  n_non_stop_unique_tokens  num_hrefs  num_self_hrefs  \
0               1.0                  0.620438         11               2
1               1.0                  0.666209         21               6
2               1.0                  0.748092          5               2
3               1.0                  0.664740         16               5
4               1.0                  0.583578         21               1

   num_imgs  num_videos  …  max_positive_polarity  avg_negative_polarity  \
0         1           0  …               1.000000              -0.153395
1         2          13  …               1.000000              -0.308167
2         1           0  …               0.433333              -0.141667
3         8           0  …               0.500000              -0.500000
4         1           2  …               0.800000              -0.441111

   min_negative_polarity  max_negative_polarity  title_subjectivity  \
0                   -0.4                  -0.10                 0.0
1                   -1.0                  -0.10                 0.0
2                   -0.2                  -0.05                 0.0
3                   -0.8                  -0.40                 0.0
4                   -1.0                  -0.05                 0.0

   title_sentiment_polarity  abs_title_subjectivity  \
0                       0.0                     0.5
1                       0.0                     0.5
2                       0.0                     0.5
3                       0.0                     0.5
4                       0.0                     0.5

   abs_title_sentiment_polarity  is_popular  article_id
0                           0.0           0           1
1                           0.0           0           3
```

```
2                          0.0           0            5
3                          0.0           0            6
4                          0.0           0            7

[5 rows x 61 columns]
```

[3]:
```
test_data = pd.read_csv("https://raw.githubusercontent.com/Iandrewburg/
 ↪Data_Science/main/Data_Science_2/Assignments/Take_Home_Final/test.csv")
test_data.head()
```

[3]:
```
   timedelta  n_tokens_title  n_tokens_content  n_unique_tokens  \
0        134              11               217         0.631579
1        415              11              1041         0.489423
2        625               9               486         0.599585
3        148              14               505         0.509018
4        294              14               274         0.620301

   n_non_stop_words  n_non_stop_unique_tokens  num_hrefs  num_self_hrefs  \
0               1.0                  0.818966          4               2
1               1.0                  0.700321         22               3
2               1.0                  0.727273          4               3
3               1.0                  0.718861          8               4
4               1.0                  0.726190          5               1

   num_imgs  num_videos  …  min_positive_polarity  max_positive_polarity  \
0         2           0  …               0.136364                    0.5
1         0          14  …               0.050000                    1.0
2         1           0  …               0.062500                    0.7
3         1           1  …               0.100000                    1.0
4         1           0  …               0.100000                    0.6

   avg_negative_polarity  min_negative_polarity  max_negative_polarity  \
0              -0.170370              -0.200000              -0.155556
1              -0.426268              -1.000000              -0.100000
2              -0.387821              -1.000000              -0.050000
3              -0.284722              -0.400000              -0.050000
4              -0.333333              -0.333333              -0.333333

   title_subjectivity  title_sentiment_polarity  abs_title_subjectivity  \
0            0.288889                 -0.155556                0.211111
1            0.975000                  0.300000                0.475000
2            0.000000                  0.000000                0.500000
3            0.000000                  0.000000                0.500000
4            0.000000                  0.000000                0.500000

   abs_title_sentiment_polarity  article_id
0                      0.155556           2
```

```
1                         0.300000           4
2                         0.000000          10
3                         0.000000          13
4                         0.000000          26

[5 rows x 60 columns]
```

[4]:
```python
test_data.columns
```

[4]:
```
Index(['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_videos', 'average_token_length',
       'num_keywords', 'data_channel_is_lifestyle',
       'data_channel_is_entertainment', 'data_channel_is_bus',
       'data_channel_is_socmed', 'data_channel_is_tech',
       'data_channel_is_world', 'kw_min_min', 'kw_max_min', 'kw_avg_min',
       'kw_min_max', 'kw_max_max', 'kw_avg_max', 'kw_min_avg', 'kw_max_avg',
       'kw_avg_avg', 'self_reference_min_shares', 'self_reference_max_shares',
       'self_reference_avg_sharess', 'weekday_is_monday', 'weekday_is_tuesday',
       'weekday_is_wednesday', 'weekday_is_thursday', 'weekday_is_friday',
       'weekday_is_saturday', 'weekday_is_sunday', 'is_weekend', 'LDA_00',
       'LDA_01', 'LDA_02', 'LDA_03', 'LDA_04', 'global_subjectivity',
       'global_sentiment_polarity', 'global_rate_positive_words',
       'global_rate_negative_words', 'rate_positive_words',
       'rate_negative_words', 'avg_positive_polarity', 'min_positive_polarity',
       'max_positive_polarity', 'avg_negative_polarity',
       'min_negative_polarity', 'max_negative_polarity', 'title_subjectivity',
       'title_sentiment_polarity', 'abs_title_subjectivity',
       'abs_title_sentiment_polarity', 'article_id'],
      dtype='object')
```

0.5 ## Exploratory Data Analysis

0.6 ### Variable Descriptions

- **`is_popular`: Whether or not the article was among the most popular ones based on shares on social media**
- *`article_id`: Unique identifier of the article*
- *`timedelta`: Days between the article publication and the dataset acquisition (non-predictive)*
- `n_tokens_title`: Number of words in the title
- `n_tokens_content`: Number of words in the content
- `n_unique_tokens`: Rate of unique words in the content
- `n_non_stop_words`: Rate of non-stop words in the content
- `n_non_stop_unique_tokens`: Rate of unique non-stop words in the content
- `num_hrefs`: Number of links
- `num_self_hrefs`: Number of links to other articles published by Mashable
- `num_imgs`: Number of images
- `num_videos`: Number of videos

- `average_token_length`: Average length of the words in the content
- `num_keywords`: Number of keywords in the metadata
- `data_channel_is_lifestyle`: Is data channel 'Lifestyle'?
- `data_channel_is_entertainment`: Is data channel 'Entertainment'?
- `data_channel_is_bus`: Is data channel 'Business'?
- `data_channel_is_socmed`: Is data channel 'Social Media'?
- `data_channel_is_tech`: Is data channel 'Tech'?
- `data_channel_is_world`: Is data channel 'World'?
- `kw_min_min`: Worst keyword (min. shares)
- `kw_max_min`: Worst keyword (max. shares)
- `kw_avg_min`: Worst keyword (avg. shares)
- `kw_min_max`: Best keyword (min. shares)
- `kw_max_max`: Best keyword (max. shares)
- `kw_avg_max`: Best keyword (avg. shares)
- `kw_min_avg`: Avg. keyword (min. shares)
- `kw_max_avg`: Avg. keyword (max. shares)
- `kw_avg_avg`: Avg. keyword (avg. shares)
- `self_reference_min_shares`: Min. shares of referenced articles in Mashable
- `self_reference_max_shares`: Max. shares of referenced articles in Mashable
- `self_reference_avg_sharess`: Avg. shares of referenced articles in Mashable
- `weekday_is_monday`: Was the article published on a Monday?
- `weekday_is_tuesday`: Was the article published on a Tuesday?
- `weekday_is_wednesday`: Was the article published on a Wednesday?
- `weekday_is_thursday`: Was the article published on a Thursday?
- `weekday_is_friday`: Was the article published on a Friday?
- `weekday_is_saturday`: Was the article published on a Saturday?
- `weekday_is_sunday`: Was the article published on a Sunday?
- `is_weekend`: Was the article published on the weekend?
- `LDA_00`: Closeness to LDA topic 0
- `LDA_01`: Closeness to LDA topic 1
- `LDA_02`: Closeness to LDA topic 2
- `LDA_03`: Closeness to LDA topic 3
- `LDA_04`: Closeness to LDA topic 4
- `global_subjectivity`: Text subjectivity
- `global_sentiment_polarity`: Text sentiment polarity
- `global_rate_positive_words`: Rate of positive words in the content
- `global_rate_negative_words`: Rate of negative words in the content
- `rate_positive_words`: Rate of positive words among non-neutral tokens
- `rate_negative_words`: Rate of negative words among non-neutral tokens
- `avg_positive_polarity`: Avg. polarity of positive words
- `min_positive_polarity`: Min. polarity of positive words
- `max_positive_polarity`: Max. polarity of positive words
- `avg_negative_polarity`: Avg. polarity of negative words
- `min_negative_polarity`: Min. polarity of negative words
- `max_negative_polarity`: Max. polarity of negative words
- `title_subjectivity`: Title subjectivity
- `title_sentiment_polarity`: Title polarity
- `abs_title_subjectivity`: Absolute subjectivity level

- `abs_title_sentiment_polarity`: Absolute polarity level

[5]:
```
train_data.describe()
```

[5]:
```
          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
25%      164.000000        9.000000        246.000000         0.471400
50%      342.000000       10.000000        409.000000         0.539894
75%      545.000000       12.000000        712.000000         0.609375
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
25%            1.000000                  0.626126      4.000000
50%            1.000000                  0.690566      8.000000
75%            1.000000                  0.755208     14.000000
max         1042.000000                650.000000    304.000000

       num_self_hrefs      num_imgs    num_videos  …  max_positive_polarity  \
count    29733.000000  29733.000000  29733.000000  …           29733.000000
mean         3.290788      4.524535      1.263546  …               0.757780
std          3.840874      8.213823      4.189080  …               0.247293
min          0.000000      0.000000      0.000000  …               0.000000
25%          1.000000      1.000000      0.000000  …               0.600000
50%          2.000000      1.000000      0.000000  …               0.800000
75%          4.000000      4.000000      1.000000  …               1.000000
max         74.000000    111.000000     91.000000  …               1.000000

       avg_negative_polarity  min_negative_polarity  max_negative_polarity  \
count           29733.000000           29733.000000           29733.000000
mean               -0.259709              -0.520981              -0.107793
std                 0.128488               0.290454               0.095672
min                -1.000000              -1.000000              -1.000000
25%                -0.328704              -0.700000              -0.125000
50%                -0.252827              -0.500000              -0.100000
75%                -0.186494              -0.300000              -0.050000
max                 0.000000               0.000000               0.000000

       title_subjectivity  title_sentiment_polarity  abs_title_subjectivity  \
count        29733.000000              29733.000000            29733.000000
mean             0.281878                  0.069691                0.341427
```

```
std           0.323461                   0.264379                0.188735
min           0.000000                  -1.000000                0.000000
25%           0.000000                   0.000000                0.166667
50%           0.144444                   0.000000                0.500000
75%           0.500000                   0.136364                0.500000
max           1.000000                   1.000000                0.500000

       abs_title_sentiment_polarity    is_popular    article_id
count                  29733.000000  29733.000000  29733.000000
mean                       0.155234      0.121649  19834.913530
std                        0.225066      0.326886  11432.376037
min                        0.000000      0.000000      1.000000
25%                        0.000000      0.000000   9965.000000
50%                        0.000000      0.000000  19859.000000
75%                        0.250000      0.000000  29742.000000
max                        1.000000      1.000000  39643.000000

[8 rows x 61 columns]
```

After inspection, there do seem to be some variables with extreme values. However, these extreme values were considered important for the analysis and left in.

[6]:
```
print(f"The shape of the training set is {train_data.shape[0]} rows, and␣
↪{train_data.shape[1]} columns.")
```

```
The shape of the training set is 29733 rows, and 61 columns.
```

[7]:
```
total_missing_values = train_data.isnull().sum()[train_data.isnull().sum() > 0].
↪sum()
print(f"There are a total of {total_missing_values} missing values in the␣
↪dataset.")
```

```
There are a total of 0 missing values in the dataset.
```

The dataset does not have any missing values, so we can move on to the feature engineering. This suggests that the dataset was already cleaned prior to importing the data.

0.7 ## Feature Engineering

0.8 ### Defining Variable Groups

Here, the variables are split up into groups for the purpose of incremental model testing with different variable groups. This will help us see if certain variable groups perform better than others. There are 7 primary variable groups listed here: - `basic_text_features` - `content_properties` - `keyword_performance` - `self_reference_metrics` - `publication_timing` - `content_topic_and_sentiment` - `title_sentiment`

Certain variables included in the initial dataset were dropped from these variable groups as some are categorical variables that were transformed into binary variables, and a baseline variable is needed. From the `publication_timing` group, `weekday_is_monday` and `is_weekend` were dropped to avoid autocorrelation. From the `content_topic_and_sentiment` variable group, `data_channel_is_lifestyle` was dropped for the same purpose.

[9]:
```
# Defining variable groups
basic_text_features = ['n_tokens_title',
                       'n_tokens_content',
                       'n_unique_tokens',
                       'n_non_stop_words',
                       'n_non_stop_unique_tokens',
                       'average_token_length',
                       'num_keywords']
content_properties = ['num_hrefs',
                      'num_self_hrefs',
                      'num_imgs',
                      'num_videos',
                      'global_subjectivity',
                      'global_sentiment_polarity',
                      'global_rate_positive_words',
                      'global_rate_negative_words']
keyword_performance = ['kw_min_min',
                       'kw_max_min',
                       'kw_avg_min',
                       'kw_min_max',
                       'kw_max_max',
                       'kw_avg_max',
                       'kw_min_avg',
                       'kw_max_avg',
                       'kw_avg_avg']
self_reference_metrics = ['self_reference_min_shares',
                          'self_reference_max_shares',
                          'self_reference_avg_sharess']

# dropped 'weekday_is_monday' and 'is_weekend'
publication_timing = ['weekday_is_tuesday',
                      'weekday_is_wednesday',
                      'weekday_is_thursday',
                      'weekday_is_friday',
                      'weekday_is_saturday',
                      'weekday_is_sunday']

# dropped 'data_channel_is_lifestyle'
content_topic_and_sentiment = ['data_channel_is_entertainment',
                               'data_channel_is_bus',
                               'data_channel_is_socmed',
                               'data_channel_is_tech',
                               'data_channel_is_world',
                               'LDA_00',
```

```
                               'LDA_01',
                               'LDA_02',
                               'LDA_03',
                               'LDA_04',
                               'rate_positive_words',
                               'rate_negative_words',
                               'avg_positive_polarity',
                               'min_positive_polarity',
                               'max_positive_polarity',
                               'avg_negative_polarity',
                               'min_negative_polarity',
                               'max_negative_polarity']
title_sentiment = ['title_subjectivity',
                   'title_sentiment_polarity',
                   'abs_title_subjectivity',
                   'abs_title_sentiment_polarity']
```

0.9 ### Feature Engineering Functions

Three functions were developed for the feature engineering process in order to loop all of the variables within a category through the loop. The first variable sqaures the features, while the second one cubes the features in the variable group. The third function interacts the features within the variable groups. Each of these functions create new variable groups for the feature engineered variables.

[10]:
```
def square_features(variables, df):
    sqaured_features = []
    for var in variables:
        feature_name = f'{var}_squared'
        df[feature_name] = df[var] ** 2
        sqaured_features.append(feature_name)
    return sqaured_features

def cube_features(variables, df):
    cubed_features = []
    for var in variables:
        feature_name = f'{var}_cubed'
        df[feature_name] = df[var] ** 3
        cubed_features.append(feature_name)
    return cubed_features

def interact_features(variables, df):
    interacted_features = []
    for (var1, var2) in combinations(variables, 2):
        feature_name = f'{var1}_{var2}_interaction'
        df[feature_name] = df[var1] * df[var2]
        interacted_features.append(feature_name)
```

```
    return interacted_features
```

[11]:
```
#################SQUARED TERMS##################
# square basic features
sqrd_basic_text_features = square_features(basic_text_features, train_data)
square_features(basic_text_features, test_data)

# square title sentiment features
sqrd_title_sentiment = square_features(title_sentiment, train_data)
square_features(title_sentiment, test_data)

# square content properties
sqrd_content_properties = square_features(content_properties, train_data)
square_features(content_properties, test_data)

# square keyword performance
sqrd_keyword_performance = square_features(keyword_performance, train_data)
square_features(keyword_performance, test_data)

# square self reference metrics
sqrd_self_reference_metrics = square_features(self_reference_metrics,␣
 ↪train_data)
square_features(self_reference_metrics, test_data)

#################CUBED TERMS##################
# CUBED basic features
cube_basic_text_features = cube_features(basic_text_features, train_data)
cube_features(basic_text_features, test_data)

# CUBED title sentiment features
cube_title_sentiment = cube_features(title_sentiment, train_data)
cube_features(title_sentiment, test_data)

# CUBED content properties
cube_content_properties = cube_features(content_properties, train_data)
cube_features(content_properties, test_data)

# CUBED keyword performance
cube_keyword_performance = cube_features(keyword_performance, train_data)
cube_features(keyword_performance, test_data)

# CUBED self reference metrics
cube_self_reference_metrics = cube_features(self_reference_metrics, train_data)
cube_features(self_reference_metrics, test_data)

###############INTERACTION TERMS#################
# Interacting the basic features
```

```
interaction_basic_text_features = interact_features(basic_text_features,␣
↪train_data)
interact_features(basic_text_features, test_data)

# Interacting the title sentiment features
interaction_title_sentiment = interact_features(title_sentiment, train_data)
interact_features(title_sentiment, test_data)

# Interacting content properties
interaction_content_properties = interact_features(content_properties,␣
↪train_data)
interact_features(content_properties, test_data)

# Interacting keyword performance
interaction_keyword_performance = interact_features(keyword_performance,␣
↪train_data)
interact_features(keyword_performance, test_data)

# Interacting self reference metrics
interaction_self_reference_metrics = interact_features(self_reference_metrics,␣
↪train_data)
interact_features(self_reference_metrics, test_data)
```

[11]:
```
['self_reference_min_shares_self_reference_max_shares_interaction',
 'self_reference_min_shares_self_reference_avg_sharess_interaction',
 'self_reference_max_shares_self_reference_avg_sharess_interaction']
```

0.10 ### Perm Importance Variables

After running the code previously, the best model was found to be M9 EBM. So, a permutation importance was ran on this model to determine the most important variables that did not score below 0 on the importance test. These variables are listed below and used in their own variable group and modeling group to see if this specific set of variables performs better. They did perform very well compared to the other groups.

[12]:
```
perm_importance_variables = ['n_tokens_title',
                             'n_tokens_content',
                             'n_unique_tokens',
                             'n_non_stop_words',
                             'n_non_stop_unique_tokens',
                             'average_token_length',
                             'num_keywords',
                             'num_hrefs',
                             'num_self_hrefs',
                             'num_imgs',
                             'num_videos',
                             'global_subjectivity',
```

```
                              'global_sentiment_polarity',
                              'kw_min_min',
                              'kw_max_min',
                              'kw_avg_min',
                              'kw_min_max',
                              'kw_max_max',
                              'kw_avg_max',
                              'kw_min_avg',
                              'kw_max_avg',
                              'kw_avg_avg',
                              'self_reference_min_shares',
                              'self_reference_max_shares',
                              'self_reference_avg_sharess',
                              'weekday_is_thursday',
                              'weekday_is_friday',
                              'weekday_is_sunday',
                              'data_channel_is_entertainment',
                              'data_channel_is_bus',
                              'data_channel_is_socmed',
                              'data_channel_is_tech',
                              'data_channel_is_world',
                              'LDA_00',
                              'LDA_01',
                              'LDA_02',
                              'LDA_03',
                              'LDA_04',
                              'rate_positive_words',
                              'avg_positive_polarity',
                              'min_positive_polarity',
                              'avg_negative_polarity',
                              'min_negative_polarity',
                              'max_negative_polarity',
                              'title_subjectivity',
                              'abs_title_subjectivity']
```

0.11 ### Defining Variable Models

Below, the variable groups are organized into groups for modeling. M1 is the baseline group with just the `basic_text_features`, and each model from there grows marginally with new groups to test out which groups perform the best. M31 has all of the groups including all of the feature engineered groups.

[14]:
```
# defining models
models = {
    'M1': basic_text_features,
    'M2': basic_text_features + content_properties,
    'M3': basic_text_features + content_properties + keyword_performance,
```

```
    'M4': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics,
    'M5': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing,
    'M6': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment,
    'M7': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment,
    'M8': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + sqrd_title_sentiment,
    'M9': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + sqrd_title_sentiment + sqrd_basic_text_features,
    'M10': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + sqrd_title_sentiment + sqrd_basic_text_features +␣
↪interaction_basic_text_features,
    'M11': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + sqrd_title_sentiment + sqrd_basic_text_features +␣
↪interaction_basic_text_features + interaction_title_sentiment,
    'M12': perm_importance_variables,
    'M13': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + sqrd_content_properties + sqrd_keyword_performance,
    'M14': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + sqrd_title_sentiment + sqrd_basic_text_features +␣
↪sqrd_content_properties,
    'M15': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + sqrd_title_sentiment + sqrd_basic_text_features +␣
↪sqrd_content_properties + sqrd_keyword_performance,
    'M16': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + sqrd_title_sentiment + sqrd_basic_text_features +␣
↪sqrd_content_properties + sqrd_keyword_performance +␣
↪sqrd_self_reference_metrics,
    'M17': basic_text_features + content_properties + keyword_performance +␣
↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
↪title_sentiment + interaction_content_properties,
```

```
    'M18': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + interaction_content_properties +
interaction_keyword_performance,
    'M19': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + interaction_content_properties +
interaction_keyword_performance + interaction_self_reference_metrics,
    'M20': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + interaction_content_properties +
interaction_keyword_performance + interaction_self_reference_metrics +
sqrd_title_sentiment + sqrd_basic_text_features + sqrd_content_properties +
sqrd_keyword_performance + sqrd_self_reference_metrics,
    'M21': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + interaction_content_properties +
interaction_basic_text_features + interaction_title_sentiment +
interaction_keyword_performance + interaction_self_reference_metrics +
sqrd_title_sentiment + sqrd_basic_text_features + sqrd_content_properties +
sqrd_keyword_performance + sqrd_self_reference_metrics,
    'M22': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + sqrd_title_sentiment + cube_title_sentiment,
    'M23': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + sqrd_title_sentiment + cube_title_sentiment +
sqrd_basic_text_features + cube_basic_text_features,
    'M24': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + sqrd_title_sentiment + cube_title_sentiment +
sqrd_basic_text_features + cube_basic_text_features +
sqrd_content_properties + cube_content_properties,
    'M25': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + sqrd_title_sentiment + cube_title_sentiment +
sqrd_basic_text_features + cube_basic_text_features +
sqrd_content_properties + cube_content_properties + sqrd_keyword_performance
+ cube_keyword_performance,
    'M26': basic_text_features + content_properties + keyword_performance +
self_reference_metrics + publication_timing + content_topic_and_sentiment +
title_sentiment + sqrd_title_sentiment + cube_title_sentiment +
sqrd_basic_text_features + cube_basic_text_features +
sqrd_content_properties + cube_content_properties + sqrd_keyword_performance
+ cube_keyword_performance + sqrd_self_reference_metrics +
cube_self_reference_metrics,
```

```
    'M27': basic_text_features + content_properties + keyword_performance +␣
 ↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
 ↪title_sentiment + sqrd_title_sentiment + cube_title_sentiment +␣
 ↪sqrd_basic_text_features + cube_basic_text_features +␣
 ↪sqrd_content_properties + cube_content_properties + sqrd_keyword_performance␣
 ↪+ cube_keyword_performance + sqrd_self_reference_metrics +␣
 ↪cube_self_reference_metrics + interaction_content_properties,
    'M28': basic_text_features + content_properties + keyword_performance +␣
 ↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
 ↪title_sentiment + sqrd_title_sentiment + cube_title_sentiment +␣
 ↪sqrd_basic_text_features + cube_basic_text_features +␣
 ↪sqrd_content_properties + cube_content_properties + sqrd_keyword_performance␣
 ↪+ cube_keyword_performance + sqrd_self_reference_metrics +␣
 ↪cube_self_reference_metrics + interaction_content_properties +␣
 ↪interaction_basic_text_features,
    'M29': basic_text_features + content_properties + keyword_performance +␣
 ↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
 ↪title_sentiment + sqrd_title_sentiment + cube_title_sentiment +␣
 ↪sqrd_basic_text_features + cube_basic_text_features +␣
 ↪sqrd_content_properties + cube_content_properties + sqrd_keyword_performance␣
 ↪+ cube_keyword_performance + sqrd_self_reference_metrics +␣
 ↪cube_self_reference_metrics + interaction_content_properties +␣
 ↪interaction_basic_text_features + interaction_title_sentiment,
    'M30': basic_text_features + content_properties + keyword_performance +␣
 ↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
 ↪title_sentiment + sqrd_title_sentiment + cube_title_sentiment +␣
 ↪sqrd_basic_text_features + cube_basic_text_features +␣
 ↪sqrd_content_properties + cube_content_properties + sqrd_keyword_performance␣
 ↪+ cube_keyword_performance + sqrd_self_reference_metrics +␣
 ↪cube_self_reference_metrics + interaction_content_properties +␣
 ↪interaction_basic_text_features + interaction_title_sentiment +␣
 ↪interaction_keyword_performance,
    'M31': basic_text_features + content_properties + keyword_performance +␣
 ↪self_reference_metrics + publication_timing + content_topic_and_sentiment +␣
 ↪title_sentiment + sqrd_title_sentiment + cube_title_sentiment +␣
 ↪sqrd_basic_text_features + cube_basic_text_features +␣
 ↪sqrd_content_properties + cube_content_properties + sqrd_keyword_performance␣
 ↪+ cube_keyword_performance + sqrd_self_reference_metrics +␣
 ↪cube_self_reference_metrics + interaction_content_properties +␣
 ↪interaction_basic_text_features + interaction_title_sentiment +␣
 ↪interaction_keyword_performance + interaction_self_reference_metrics


}
```

0.12 ### Splitting the data

The data was split at 20% into a training and validation set. The columns `is_popular`, `timedelta`, and `article_id` were dropped from the X dataset, while the y dataset is set to `is_popular`.

[15]:
```
# splitting 'train_data' into training and validation sets
X = train_data.drop(['is_popular', 'timedelta', 'article_id'], axis=1)
y = train_data['is_popular']
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=20240407)
```

0.13 # Models

0.14 ### RMSLE Function

The following RMSLE function was established for the purpose of additional evaluation between the models. Although, this may not be the most effective metric for certain models, such as neural networks, it will help with the interpretations of determining the best model incase the AUC scores are too similar. It will act as a tie-breaker.

[16]:
```
def calculateRMSLE(prediction, y_obs):
    return round(np.sqrt(
        np.mean(
            (
                np.log(np.where(prediction < 0, 0, prediction) + 1) -
                np.log(y_obs + 1)
            )**2
        )
    ), 4)
```

0.15 ### Results List Initilialization

The results list will contain all of the AUC and RMSLE scores

[17]:
```
# initilializing results list
results = []
```

0.16 ## Logistic Regression

0.17 ### Simple Logistic Regression

[18]:
```
for model_name, features in models.items():
    # appending "Logistic Regression" to the model name
    full_model_name = f"{model_name} Logistic Regression"

    # pipeline steps
    steps = [
        ("scale_features", ColumnTransformer([("scale", StandardScaler(),␣
↪features)], remainder='drop')),
```

```
        ("log_reg", LogisticRegression())
    ]

    # creating the pipeline
    pipeline = Pipeline(steps)

    # fitting the model on training data
    pipeline.fit(X_train[features], y_train)

    # predicting probabilities on the training and validation data
    train_prob = pipeline.predict_proba(X_train[features])[:, 1]
    val_prob = pipeline.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    results.append([full_model_name, train_auc, val_auc, train_rmsle,␣
 ↪val_rmsle])

# set the results_df and columns, this will be important for the rest of the␣
 ↪code
results_df = pd.DataFrame(results, columns=['Model', 'Training AUC',␣
 ↪'Validation AUC', 'Training RMSLE', 'Validation RMSLE'])

results_df.tail(31)
```

[18]:

	Model	Training AUC	Validation AUC	Training RMSLE	\
0	M1 Logistic Regression	0.548108	0.555135	0.2271	
1	M2 Logistic Regression	0.624687	0.627810	0.2253	
2	M3 Logistic Regression	0.682657	0.686424	0.2225	
3	M4 Logistic Regression	0.686342	0.688129	0.2224	
4	M5 Logistic Regression	0.687915	0.684988	0.2223	
5	M6 Logistic Regression	0.693311	0.694309	0.2220	
6	M7 Logistic Regression	0.694318	0.695099	0.2219	
7	M8 Logistic Regression	0.695176	0.696331	0.2219	
8	M9 Logistic Regression	0.695775	0.694353	0.2218	
9	M10 Logistic Regression	0.699305	0.693886	0.2216	
10	M11 Logistic Regression	0.699465	0.694220	0.2216	
11	M12 Logistic Regression	0.692398	0.697022	0.2220	
12	M13 Logistic Regression	0.701580	0.700482	0.2213	
13	M14 Logistic Regression	0.695972	0.694173	0.2218	

14	M15 Logistic Regression	0.702613	0.699578	0.2212
15	M16 Logistic Regression	0.709128	0.701346	0.2206
16	M17 Logistic Regression	0.697060	0.688289	0.2216
17	M18 Logistic Regression	0.703661	0.687535	0.2211
18	M19 Logistic Regression	0.710787	0.688838	0.2205
19	M20 Logistic Regression	0.716079	0.695409	0.2199
20	M21 Logistic Regression	0.718754	0.696427	0.2197
21	M22 Logistic Regression	0.695714	0.696995	0.2219
22	M23 Logistic Regression	0.697076	0.695949	0.2218
23	M24 Logistic Regression	0.698787	0.695604	0.2216
24	M25 Logistic Regression	0.708691	0.705051	0.2208
25	M26 Logistic Regression	0.715142	0.706303	0.2201
26	M27 Logistic Regression	0.716938	0.698923	0.2199
27	M28 Logistic Regression	0.719412	0.698379	0.2197
28	M29 Logistic Regression	0.719210	0.698091	0.2197
29	M30 Logistic Regression	0.721592	0.697905	0.2195
30	M31 Logistic Regression	0.722127	0.697809	0.2194

	Validation RMSLE
0	0.2314
1	0.2291
2	0.2259
3	0.2259
4	0.2260
5	0.2255
6	0.2253
7	0.2251
8	0.2252
9	0.2251
10	0.2251
11	0.2251
12	0.2245
13	0.2252
14	0.2244
15	0.2247
16	0.2261
17	0.2259
18	0.2262
19	0.2258
20	0.2255
21	0.2251
22	0.2251
23	0.2252
24	0.2240
25	0.2244
26	0.2253
27	0.2252

```
28              0.2253
29              0.2255
30              0.2254
```

0.18 #### Interpretation

This basic logistic regression model seems to be performing quite well. The scores could still be improved through other models, the training and validation sets are very closely related, suggesting that there is not an overfitting issue here. However, as we are not expecting this data to be able to fit linearlly, it might be best to try out some other models and see how they perform as compared to this baseline model.

0.19 ### Tuned Logistic Regression

[19]:
```
for model_name, features in models.items():
    # added a timer to visualize the progress of the model on each variable␣
 ↪groups
    start_time = time.time()
    # defining
    steps = [
        ("scale_features", ColumnTransformer([("scale", StandardScaler(),␣
 ↪features)], remainder='drop')),
        ("log_reg", LogisticRegression(solver='liblinear'))
    ]

    # creating the pipeline
    pipeline = Pipeline(steps)

    # defining a range of inverse regularization strength `C`
    param_grid = {
        'log_reg__C': [0.001, 0.01, 0.1, 1, 10, 100],
        'log_reg__penalty': ['l2']  # L2 regularization
    }

    # GridSearchCV
    grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='roc_auc')

    # fit the model
    grid_search.fit(X_train[features], y_train)

    # determine best estimators
    best_model = grid_search.best_estimator_

    # predict
    train_prob = best_model.predict_proba(X_train[features])[:, 1]
    val_prob = best_model.predict_proba(X_val[features])[:, 1]
```

```
    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    results.append([f"{model_name} Logistic Regression Tuned", train_auc,␣
 ↪val_auc, train_rmsle, val_rmsle])

    end_time = time.time()
    print(f"Completed {model_name} in {end_time - start_time:.2f} seconds")

results_df = pd.DataFrame(results, columns=['Model', 'Training AUC',␣
 ↪'Validation AUC', 'Training RMSLE', 'Validation RMSLE'])

results_df.tail(31)
```

```
Completed M1 in 0.98 seconds
Completed M2 in 1.91 seconds
Completed M3 in 4.39 seconds
Completed M4 in 4.96 seconds
Completed M5 in 6.05 seconds
Completed M6 in 10.84 seconds
Completed M7 in 12.25 seconds
Completed M8 in 17.74 seconds
Completed M9 in 25.21 seconds
Completed M10 in 111.26 seconds
Completed M11 in 91.14 seconds
Completed M12 in 8.28 seconds
Completed M13 in 31.09 seconds
Completed M14 in 29.58 seconds
Completed M15 in 45.38 seconds
Completed M16 in 44.61 seconds
Completed M17 in 28.90 seconds
Completed M18 in 73.55 seconds
Completed M19 in 80.24 seconds
Completed M20 in 187.04 seconds
Completed M21 in 341.66 seconds
Completed M22 in 22.43 seconds
Completed M23 in 37.47 seconds
Completed M24 in 60.85 seconds
Completed M25 in 88.07 seconds
Completed M26 in 106.10 seconds
Completed M27 in 141.23 seconds
```

```
Completed M28 in 263.02 seconds
Completed M29 in 358.95 seconds
Completed M30 in 408.96 seconds
Completed M31 in 436.17 seconds
```

[19]:

```
                            Model  Training AUC  Validation AUC  \
31   M1 Logistic Regression Tuned      0.578511        0.597577
32   M2 Logistic Regression Tuned      0.625409        0.628886
33   M3 Logistic Regression Tuned      0.683566        0.687520
34   M4 Logistic Regression Tuned      0.686853        0.688830
35   M5 Logistic Regression Tuned      0.688710        0.686084
36   M6 Logistic Regression Tuned      0.693308        0.694601
37   M7 Logistic Regression Tuned      0.694352        0.695420
38   M8 Logistic Regression Tuned      0.695495        0.696638
39   M9 Logistic Regression Tuned      0.695925        0.695024
40  M10 Logistic Regression Tuned      0.699440        0.694135
41  M11 Logistic Regression Tuned      0.698920        0.696317
42  M12 Logistic Regression Tuned      0.693144        0.698134
43  M13 Logistic Regression Tuned      0.701555        0.701145
44  M14 Logistic Regression Tuned      0.696147        0.694908
45  M15 Logistic Regression Tuned      0.702992        0.700911
46  M16 Logistic Regression Tuned      0.709479        0.702848
47  M17 Logistic Regression Tuned      0.695592        0.691340
48  M18 Logistic Regression Tuned      0.700774        0.694948
49  M19 Logistic Regression Tuned      0.707631        0.697339
50  M20 Logistic Regression Tuned      0.712031        0.700753
51  M21 Logistic Regression Tuned      0.713567        0.701898
52  M22 Logistic Regression Tuned      0.695370        0.696867
53  M23 Logistic Regression Tuned      0.696719        0.695677
54  M24 Logistic Regression Tuned      0.698321        0.695671
55  M25 Logistic Regression Tuned      0.709057        0.705078
56  M26 Logistic Regression Tuned      0.715581        0.706397
57  M27 Logistic Regression Tuned      0.717523        0.699163
58  M28 Logistic Regression Tuned      0.719902        0.699376
59  M29 Logistic Regression Tuned      0.719965        0.699616
60  M30 Logistic Regression Tuned      0.723291        0.699522
61  M31 Logistic Regression Tuned      0.723497        0.699059

    Training RMSLE  Validation RMSLE
31          0.2267            0.2305
32          0.2388            0.2415
33          0.2225            0.2259
34          0.2224            0.2259
35          0.2223            0.2260
36          0.2221            0.2255
37          0.2220            0.2253
38          0.2219            0.2251
```

39	0.2219	0.2253
40	0.2216	0.2251
41	0.2217	0.2250
42	0.2220	0.2251
43	0.2214	0.2246
44	0.2218	0.2252
45	0.2213	0.2245
46	0.2207	0.2247
47	0.2227	0.2266
48	0.2223	0.2263
49	0.2216	0.2264
50	0.2213	0.2260
51	0.2212	0.2257
52	0.2220	0.2251
53	0.2219	0.2252
54	0.2217	0.2252
55	0.2208	0.2240
56	0.2201	0.2243
57	0.2198	0.2253
58	0.2196	0.2251
59	0.2196	0.2251
60	0.2192	0.2253
61	0.2192	0.2254

0.20 #### Interpretation

After applying the GridSearchCV and several parameters, the model performances do not improve. This is likely due to the fact that the models were already performing relatively well without the tuning, and adding regularization did not help since this is typically used for reducing overfitting issues.

0.21 ## Lasso Model

[20]:
```
for group_name, features in models.items():
    start_time = time.time()  # Start timer

    # pipeline steps
    steps = [
        ("scale_features", ColumnTransformer([("scale_numeric_features",␣
↪MinMaxScaler(), features)], remainder='drop')),
        ("lasso", LassoCV())
    ]

    # create pipeline
    pipe_lasso = Pipeline(steps)

    # fit the model
```

```
    pipe_lasso.fit(X_train[features], y_train)

    # predict
    train_scores = pipe_lasso.predict(X_train[features])
    val_scores = pipe_lasso.predict(X_val[features])

    # convert scores to binary predictions based on the median threshold
    threshold = np.median(train_scores)
    train_pred = np.where(train_scores > threshold, 1, 0)
    val_pred = np.where(val_scores > threshold, 1, 0)

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_pred)
    val_auc = roc_auc_score(y_val, val_pred)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_pred, y_train)
    val_rmsle = calculateRMSLE(val_pred, y_val)

    # Append results
    new_row = pd.DataFrame([[f"{group_name} Lasso", train_auc, val_auc,␣
↪train_rmsle, val_rmsle]],
                            columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 0.29 seconds
Completed M2 in 0.29 seconds
Completed M3 in 0.35 seconds
Completed M4 in 0.40 seconds
Completed M5 in 0.41 seconds
Completed M6 in 1.13 seconds
Completed M7 in 1.11 seconds
Completed M8 in 1.14 seconds
Completed M9 in 1.28 seconds
Completed M10 in 1.97 seconds
Completed M11 in 2.08 seconds
Completed M12 in 0.87 seconds
Completed M13 in 1.56 seconds
Completed M14 in 1.33 seconds
Completed M15 in 1.60 seconds
Completed M16 in 1.27 seconds
```

```
Completed M17 in 1.46 seconds
Completed M18 in 2.36 seconds
Completed M19 in 2.14 seconds
Completed M20 in 3.06 seconds
Completed M21 in 3.99 seconds
Completed M22 in 1.40 seconds
Completed M23 in 1.46 seconds
Completed M24 in 1.82 seconds
Completed M25 in 2.70 seconds
Completed M26 in 2.95 seconds
Completed M27 in 3.77 seconds
Completed M28 in 4.38 seconds
Completed M29 in 4.61 seconds
Completed M30 in 5.34 seconds
Completed M31 in 5.89 seconds
```

[20]:

	Model	Training AUC	Validation AUC	Training RMSLE	Validation RMSLE
62	M1 Lasso	0.523024	0.531205	0.4853	0.4840
63	M2 Lasso	0.589911	0.592422	0.4711	0.4696
64	M3 Lasso	0.629210	0.625074	0.4625	0.4643
65	M4 Lasso	0.631790	0.629762	0.4620	0.4616
66	M5 Lasso	0.635561	0.631840	0.4611	0.4617
67	M6 Lasso	0.642905	0.646199	0.4595	0.4580
68	M7 Lasso	0.641516	0.654014	0.4598	0.4550
69	M8 Lasso	0.642905	0.649153	0.4595	0.4568
70	M9 Lasso	0.642508	0.644775	0.4596	0.4582
71	M10 Lasso	0.644294	0.643148	0.4592	0.4592
72	M11 Lasso	0.645088	0.645429	0.4590	0.4587
73	M12 Lasso	0.642707	0.652782	0.4595	0.4550
74	M13 Lasso	0.648661	0.646767	0.4582	0.4580
75	M14 Lasso	0.643699	0.647354	0.4593	0.4569
76	M15 Lasso	0.647669	0.642484	0.4584	0.4593
77	M16 Lasso	0.651241	0.650000	0.4576	0.4571
78	M17 Lasso	0.643302	0.649442	0.4594	0.4565
79	M18 Lasso	0.643501	0.651839	0.4594	0.4549
80	M19 Lasso	0.648264	0.653100	0.4583	0.4532
81	M20 Lasso	0.652035	0.644014	0.4575	0.4584
82	M21 Lasso	0.653424	0.646786	0.4572	0.4569
83	M22 Lasso	0.643302	0.647258	0.4594	0.4570
84	M23 Lasso	0.641913	0.645256	0.4597	0.4578
85	M24 Lasso	0.642508	0.647450	0.4596	0.4568
86	M25 Lasso	0.648463	0.645429	0.4583	0.4587
87	M26 Lasso	0.653821	0.648778	0.4571	0.4566
88	M27 Lasso	0.652234	0.645342	0.4574	0.4582
89	M28 Lasso	0.652829	0.641936	0.4573	0.4582
90	M29 Lasso	0.652432	0.642504	0.4574	0.4582
91	M30 Lasso	0.653028	0.646401	0.4573	0.4573

```
92  M31 Lasso      0.653226        0.646680          0.4572            0.4575
```

0.22 #### Interpretation

The Lasso model is performing significantly worse than the logistic regression. This is reflected in both the RMSLE and AUC scores; however, the training and validation scores are very similar to each other, suggesting that there is not overfitting here. The Lasso model is likely not a good canidate for the best predictive model.

0.23 ## Stacking Model

The stacking model is an ensemble method that stacks other types of models to see if combining models will create a better model overall. This is a model that I have not personally used before, so it is for experimentation purposes.

[21]:
```
# defining the base models
base_models = [
    ('dt', DecisionTreeClassifier(max_depth=5, random_state=20240407)),
    ('rf', RandomForestClassifier(max_depth=5, n_estimators=100,␣
 ↪random_state=20240407)),
    ('xgb', xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss',␣
 ↪max_depth=3, n_estimators=100, random_state=20240407))
]

# Meta-model
meta_model = LogisticRegression()

# Stacking classifier
stacking_model = StackingClassifier(estimators=base_models,␣
 ↪final_estimator=meta_model, cv=5)

for model_name, features in models.items():
    start_time = time.time()  # Start timer

    # pipeline
    pipeline = Pipeline([
        ("scale_features", ColumnTransformer([("scale", StandardScaler(),␣
 ↪features)], remainder='drop')),
        ("stacking", stacking_model)
    ])

    # Fit model
    pipeline.fit(X_train[features], y_train)

    # Predict probabilities
    train_prob = pipeline.predict_proba(X_train[features])[:, 1]
    val_prob = pipeline.predict_proba(X_val[features])[:, 1]
```

```python
    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    new_row = pd.DataFrame([[f"{model_name} STACKED", train_auc, val_auc,␣
↪train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {model_name} in {end_time - start_time:.2f} seconds")


results_df.tail(31)
```

```
Completed M1 in 27.42 seconds
Completed M2 in 37.75 seconds
Completed M3 in 47.45 seconds
Completed M4 in 56.97 seconds
Completed M5 in 49.07 seconds
Completed M6 in 68.92 seconds
Completed M7 in 67.88 seconds
Completed M8 in 66.06 seconds
Completed M9 in 76.81 seconds
Completed M10 in 104.90 seconds
Completed M11 in 104.74 seconds
Completed M12 in 59.84 seconds
Completed M13 in 82.94 seconds
Completed M14 in 80.55 seconds
Completed M15 in 92.28 seconds
Completed M16 in 92.47 seconds
Completed M17 in 103.85 seconds
Completed M18 in 149.08 seconds
Completed M19 in 158.47 seconds
Completed M20 in 174.34 seconds
Completed M21 in 197.28 seconds
Completed M22 in 65.26 seconds
Completed M23 in 76.45 seconds
Completed M24 in 91.53 seconds
Completed M25 in 106.27 seconds
```

```
Completed M26 in 108.24 seconds
Completed M27 in 143.81 seconds
Completed M28 in 438.40 seconds
Completed M29 in 171.27 seconds
Completed M30 in 205.02 seconds
Completed M31 in 206.20 seconds
```

[21]:

```
           Model  Training AUC  Validation AUC  Training RMSLE  \
93    M1 STACKED      0.999279        0.582803          0.1769
94    M2 STACKED      0.999535        0.647539          0.1329
95    M3 STACKED      1.000000        0.694660          0.0860
96    M4 STACKED      1.000000        0.701611          0.0851
97    M5 STACKED      1.000000        0.704613          0.0814
98    M6 STACKED      1.000000        0.714806          0.0774
99    M7 STACKED      1.000000        0.715391          0.0787
100   M8 STACKED      1.000000        0.716757          0.0797
101   M9 STACKED      1.000000        0.712760          0.0766
102  M10 STACKED      1.000000        0.710793          0.0816
103  M11 STACKED      1.000000        0.708077          0.0782
104  M12 STACKED      1.000000        0.714303          0.0787
105  M13 STACKED      1.000000        0.712307          0.0804
106  M14 STACKED      1.000000        0.708617          0.0783
107  M15 STACKED      1.000000        0.715043          0.0780
108  M16 STACKED      1.000000        0.714724          0.0809
109  M17 STACKED      1.000000        0.707926          0.0800
110  M18 STACKED      1.000000        0.709377          0.0832
111  M19 STACKED      1.000000        0.708620          0.0855
112  M20 STACKED      1.000000        0.717524          0.0840
113  M21 STACKED      1.000000        0.714015          0.0824
114  M22 STACKED      1.000000        0.704769          0.0785
115  M23 STACKED      1.000000        0.714010          0.0798
116  M24 STACKED      1.000000        0.710349          0.0793
117  M25 STACKED      1.000000        0.718301          0.0807
118  M26 STACKED      1.000000        0.704987          0.0812
119  M27 STACKED      1.000000        0.708281          0.0844
120  M28 STACKED      1.000000        0.711128          0.0818
121  M29 STACKED      1.000000        0.711505          0.0822
122  M30 STACKED      1.000000        0.711104          0.0806
123  M31 STACKED      1.000000        0.713077          0.0811

     Validation RMSLE
93             0.2308
94             0.2291
95             0.2254
96             0.2257
97             0.2256
98             0.2253
```

```
99                0.2244
100               0.2244
101               0.2252
102               0.2249
103               0.2251
104               0.2244
105               0.2252
106               0.2251
107               0.2245
108               0.2251
109               0.2258
110               0.2256
111               0.2257
112               0.2251
113               0.2248
114               0.2250
115               0.2247
116               0.2250
117               0.2242
118               0.2254
119               0.2254
120               0.2245
121               0.2252
122               0.2248
123               0.2250
```

0.24 #### Interpretation

There seems to be some improvment in the validation AUC scores by a small margin; however, the training set cannot be overlooked with a clear overfitting issue. More measures may need to be taken into considation in the future to reduce this overfitting, but it does not seem like this model will work out.

0.25 ## Decision Tree Classifier

Multiple depths were tested for the decision tree classifier, ending with a GridSearchCV to consider an array of max depths and determine the most appropriate max depth per variable group.

0.25.1 Decision Tree Classifer Max Depth 5

[22]:
```python
for group_name, features in models.items():
    start_time = time.time()  # Start timer

    steps = [
        ("scale_features", ColumnTransformer([("scale_numeric_features",␣
↪MinMaxScaler(), features)], remainder='drop')),
        ("tree", DecisionTreeClassifier(max_depth=5, random_state=20240407))
```

```
    ]
    pipe_tree = Pipeline(steps)

    # Fit the model
    pipe_tree.fit(X_train[features], y_train)

    # Predict probabilities
    train_prob = pipe_tree.predict_proba(X_train[features])[:, 1]
    val_prob = pipe_tree.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    new_row = pd.DataFrame([[f"{group_name} Decision Tree MD5", train_auc,␣
 ↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
 ↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 0.10 seconds
Completed M2 in 0.18 seconds
Completed M3 in 0.27 seconds
Completed M4 in 0.27 seconds
Completed M5 in 0.29 seconds
Completed M6 in 0.42 seconds
Completed M7 in 0.45 seconds
Completed M8 in 0.46 seconds
Completed M9 in 0.53 seconds
Completed M10 in 0.78 seconds
Completed M11 in 0.81 seconds
Completed M12 in 0.40 seconds
Completed M13 in 0.63 seconds
Completed M14 in 0.61 seconds
Completed M15 in 0.70 seconds
Completed M16 in 0.73 seconds
Completed M17 in 0.79 seconds
```

```
Completed M18 in 1.35 seconds
Completed M19 in 1.36 seconds
Completed M20 in 1.65 seconds
Completed M21 in 1.86 seconds
Completed M22 in 0.49 seconds
Completed M23 in 0.60 seconds
Completed M24 in 0.75 seconds
Completed M25 in 0.94 seconds
Completed M26 in 0.97 seconds
Completed M27 in 1.32 seconds
Completed M28 in 1.58 seconds
Completed M29 in 1.61 seconds
Completed M30 in 2.09 seconds
Completed M31 in 2.20 seconds
```

[22]:

	Model	Training AUC	Validation AUC	Training RMSLE \
124	M1 Decision Tree MD5	0.593985	0.579518	0.2254
125	M2 Decision Tree MD5	0.645188	0.613544	0.2229
126	M3 Decision Tree MD5	0.689169	0.681748	0.2207
127	M4 Decision Tree MD5	0.702222	0.677891	0.2197
128	M5 Decision Tree MD5	0.702322	0.675899	0.2197
129	M6 Decision Tree MD5	0.701975	0.673412	0.2197
130	M7 Decision Tree MD5	0.702134	0.669154	0.2197
131	M8 Decision Tree MD5	0.702134	0.668961	0.2197
132	M9 Decision Tree MD5	0.702134	0.669002	0.2197
133	M10 Decision Tree MD5	0.699318	0.672435	0.2196
134	M11 Decision Tree MD5	0.699161	0.671234	0.2196
135	M12 Decision Tree MD5	0.701146	0.674605	0.2197
136	M13 Decision Tree MD5	0.702134	0.669234	0.2197
137	M14 Decision Tree MD5	0.702134	0.669234	0.2197
138	M15 Decision Tree MD5	0.702134	0.668961	0.2197
139	M16 Decision Tree MD5	0.702134	0.668961	0.2197
140	M17 Decision Tree MD5	0.700275	0.674260	0.2193
141	M18 Decision Tree MD5	0.700671	0.679698	0.2191
142	M19 Decision Tree MD5	0.700671	0.679698	0.2191
143	M20 Decision Tree MD5	0.700671	0.679698	0.2191
144	M21 Decision Tree MD5	0.700574	0.680021	0.2191
145	M22 Decision Tree MD5	0.702134	0.669002	0.2197
146	M23 Decision Tree MD5	0.702134	0.669002	0.2197
147	M24 Decision Tree MD5	0.702134	0.669002	0.2197
148	M25 Decision Tree MD5	0.702134	0.669042	0.2197
149	M26 Decision Tree MD5	0.702134	0.668961	0.2197
150	M27 Decision Tree MD5	0.700275	0.674260	0.2193
151	M28 Decision Tree MD5	0.699245	0.676840	0.2193
152	M29 Decision Tree MD5	0.698546	0.681102	0.2193
153	M30 Decision Tree MD5	0.700574	0.680021	0.2191
154	M31 Decision Tree MD5	0.700574	0.680021	0.2191

```
     Validation RMSLE
124            0.2309
125            0.2310
126            0.2274
127            0.2263
128            0.2265
129            0.2268
130            0.2275
131            0.2277
132            0.2275
133            0.2274
134            0.2279
135            0.2270
136            0.2272
137            0.2272
138            0.2277
139            0.2277
140            0.2286
141            0.2281
142            0.2281
143            0.2281
144            0.2283
145            0.2275
146            0.2275
147            0.2275
148            0.2274
149            0.2277
150            0.2286
151            0.2283
152            0.2280
153            0.2283
154            0.2283
```

0.26 #### Interpretation

The Max Depth 5 Decision Tree Classifier model performs decently, but still not consistently better than the initial logistic regression. Both the training and validation sets seem to be performing similarly here, which suggests that overfitting may not be an issue at max depth 5. We will now test other max depths to see how they perform.

0.27 ### Decision Tree Classifer Max Depth 6

[23]:
```
for group_name, features in models.items():
    start_time = time.time()  # Start timer

    steps = [
```

```
        ("scale_features", ColumnTransformer([("scale_numeric_features",␣
↪MinMaxScaler(), features)], remainder='drop')),
        ("tree", DecisionTreeClassifier(max_depth=6, random_state=20240407))
    ]
    pipe_tree = Pipeline(steps)

    # Fit the model
    pipe_tree.fit(X_train[features], y_train)

    # Predict probabilities
    train_prob = pipe_tree.predict_proba(X_train[features])[:, 1]
    val_prob = pipe_tree.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    new_row = pd.DataFrame([[f"{group_name} Decision Tree MD6", train_auc,␣
↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 0.10 seconds
Completed M2 in 0.20 seconds
Completed M3 in 0.32 seconds
Completed M4 in 0.34 seconds
Completed M5 in 0.34 seconds
Completed M6 in 0.52 seconds
Completed M7 in 0.53 seconds
Completed M8 in 0.55 seconds
Completed M9 in 0.61 seconds
Completed M10 in 0.92 seconds
Completed M11 in 0.97 seconds
Completed M12 in 0.46 seconds
Completed M13 in 0.72 seconds
Completed M14 in 0.70 seconds
```

```
Completed M15 in 0.82 seconds
Completed M16 in 0.85 seconds
Completed M17 in 0.92 seconds
Completed M18 in 1.51 seconds
Completed M19 in 1.50 seconds
Completed M20 in 1.92 seconds
Completed M21 in 2.26 seconds
Completed M22 in 0.62 seconds
Completed M23 in 0.70 seconds
Completed M24 in 0.88 seconds
Completed M25 in 1.09 seconds
Completed M26 in 1.15 seconds
Completed M27 in 1.54 seconds
Completed M28 in 1.85 seconds
Completed M29 in 1.88 seconds
Completed M30 in 2.42 seconds
Completed M31 in 2.58 seconds
```

[23]:

	Model	Training AUC	Validation AUC	Training RMSLE \
155	M1 Decision Tree MD6	0.606142	0.563066	0.2242
156	M2 Decision Tree MD6	0.659318	0.615339	0.2211
157	M3 Decision Tree MD6	0.702358	0.683364	0.2188
158	M4 Decision Tree MD6	0.715865	0.669937	0.2173
159	M5 Decision Tree MD6	0.716589	0.672902	0.2173
160	M6 Decision Tree MD6	0.717090	0.670325	0.2172
161	M7 Decision Tree MD6	0.717947	0.666980	0.2174
162	M8 Decision Tree MD6	0.717947	0.666980	0.2174
163	M9 Decision Tree MD6	0.717947	0.666983	0.2174
164	M10 Decision Tree MD6	0.711630	0.673889	0.2175
165	M11 Decision Tree MD6	0.711634	0.675729	0.2174
166	M12 Decision Tree MD6	0.715329	0.672186	0.2172
167	M13 Decision Tree MD6	0.718217	0.665591	0.2173
168	M14 Decision Tree MD6	0.718217	0.666726	0.2173
169	M15 Decision Tree MD6	0.718217	0.666726	0.2173
170	M16 Decision Tree MD6	0.718217	0.667106	0.2173
171	M17 Decision Tree MD6	0.714122	0.668302	0.2168
172	M18 Decision Tree MD6	0.716974	0.673574	0.2164
173	M19 Decision Tree MD6	0.716974	0.672723	0.2164
174	M20 Decision Tree MD6	0.716974	0.673623	0.2164
175	M21 Decision Tree MD6	0.716640	0.672329	0.2164
176	M22 Decision Tree MD6	0.717947	0.667172	0.2174
177	M23 Decision Tree MD6	0.717947	0.666791	0.2174
178	M24 Decision Tree MD6	0.718217	0.665591	0.2173
179	M25 Decision Tree MD6	0.718217	0.665399	0.2173
180	M26 Decision Tree MD6	0.718217	0.665591	0.2173
181	M27 Decision Tree MD6	0.714122	0.666611	0.2168
182	M28 Decision Tree MD6	0.713867	0.668258	0.2168

183	M29 Decision Tree MD6	0.712036	0.679049	0.2168
184	M30 Decision Tree MD6	0.716640	0.672216	0.2164
185	M31 Decision Tree MD6	0.716640	0.672379	0.2164

	Validation RMSLE
155	0.2322
156	0.2326
157	0.2288
158	0.2285
159	0.2283
160	0.2282
161	0.2286
162	0.2286
163	0.2286
164	0.2285
165	0.2285
166	0.2281
167	0.2287
168	0.2287
169	0.2287
170	0.2283
171	0.2299
172	0.2297
173	0.2298
174	0.2296
175	0.2302
176	0.2284
177	0.2287
178	0.2287
179	0.2288
180	0.2287
181	0.2303
182	0.2299
183	0.2289
184	0.2303
185	0.2301

0.28 #### Interpretation

There seems to be a slight increase in the training AUC and a slight decrease in the validation AUC, suggesting that the increase in max depth may be triggering overfitting. Thus, this model performs worse than the Max Depth 5.

0.29 ### Decision Tree Classifer Max Depth 7

[24]:
```
for group_name, features in models.items():
    start_time = time.time()  # Start timer

    steps = [
        ("scale_features", ColumnTransformer([("scale_numeric_features",␣
↪MinMaxScaler(), features)], remainder='drop')),
        ("tree", DecisionTreeClassifier(max_depth=7, random_state=20240407))
    ]
    pipe_tree = Pipeline(steps)

    # Fit the model
    pipe_tree.fit(X_train[features], y_train)

    # Predict probabilities
    train_prob = pipe_tree.predict_proba(X_train[features])[:, 1]
    val_prob = pipe_tree.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    new_row = pd.DataFrame([[f"{group_name} Decision Tree MD7", train_auc,␣
↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 0.12 seconds
Completed M2 in 0.21 seconds
Completed M3 in 0.36 seconds
Completed M4 in 0.39 seconds
Completed M5 in 0.39 seconds
Completed M6 in 0.59 seconds
Completed M7 in 0.59 seconds
Completed M8 in 0.64 seconds
Completed M9 in 0.70 seconds
```

```
Completed M10 in 1.05 seconds
Completed M11 in 1.10 seconds
Completed M12 in 0.53 seconds
Completed M13 in 0.83 seconds
Completed M14 in 0.81 seconds
Completed M15 in 0.93 seconds
Completed M16 in 0.97 seconds
Completed M17 in 1.05 seconds
Completed M18 in 1.67 seconds
Completed M19 in 1.76 seconds
Completed M20 in 2.17 seconds
Completed M21 in 2.49 seconds
Completed M22 in 0.65 seconds
Completed M23 in 0.79 seconds
Completed M24 in 1.01 seconds
Completed M25 in 1.26 seconds
Completed M26 in 1.33 seconds
Completed M27 in 1.76 seconds
Completed M28 in 2.10 seconds
Completed M29 in 2.25 seconds
Completed M30 in 2.96 seconds
Completed M31 in 2.91 seconds
```

[24]:

```
                    Model  Training AUC  Validation AUC  Training RMSLE  \
186   M1 Decision Tree MD7      0.618633        0.574762          0.2229
187   M2 Decision Tree MD7      0.676916        0.621189          0.2183
188   M3 Decision Tree MD7      0.716705        0.679916          0.2162
189   M4 Decision Tree MD7      0.731960        0.658502          0.2138
190   M5 Decision Tree MD7      0.732297        0.660033          0.2137
191   M6 Decision Tree MD7      0.732842        0.668368          0.2138
192   M7 Decision Tree MD7      0.734117        0.666903          0.2140
193   M8 Decision Tree MD7      0.734678        0.667129          0.2139
194   M9 Decision Tree MD7      0.734117        0.665766          0.2140
195  M10 Decision Tree MD7      0.726670        0.676924          0.2143
196  M11 Decision Tree MD7      0.725687        0.675754          0.2141
197  M12 Decision Tree MD7      0.730659        0.673338          0.2139
198  M13 Decision Tree MD7      0.735542        0.660959          0.2139
199  M14 Decision Tree MD7      0.734974        0.660638          0.2139
200  M15 Decision Tree MD7      0.734974        0.662002          0.2139
201  M16 Decision Tree MD7      0.734974        0.662615          0.2139
202  M17 Decision Tree MD7      0.728223        0.665090          0.2133
203  M18 Decision Tree MD7      0.731147        0.669459          0.2124
204  M19 Decision Tree MD7      0.729218        0.671922          0.2124
205  M20 Decision Tree MD7      0.729301        0.670441          0.2124
206  M21 Decision Tree MD7      0.729804        0.666320          0.2122
207  M22 Decision Tree MD7      0.734117        0.667091          0.2140
208  M23 Decision Tree MD7      0.734117        0.666697          0.2140
```

209	M24 Decision Tree MD7	0.734974	0.660912	0.2139
210	M25 Decision Tree MD7	0.735542	0.660876	0.2139
211	M26 Decision Tree MD7	0.735542	0.660919	0.2139
212	M27 Decision Tree MD7	0.728339	0.667062	0.2133
213	M28 Decision Tree MD7	0.726385	0.669269	0.2128
214	M29 Decision Tree MD7	0.725104	0.680026	0.2127
215	M30 Decision Tree MD7	0.730528	0.667052	0.2123
216	M31 Decision Tree MD7	0.729804	0.664633	0.2122

	Validation RMSLE
186	0.2334
187	0.2361
188	0.2296
189	0.2317
190	0.2325
191	0.2314
192	0.2311
193	0.2310
194	0.2312
195	0.2302
196	0.2302
197	0.2309
198	0.2317
199	0.2320
200	0.2317
201	0.2310
202	0.2326
203	0.2320
204	0.2319
205	0.2319
206	0.2334
207	0.2310
208	0.2314
209	0.2316
210	0.2319
211	0.2317
212	0.2319
213	0.2332
214	0.2315
215	0.2333
216	0.2340

0.30 #### Interpretation

The results here further validate the interpretation from the max depth 6 model. This increase in max depth is causing overfitting in the models. The Grid Search will allow us to see which max depths are most appropriate for these models to avoid overfitting.

0.31 ### Decision Tree Classifer Grid Search

[25]:
```
for group_name, features in models.items():
    start_time = time.time()  # Start timer

    # pipeline steps
    steps = [
        ("scale_features", ColumnTransformer([("scale_numeric_features",␣
↪MinMaxScaler(), features)], remainder='drop')),
        ("tree", DecisionTreeClassifier(random_state=20240407))
    ]
    pipe_tree = Pipeline(steps)

    # the parameter grid to search over
    param_grid = {
        "tree__max_depth": range(3, 9)
    }

    # initialize GridSearchCV
    grid_search = GridSearchCV(pipe_tree, param_grid, cv=5, scoring='roc_auc',␣
↪n_jobs=-1)

    # Fit the model
    grid_search.fit(X_train[features], y_train)

    # Best model after grid search
    best_model = grid_search.best_estimator_

    # Predict probabilities
    train_prob = best_model.predict_proba(X_train[features])[:, 1]
    val_prob = best_model.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    best_depth = best_model.named_steps['tree'].max_depth
    new_row = pd.DataFrame([[f"{group_name} Decision Tree Grid Search",␣
↪train_auc, val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)
```

```
    end_time = time.time()  # End timer
    print(f"Completed {group_name} with best max_depth={best_depth} in␣
↪{end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 with best max_depth=7 in 5.48 seconds
Completed M2 with best max_depth=5 in 1.02 seconds
Completed M3 with best max_depth=4 in 1.30 seconds
Completed M4 with best max_depth=5 in 1.51 seconds
Completed M5 with best max_depth=5 in 1.57 seconds
Completed M6 with best max_depth=4 in 2.19 seconds
Completed M7 with best max_depth=4 in 2.31 seconds
Completed M8 with best max_depth=4 in 2.44 seconds
Completed M9 with best max_depth=4 in 2.69 seconds
Completed M10 with best max_depth=5 in 4.27 seconds
Completed M11 with best max_depth=5 in 4.52 seconds
Completed M12 with best max_depth=4 in 2.01 seconds
Completed M13 with best max_depth=4 in 3.25 seconds
Completed M14 with best max_depth=4 in 3.17 seconds
Completed M15 with best max_depth=4 in 3.55 seconds
Completed M16 with best max_depth=4 in 3.77 seconds
Completed M17 with best max_depth=3 in 3.89 seconds
Completed M18 with best max_depth=5 in 6.64 seconds
Completed M19 with best max_depth=3 in 6.39 seconds
Completed M20 with best max_depth=4 in 8.51 seconds
Completed M21 with best max_depth=3 in 9.25 seconds
Completed M22 with best max_depth=4 in 2.67 seconds
Completed M23 with best max_depth=4 in 3.21 seconds
Completed M24 with best max_depth=4 in 4.23 seconds
Completed M25 with best max_depth=4 in 4.97 seconds
Completed M26 with best max_depth=4 in 5.58 seconds
Completed M27 with best max_depth=3 in 6.80 seconds
Completed M28 with best max_depth=5 in 8.84 seconds
Completed M29 with best max_depth=5 in 8.87 seconds
Completed M30 with best max_depth=3 in 10.53 seconds
Completed M31 with best max_depth=3 in 10.65 seconds
```

[25]:

	Model	Training AUC	Validation AUC \
217	M1 Decision Tree Grid Search	0.618633	0.574762
218	M2 Decision Tree Grid Search	0.645188	0.613544
219	M3 Decision Tree Grid Search	0.677494	0.677595
220	M4 Decision Tree Grid Search	0.702222	0.677891
221	M5 Decision Tree Grid Search	0.702322	0.675899
222	M6 Decision Tree Grid Search	0.688481	0.671874
223	M7 Decision Tree Grid Search	0.688161	0.671863

224	M8 Decision Tree Grid Search	0.688161	0.671863
225	M9 Decision Tree Grid Search	0.688161	0.671863
226	M10 Decision Tree Grid Search	0.699318	0.672435
227	M11 Decision Tree Grid Search	0.699161	0.671234
228	M12 Decision Tree Grid Search	0.688481	0.671874
229	M13 Decision Tree Grid Search	0.688161	0.671863
230	M14 Decision Tree Grid Search	0.688161	0.671863
231	M15 Decision Tree Grid Search	0.688161	0.671863
232	M16 Decision Tree Grid Search	0.688161	0.671863
233	M17 Decision Tree Grid Search	0.678634	0.673502
234	M18 Decision Tree Grid Search	0.700671	0.679698
235	M19 Decision Tree Grid Search	0.678560	0.673000
236	M20 Decision Tree Grid Search	0.690074	0.683425
237	M21 Decision Tree Grid Search	0.678560	0.673000
238	M22 Decision Tree Grid Search	0.688161	0.671863
239	M23 Decision Tree Grid Search	0.688161	0.671863
240	M24 Decision Tree Grid Search	0.688161	0.671863
241	M25 Decision Tree Grid Search	0.688161	0.671863
242	M26 Decision Tree Grid Search	0.688161	0.671863
243	M27 Decision Tree Grid Search	0.678634	0.673502
244	M28 Decision Tree Grid Search	0.699245	0.676840
245	M29 Decision Tree Grid Search	0.698546	0.681102
246	M30 Decision Tree Grid Search	0.678560	0.673000
247	M31 Decision Tree Grid Search	0.678560	0.673000

	Training RMSLE	Validation RMSLE
217	0.2229	0.2334
218	0.2229	0.2310
219	0.2219	0.2268
220	0.2197	0.2263
221	0.2197	0.2265
222	0.2212	0.2271
223	0.2212	0.2272
224	0.2212	0.2272
225	0.2212	0.2272
226	0.2196	0.2274
227	0.2196	0.2279
228	0.2212	0.2271
229	0.2212	0.2272
230	0.2212	0.2272
231	0.2212	0.2272
232	0.2212	0.2272
233	0.2219	0.2271
234	0.2191	0.2281
235	0.2219	0.2270
236	0.2209	0.2271
237	0.2219	0.2270

238	0.2212	0.2272
239	0.2212	0.2272
240	0.2212	0.2272
241	0.2212	0.2272
242	0.2212	0.2272
243	0.2219	0.2271
244	0.2193	0.2283
245	0.2193	0.2280
246	0.2219	0.2270
247	0.2219	0.2270

0.32 #### Interpretation

The Grid Search found that the models performed best with lower max depth values. Max Depth 3, 4, and 5 were the most frequented, out of a range from 3 to 9. The AUC scores suggest that the models are performing better than when set to a specific max depth, and the overfitting issue is resolved; however, these models are not performing better than the logistic regression model from earlier, and thus, are not ideal to be selected as the best predictive models.

0.33 ## Random Forest

[26]:
```
for group_name, features in models.items():
    start_time = time.time()  # Start timer

    steps = [
        ("scale_features", ColumnTransformer([("scale_numeric_features",␣
↪MinMaxScaler(), features)], remainder='drop')),
        ("random_forest", RandomForestClassifier(random_state=20240407))
    ]
    pipe_rf = Pipeline(steps)

    # the parameter grid to search over
    param_grid = {
        "random_forest__max_depth": [None, 3, 5, 7],
        "random_forest__n_estimators": [10, 50, 100],
        "random_forest__min_samples_split": [2, 4]
    }

    # Initialize GridSearchCV
    grid_search = GridSearchCV(pipe_rf, param_grid, cv=5, scoring='roc_auc',␣
↪n_jobs=-1)

    # Fit the model
    grid_search.fit(X_train[features], y_train)

    # Best model after grid search
    best_model = grid_search.best_estimator_
```

```
    # Predict probabilities
    train_prob = best_model.predict_proba(X_train[features])[:, 1]
    val_prob = best_model.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    best_params = grid_search.best_params_
    new_row = pd.DataFrame([[f"{group_name} Random Forest", train_auc, val_auc,␣
↪train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {group_name} with best parameters {best_params} in␣
↪{end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
17.49 seconds
Completed M2 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
23.53 seconds
Completed M3 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
31.09 seconds
Completed M4 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
36.96 seconds
Completed M5 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
32.57 seconds
Completed M6 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
43.04 seconds
Completed M7 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
```

```
42.59 seconds
Completed M8 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
41.54 seconds
Completed M9 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
47.13 seconds
Completed M10 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
60.64 seconds
Completed M11 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
61.48 seconds
Completed M12 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
40.40 seconds
Completed M13 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
50.61 seconds
Completed M14 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
49.37 seconds
Completed M15 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
56.47 seconds
Completed M16 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
55.60 seconds
Completed M17 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
63.90 seconds
Completed M18 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
79.08 seconds
Completed M19 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
85.83 seconds
Completed M20 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
92.56 seconds
Completed M21 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
103.58 seconds
Completed M22 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
40.73 seconds
Completed M23 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
```

```
47.14 seconds
Completed M24 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
56.03 seconds
Completed M25 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
63.60 seconds
Completed M26 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
63.69 seconds
Completed M27 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
82.93 seconds
Completed M28 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
88.33 seconds
Completed M29 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 4, 'random_forest__n_estimators': 100} in
94.43 seconds
Completed M30 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
109.03 seconds
Completed M31 with best parameters {'random_forest__max_depth': 7,
'random_forest__min_samples_split': 2, 'random_forest__n_estimators': 100} in
109.70 seconds
```

[26]:

	Model	Training AUC	Validation AUC	Training RMSLE	\
248	M1 Random Forest	0.692130	0.603921	0.2215	
249	M2 Random Forest	0.733817	0.656886	0.2172	
250	M3 Random Forest	0.774158	0.708519	0.2132	
251	M4 Random Forest	0.782287	0.711795	0.2109	
252	M5 Random Forest	0.781630	0.712681	0.2114	
253	M6 Random Forest	0.792354	0.714636	0.2109	
254	M7 Random Forest	0.792197	0.715297	0.2110	
255	M8 Random Forest	0.791399	0.713278	0.2111	
256	M9 Random Forest	0.795269	0.714616	0.2106	
257	M10 Random Forest	0.795790	0.715590	0.2104	
258	M11 Random Forest	0.793728	0.717255	0.2105	
259	M12 Random Forest	0.794477	0.714816	0.2108	
260	M13 Random Forest	0.792061	0.713364	0.2105	
261	M14 Random Forest	0.794005	0.715745	0.2107	
262	M15 Random Forest	0.792506	0.716487	0.2105	
263	M16 Random Forest	0.792441	0.713936	0.2100	
264	M17 Random Forest	0.793682	0.713212	0.2101	
265	M18 Random Forest	0.793187	0.714930	0.2094	
266	M19 Random Forest	0.792326	0.715411	0.2090	
267	M20 Random Forest	0.791841	0.714108	0.2089	

268	M21 Random Forest	0.793415	0.717114	0.2087
269	M22 Random Forest	0.794812	0.714967	0.2110
270	M23 Random Forest	0.795248	0.712613	0.2112
271	M24 Random Forest	0.796912	0.710943	0.2103
272	M25 Random Forest	0.796458	0.715752	0.2102
273	M26 Random Forest	0.788110	0.713519	0.2102
274	M27 Random Forest	0.790270	0.714281	0.2095
275	M28 Random Forest	0.791918	0.711802	0.2093
276	M29 Random Forest	0.793466	0.713869	0.2093
277	M30 Random Forest	0.794712	0.713761	0.2087
278	M31 Random Forest	0.793070	0.713840	0.2086

	Validation RMSLE
248	0.2301
249	0.2279
250	0.2241
251	0.2241
252	0.2242
253	0.2240
254	0.2240
255	0.2241
256	0.2240
257	0.2239
258	0.2239
259	0.2239
260	0.2240
261	0.2240
262	0.2236
263	0.2239
264	0.2241
265	0.2237
266	0.2238
267	0.2241
268	0.2236
269	0.2239
270	0.2241
271	0.2243
272	0.2238
273	0.2240
274	0.2240
275	0.2240
276	0.2240
277	0.2239
278	0.2238

0.34 #### Interpretation

This random forest model is not performing too poorly. Grid Search CV was used on max depth, number of splits, and number of estimators, with tax depth typically being 7, number of estimators typically being 100, and a mixture of splits from 2 to 4. There is an overfitting problem with the training AUC scores being around 0.08 higher than the validation scores. The validation scores are higher than the logistic regression, and with the RMSLE scores not being too far apart from each other, the overfitting is not too significant. This would suggest that this random forest model is performing as one of the best models so far.

0.35 ## XGB

[27]:
```
for group_name, features in models.items():
    start_time = time.time()  # Timer start

    steps = [
        ("scale_features", ColumnTransformer([("scale_numeric_features",␣
 ↪MinMaxScaler(), features)], remainder='drop')),
        ("xgb", xgb.XGBClassifier(use_label_encoder=False,␣
 ↪eval_metric='logloss'))
    ]
    pipe_xgb = Pipeline(steps)

    # the parameter grid
    param_grid = {
        "xgb__n_estimators": [100, 200],
        "xgb__max_depth": [3, 5, 7],
        "xgb__learning_rate": [0.01, 0.1]
    }

    # Initialize GridSearchCV
    grid_search = GridSearchCV(pipe_xgb, param_grid, cv=5, scoring='roc_auc',␣
 ↪n_jobs=-1)

    # Fit the model
    grid_search.fit(X_train[features], y_train)

    # Best model after grid search
    best_model = grid_search.best_estimator_

    # Predict probabilities
    train_prob = best_model.predict_proba(X_train[features])[:, 1]
    val_prob = best_model.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)
```

```python
    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    best_params = grid_search.best_params_
    new_row = pd.DataFrame([[f"{group_name} XGBoost", train_auc, val_auc,␣
↪train_rmsle, val_rmsle]],
                              columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {group_name} with best parameters {best_params} in␣
↪{end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 with best parameters {'xgb__learning_rate': 0.01, 'xgb__max_depth':
5, 'xgb__n_estimators': 200} in 3.63 seconds
Completed M2 with best parameters {'xgb__learning_rate': 0.01, 'xgb__max_depth':
5, 'xgb__n_estimators': 200} in 6.17 seconds
Completed M3 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 9.08 seconds
Completed M4 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 10.31 seconds
Completed M5 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 10.84 seconds
Completed M6 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 16.22 seconds
Completed M7 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 16.97 seconds
Completed M8 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 17.61 seconds
Completed M9 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 20.20 seconds
Completed M10 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 29.73 seconds
Completed M11 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 31.75 seconds
Completed M12 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 14.56 seconds
Completed M13 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 22.79 seconds
Completed M14 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 22.73 seconds
Completed M15 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
```

```
3, 'xgb__n_estimators': 100} in 26.80 seconds
Completed M16 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 27.66 seconds
Completed M17 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 28.88 seconds
Completed M18 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 48.34 seconds
Completed M19 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 49.82 seconds
Completed M20 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 64.94 seconds
Completed M21 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 80.72 seconds
Completed M22 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 18.37 seconds
Completed M23 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 23.03 seconds
Completed M24 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 29.36 seconds
Completed M25 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 37.33 seconds
Completed M26 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 40.43 seconds
Completed M27 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 55.82 seconds
Completed M28 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 68.48 seconds
Completed M29 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 71.14 seconds
Completed M30 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 97.89 seconds
Completed M31 with best parameters {'xgb__learning_rate': 0.1, 'xgb__max_depth':
3, 'xgb__n_estimators': 100} in 102.59 seconds
```

[27]:

```
             Model  Training AUC  Validation AUC  Training RMSLE  \
279   M1 XGBoost      0.665216        0.596593          0.2241
280   M2 XGBoost      0.719108        0.659764          0.2198
281   M3 XGBoost      0.747470        0.712439          0.2162
282   M4 XGBoost      0.759199        0.716104          0.2146
283   M5 XGBoost      0.759368        0.716645          0.2145
284   M6 XGBoost      0.768038        0.720897          0.2138
285   M7 XGBoost      0.769562        0.719086          0.2135
286   M8 XGBoost      0.769562        0.719086          0.2135
287   M9 XGBoost      0.769550        0.719653          0.2135
288  M10 XGBoost      0.771894        0.722050          0.2131
289  M11 XGBoost      0.774262        0.722890          0.2127
290  M12 XGBoost      0.767690        0.722454          0.2138
```

291	M13 XGBoost	0.770440	0.720937	0.2134
292	M14 XGBoost	0.769978	0.719701	0.2135
293	M15 XGBoost	0.769828	0.719955	0.2135
294	M16 XGBoost	0.769828	0.719955	0.2135
295	M17 XGBoost	0.773609	0.717837	0.2129
296	M18 XGBoost	0.776353	0.720049	0.2124
297	M19 XGBoost	0.778091	0.722872	0.2123
298	M20 XGBoost	0.778148	0.722158	0.2122
299	M21 XGBoost	0.780335	0.721468	0.2118
300	M22 XGBoost	0.769562	0.719086	0.2135
301	M23 XGBoost	0.769511	0.719509	0.2136
302	M24 XGBoost	0.769979	0.719701	0.2135
303	M25 XGBoost	0.769828	0.719955	0.2135
304	M26 XGBoost	0.769828	0.719955	0.2135
305	M27 XGBoost	0.773841	0.718111	0.2128
306	M28 XGBoost	0.776852	0.721188	0.2123
307	M29 XGBoost	0.776864	0.720156	0.2122
308	M30 XGBoost	0.780568	0.721286	0.2117
309	M31 XGBoost	0.780335	0.721468	0.2118

	Validation RMSLE
279	0.2309
280	0.2287
281	0.2243
282	0.2243
283	0.2242
284	0.2241
285	0.2242
286	0.2242
287	0.2241
288	0.2235
289	0.2236
290	0.2237
291	0.2239
292	0.2241
293	0.2241
294	0.2241
295	0.2241
296	0.2241
297	0.2240
298	0.2241
299	0.2237
300	0.2242
301	0.2241
302	0.2241
303	0.2241
304	0.2241

```
305           0.2241
306           0.2237
307           0.2239
308           0.2237
309           0.2237
```

0.36 #### Interpretation

The XG Boost model does seem to be a slight improvement as compared to the random forest. The gap between the training and validation AUC scores seems to have been minimized slightly, and the validation AUC reached 0.72, which is the highest so far. This model seems to be performing very well compared to the previous models.

0.37 ## Light Gradient Boosting Model

LightGBM is a fast and efficient gradient boosting framework that uses tree-based learning. It's designed to be faster and use less memory than other methods, making it great for handling large datasets. LightGBM can also work with categorical features directly, which simplifies the data preparation process. This makes it a preferred choice for many machine learning tasks that involve large amounts of data.

0.38 ### Simple Light Gradient Boosting

[28]:
```
for group_name, features in models.items():
    start_time = time.time()

    # creating datasets for LightGBM
    lgb_train = lgb.Dataset(X_train[features], label=y_train)
    lgb_val = lgb.Dataset(X_val[features], label=y_val, reference=lgb_train)

    # simplified params
    params = {
        'objective': 'binary',
        'metric': 'auc',
        'verbose': -1,
        'random_state': 20240325
    }

    # Train model
    num_boost_round = 100
    lgb_model = lgb.train(params,
                          lgb_train,
                          num_boost_round=num_boost_round,
                          valid_sets=[lgb_val])

    # predict
```

```
    train_prob = lgb_model.predict(X_train[features], num_iteration=lgb_model.
 ↪best_iteration)
    val_prob = lgb_model.predict(X_val[features], num_iteration=lgb_model.
 ↪best_iteration)

    # calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(y_train, train_prob)
    val_rmsle = calculateRMSLE(y_val, val_prob)

    # Append results
    new_row = pd.DataFrame([[f"{group_name} LightGBM Simple", train_auc,␣
 ↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
 ↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 0.61 seconds
Completed M2 in 0.52 seconds
Completed M3 in 0.56 seconds
Completed M4 in 0.56 seconds
Completed M5 in 0.57 seconds
Completed M6 in 0.64 seconds
Completed M7 in 0.65 seconds
Completed M8 in 0.68 seconds
Completed M9 in 0.71 seconds
Completed M10 in 0.79 seconds
Completed M11 in 0.83 seconds
Completed M12 in 0.63 seconds
Completed M13 in 0.77 seconds
Completed M14 in 0.89 seconds
Completed M15 in 0.91 seconds
Completed M16 in 0.93 seconds
Completed M17 in 0.91 seconds
Completed M18 in 1.13 seconds
Completed M19 in 1.19 seconds
Completed M20 in 1.16 seconds
Completed M21 in 1.45 seconds
Completed M22 in 0.70 seconds
```

```
Completed M23 in 0.74 seconds
Completed M24 in 0.82 seconds
Completed M25 in 0.91 seconds
Completed M26 in 0.92 seconds
Completed M27 in 1.05 seconds
Completed M28 in 1.16 seconds
Completed M29 in 1.26 seconds
Completed M30 in 1.70 seconds
Completed M31 in 1.73 seconds
```

[28]:

	Model	Training AUC	Validation AUC	Training RMSLE	\
310	M1 LightGBM Simple	0.842516	0.591113	0.2070	
311	M2 LightGBM Simple	0.892884	0.641908	0.1952	
312	M3 LightGBM Simple	0.916443	0.703488	0.1869	
313	M4 LightGBM Simple	0.919905	0.706887	0.1838	
314	M5 LightGBM Simple	0.915052	0.706785	0.1845	
315	M6 LightGBM Simple	0.933078	0.711330	0.1802	
316	M7 LightGBM Simple	0.937193	0.708391	0.1791	
317	M8 LightGBM Simple	0.937193	0.708391	0.1791	
318	M9 LightGBM Simple	0.937193	0.708391	0.1791	
319	M10 LightGBM Simple	0.937184	0.709963	0.1774	
320	M11 LightGBM Simple	0.938791	0.716654	0.1774	
321	M12 LightGBM Simple	0.931819	0.716769	0.1805	
322	M13 LightGBM Simple	0.939260	0.714709	0.1791	
323	M14 LightGBM Simple	0.937708	0.711139	0.1786	
324	M15 LightGBM Simple	0.939260	0.714709	0.1791	
325	M16 LightGBM Simple	0.939260	0.714709	0.1791	
326	M17 LightGBM Simple	0.939677	0.709743	0.1756	
327	M18 LightGBM Simple	0.944691	0.714028	0.1732	
328	M19 LightGBM Simple	0.948079	0.711990	0.1730	
329	M20 LightGBM Simple	0.946868	0.713726	0.1733	
330	M21 LightGBM Simple	0.945169	0.709520	0.1724	
331	M22 LightGBM Simple	0.937193	0.708391	0.1791	
332	M23 LightGBM Simple	0.937193	0.708391	0.1791	
333	M24 LightGBM Simple	0.937708	0.711139	0.1786	
334	M25 LightGBM Simple	0.939260	0.714709	0.1791	
335	M26 LightGBM Simple	0.939260	0.714709	0.1791	
336	M27 LightGBM Simple	0.942648	0.714965	0.1763	
337	M28 LightGBM Simple	0.942749	0.711064	0.1740	
338	M29 LightGBM Simple	0.944236	0.719813	0.1745	
339	M30 LightGBM Simple	0.945702	0.716667	0.1726	
340	M31 LightGBM Simple	0.945169	0.709520	0.1724	

	Validation RMSLE
310	0.2311
311	0.2294
312	0.2251

```
313              0.2259
314              0.2261
315              0.2251
316              0.2256
317              0.2256
318              0.2256
319              0.2247
320              0.2245
321              0.2255
322              0.2250
323              0.2258
324              0.2250
325              0.2250
326              0.2255
327              0.2248
328              0.2249
329              0.2254
330              0.2251
331              0.2256
332              0.2256
333              0.2258
334              0.2250
335              0.2250
336              0.2249
337              0.2255
338              0.2246
339              0.2247
340              0.2251
```

0.39 #### Interpretation

The simple LightGBM seems to have some problematic overfitting. So this model is not performing the best. Additional parameters will be tested to try and reduce the overfitting problem. The validation AUC seems to be performing well, but with a training AUC 0.9+, there is clear overfitting.

0.40 ### Tuned Light Gradient Boosting

[29]:
```
for group_name, features in models.items():
    start_time = time.time()

    # Create datasets for LightGBM
    lgb_train = lgb.Dataset(X_train[features], label=y_train)
    lgb_val = lgb.Dataset(X_val[features], label=y_val, reference=lgb_train)

    # adjusting parameters to reduce overfitting
    params = {
```

```
        'objective': 'binary',
        'metric': 'auc',
        'learning_rate': 0.05,  # Lowered learning rate
        'num_leaves': 20,  # Fewer leaves
        'lambda_l1': 0.5,  # Added L1 regularization
        'lambda_l2': 0.5,  # Added L2 regularization
        'verbose': -1,
        'random_state': 20240325
    }

    # Train model
    lgb_model = lgb.train(params,
                          lgb_train,
                          valid_sets=[lgb_val],
                          num_boost_round=1000)  # Maximum number of boosting␣
↪rounds

    # Prediction
    train_prob = lgb_model.predict(X_train[features], num_iteration=lgb_model.
↪best_iteration)
    val_prob = lgb_model.predict(X_val[features], num_iteration=lgb_model.
↪best_iteration)

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(y_train, train_prob)
    val_rmsle = calculateRMSLE(y_val, val_prob)

    # Append results
    new_row = pd.DataFrame([[f"{group_name} LightGBM Tuned", train_auc,␣
↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")


results_df.tail(31)
```

```
Completed M1 in 4.61 seconds
Completed M2 in 4.71 seconds
Completed M3 in 5.20 seconds
```

```
Completed M4 in 5.47 seconds
Completed M5 in 5.12 seconds
Completed M6 in 5.75 seconds
Completed M7 in 5.78 seconds
Completed M8 in 5.53 seconds
Completed M9 in 6.23 seconds
Completed M10 in 6.44 seconds
Completed M11 in 6.64 seconds
Completed M12 in 5.80 seconds
Completed M13 in 5.84 seconds
Completed M14 in 6.34 seconds
Completed M15 in 6.39 seconds
Completed M16 in 6.19 seconds
Completed M17 in 6.95 seconds
Completed M18 in 7.29 seconds
Completed M19 in 8.21 seconds
Completed M20 in 8.21 seconds
Completed M21 in 9.86 seconds
Completed M22 in 5.72 seconds
Completed M23 in 6.60 seconds
Completed M24 in 6.39 seconds
Completed M25 in 7.16 seconds
Completed M26 in 7.77 seconds
Completed M27 in 8.05 seconds
Completed M28 in 8.98 seconds
Completed M29 in 9.13 seconds
Completed M30 in 9.72 seconds
Completed M31 in 10.65 seconds
```

[29]:

```
                   Model  Training AUC  Validation AUC  Training RMSLE  \
341    M1 LightGBM Tuned      0.939188        0.573151          0.1857
342    M2 LightGBM Tuned      0.976504        0.618826          0.1605
343    M3 LightGBM Tuned      0.989452        0.687944          0.1446
344    M4 LightGBM Tuned      0.989442        0.699864          0.1414
345    M5 LightGBM Tuned      0.991744        0.698887          0.1390
346    M6 LightGBM Tuned      0.995478        0.706937          0.1317
347    M7 LightGBM Tuned      0.996543        0.703983          0.1292
348    M8 LightGBM Tuned      0.996543        0.703983          0.1292
349    M9 LightGBM Tuned      0.996543        0.703983          0.1292
350   M10 LightGBM Tuned      0.997476        0.704864          0.1250
351   M11 LightGBM Tuned      0.997174        0.702369          0.1253
352   M12 LightGBM Tuned      0.995949        0.710317          0.1312
353   M13 LightGBM Tuned      0.996994        0.702532          0.1283
354   M14 LightGBM Tuned      0.996551        0.701504          0.1296
355   M15 LightGBM Tuned      0.996994        0.702532          0.1283
356   M16 LightGBM Tuned      0.996994        0.702532          0.1283
357   M17 LightGBM Tuned      0.997594        0.705479          0.1237
```

358	M18 LightGBM Tuned	0.998806	0.703776	0.1166
359	M19 LightGBM Tuned	0.998681	0.702071	0.1167
360	M20 LightGBM Tuned	0.998657	0.705644	0.1158
361	M21 LightGBM Tuned	0.998908	0.706279	0.1134
362	M22 LightGBM Tuned	0.996543	0.703983	0.1292
363	M23 LightGBM Tuned	0.996543	0.703983	0.1292
364	M24 LightGBM Tuned	0.996482	0.702687	0.1296
365	M25 LightGBM Tuned	0.996959	0.703153	0.1283
366	M26 LightGBM Tuned	0.996959	0.703153	0.1283
367	M27 LightGBM Tuned	0.997354	0.702421	0.1230
368	M28 LightGBM Tuned	0.998384	0.706762	0.1197
369	M29 LightGBM Tuned	0.998254	0.703443	0.1189
370	M30 LightGBM Tuned	0.999104	0.709370	0.1134
371	M31 LightGBM Tuned	0.998908	0.706279	0.1134

	Validation RMSLE
341	0.2336
342	0.2321
343	0.2273
344	0.2273
345	0.2274
346	0.2261
347	0.2265
348	0.2265
349	0.2265
350	0.2257
351	0.2263
352	0.2255
353	0.2264
354	0.2265
355	0.2264
356	0.2264
357	0.2262
358	0.2266
359	0.2270
360	0.2267
361	0.2264
362	0.2265
363	0.2265
364	0.2265
365	0.2263
366	0.2263
367	0.2272
368	0.2266
369	0.2271
370	0.2255
371	0.2264

0.41 #### Interpretation

The number of leaves and the learning rate were decreased, with L1 and L2 regularization added to reduce overfitting. This did not reduce overfitting, but actually created more overfitting. So, the simplier LightGBM performed better on all of the models. Nevertheless, the XGB model is still performing better. Additionally, the training AUC was increased and the validation AUC was decreased in the tuned lightGBM.

0.42 ## Cat Boosting

CatBoost is a machine learning algorithm that is very good with categorical data, without the need for extensive preprocessing like one-hot encoding. It provides fast results and is designed to prevent overfitting, making it highly effective, especially in datasets with many categorical features. CatBoost automatically detects the best parameters for the model during training, simplifying the process of model tuning. This makes it highly user-friendly and effective for a wide range of regression and classification tasks.

0.42.1 Simple Cat Boost

[30]:
```
for group_name, features in models.items():
    start_time = time.time()

    # Defining CatBoost model
    cb_model = CatBoostClassifier(
        iterations=500,
        learning_rate=0.01,
        depth=4,
        random_state=20240325,
        verbose=False
    )

    # Fit the model
    cb_model.fit(X_train[features], y_train, eval_set=(X_val[features], y_val),␣
↪early_stopping_rounds=50, verbose=False)

    # Predict
    train_prob = cb_model.predict_proba(X_train[features])[:, 1]
    val_prob = cb_model.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(y_train, train_prob)
    val_rmsle = calculateRMSLE(y_val, val_prob)

    # Append results
```

```
    new_row = pd.DataFrame([[f"{group_name} CatBoost Simple", train_auc,␣
↪val_auc, train_rmsle, val_rmsle]],
                            columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 3.44 seconds
Completed M2 in 3.24 seconds
Completed M3 in 3.49 seconds
Completed M4 in 3.48 seconds
Completed M5 in 3.20 seconds
Completed M6 in 3.46 seconds
Completed M7 in 3.58 seconds
Completed M8 in 3.90 seconds
Completed M9 in 3.89 seconds
Completed M10 in 4.02 seconds
Completed M11 in 4.21 seconds
Completed M12 in 3.77 seconds
Completed M13 in 3.98 seconds
Completed M14 in 3.76 seconds
Completed M15 in 4.07 seconds
Completed M16 in 4.36 seconds
Completed M17 in 4.11 seconds
Completed M18 in 4.50 seconds
Completed M19 in 4.69 seconds
Completed M20 in 5.43 seconds
Completed M21 in 5.57 seconds
Completed M22 in 3.73 seconds
Completed M23 in 4.11 seconds
Completed M24 in 4.22 seconds
Completed M25 in 4.32 seconds
Completed M26 in 4.60 seconds
Completed M27 in 5.31 seconds
Completed M28 in 5.18 seconds
Completed M29 in 5.47 seconds
Completed M30 in 6.38 seconds
Completed M31 in 5.87 seconds
```

[30]:

```
                    Model  Training AUC  Validation AUC  Training RMSLE  \
372    M1 CatBoost Simple      0.618453        0.595543          0.2255
373    M2 CatBoost Simple      0.669739        0.656781          0.2227
374    M3 CatBoost Simple      0.707846        0.708040          0.2203
```

375	M4 CatBoost Simple	0.720150	0.714292	0.2191
376	M5 CatBoost Simple	0.720699	0.713801	0.2191
377	M6 CatBoost Simple	0.723002	0.715514	0.2190
378	M7 CatBoost Simple	0.723666	0.717000	0.2189
379	M8 CatBoost Simple	0.723455	0.716248	0.2189
380	M9 CatBoost Simple	0.723954	0.717557	0.2188
381	M10 CatBoost Simple	0.723738	0.715799	0.2189
382	M11 CatBoost Simple	0.724089	0.716550	0.2187
383	M12 CatBoost Simple	0.722991	0.717070	0.2190
384	M13 CatBoost Simple	0.723679	0.716868	0.2188
385	M14 CatBoost Simple	0.723793	0.716271	0.2189
386	M15 CatBoost Simple	0.723694	0.716679	0.2188
387	M16 CatBoost Simple	0.723898	0.716920	0.2188
388	M17 CatBoost Simple	0.724818	0.717518	0.2188
389	M18 CatBoost Simple	0.724918	0.717098	0.2187
390	M19 CatBoost Simple	0.725062	0.717590	0.2186
391	M20 CatBoost Simple	0.725010	0.717192	0.2186
392	M21 CatBoost Simple	0.724701	0.717163	0.2186
393	M22 CatBoost Simple	0.723191	0.716020	0.2189
394	M23 CatBoost Simple	0.723381	0.715913	0.2189
395	M24 CatBoost Simple	0.723900	0.716267	0.2188
396	M25 CatBoost Simple	0.723863	0.717047	0.2188
397	M26 CatBoost Simple	0.723881	0.715158	0.2188
398	M27 CatBoost Simple	0.725017	0.715846	0.2186
399	M28 CatBoost Simple	0.725118	0.716557	0.2186
400	M29 CatBoost Simple	0.724827	0.714946	0.2186
401	M30 CatBoost Simple	0.725759	0.716240	0.2185
402	M31 CatBoost Simple	0.725350	0.716669	0.2185

	Validation RMSLE
372	0.2303
373	0.2279
374	0.2244
375	0.2240
376	0.2241
377	0.2239
378	0.2238
379	0.2239
380	0.2238
381	0.2239
382	0.2239
383	0.2238
384	0.2238
385	0.2239
386	0.2238
387	0.2238
388	0.2238

```
389               0.2237
390               0.2237
391               0.2237
392               0.2237
393               0.2239
394               0.2239
395               0.2239
396               0.2238
397               0.2239
398               0.2239
399               0.2238
400               0.2239
401               0.2238
402               0.2238
```

0.43 #### Interpretation

This cat boost model is called simple since a more complex model was developed after this to test how the adjustment in parameters will influence the outcome. This "simple" model performs quite well, with no overfitting issue here. The training and validation AUC scores are very similar, and perform very well as compared to the XGB.The XGB does perform better since the validation AUC scores are slightly higher. A noteable observation from the simple cat boost model is how fast this model operated as compared to other models. This is something important for future projects to take into consideration, as pocessing time is important to reduce in larger scale projects.

0.43.1 Tuned Cat Boost

[31]:
```
for group_name, features in models.items():
    start_time = time.time()

    cb_model = CatBoostClassifier(
        iterations=2000,  # Explore more iterations for deeper learning
        learning_rate=0.001,  # Further reduce learning rate for more gradual␣
 ↪learning
        depth=7,  # Slightly increase depth for capturing more complex patterns
        l2_leaf_reg=5,  # Increase L2 regularization to control overfit depth's␣
 ↪complexity
        bagging_temperature=1,  # Introduce bagging for randomness, reducing␣
 ↪overfitting
        early_stopping_rounds=100,
        random_state=20240325,
        verbose=False)  # Use only a portion of data for each tree, increasing␣
 ↪diversity

    cb_model.fit(X_train[features], y_train, eval_set=(X_val[features], y_val),␣
 ↪early_stopping_rounds=50, verbose=False)
```

```
    train_prob = cb_model.predict_proba(X_train[features])[:, 1]
    val_prob = cb_model.predict_proba(X_val[features])[:, 1]

    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)


    new_row = pd.DataFrame([[f"{group_name} CatBoost Tuned", train_auc,␣
 ↪val_auc, train_rmsle, val_rmsle]],
                            columns=['Model', 'Training AUC', 'Validation AUC',␣
 ↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 16.63 seconds
Completed M2 in 17.90 seconds
Completed M3 in 20.68 seconds
Completed M4 in 21.23 seconds
Completed M5 in 21.07 seconds
Completed M6 in 24.97 seconds
Completed M7 in 26.55 seconds
Completed M8 in 27.29 seconds
Completed M9 in 29.09 seconds
Completed M10 in 33.91 seconds
Completed M11 in 35.61 seconds
Completed M12 in 24.77 seconds
Completed M13 in 30.18 seconds
Completed M14 in 30.29 seconds
Completed M15 in 32.84 seconds
Completed M16 in 32.79 seconds
Completed M17 in 34.28 seconds
Completed M18 in 45.00 seconds
Completed M19 in 45.59 seconds
Completed M20 in 52.07 seconds
Completed M21 in 58.94 seconds
Completed M22 in 28.65 seconds
Completed M23 in 30.82 seconds
Completed M24 in 34.09 seconds
Completed M25 in 38.57 seconds
```

```
Completed M26 in 39.91 seconds
Completed M27 in 47.98 seconds
Completed M28 in 53.11 seconds
Completed M29 in 55.45 seconds
Completed M30 in 66.49 seconds
Completed M31 in 66.22 seconds
```

[31]:

	Model	Training AUC	Validation AUC	Training RMSLE
403	M1 CatBoost Tuned	0.626457	0.593662	0.2306
404	M2 CatBoost Tuned	0.682740	0.655602	0.2274
405	M3 CatBoost Tuned	0.716299	0.704937	0.2244
406	M4 CatBoost Tuned	0.728570	0.711844	0.2229
407	M5 CatBoost Tuned	0.729916	0.711713	0.2228
408	M6 CatBoost Tuned	0.734208	0.712797	0.2226
409	M7 CatBoost Tuned	0.736047	0.712959	0.2225
410	M8 CatBoost Tuned	0.736876	0.713473	0.2224
411	M9 CatBoost Tuned	0.736553	0.713060	0.2225
412	M10 CatBoost Tuned	0.736948	0.713204	0.2224
413	M11 CatBoost Tuned	0.737306	0.713051	0.2224
414	M12 CatBoost Tuned	0.734932	0.714174	0.2226
415	M13 CatBoost Tuned	0.734330	0.713430	0.2225
416	M14 CatBoost Tuned	0.736037	0.713077	0.2225
417	M15 CatBoost Tuned	0.735528	0.714038	0.2225
418	M16 CatBoost Tuned	0.734902	0.713281	0.2224
419	M17 CatBoost Tuned	0.738524	0.712682	0.2223
420	M18 CatBoost Tuned	0.737946	0.714574	0.2221
421	M19 CatBoost Tuned	0.737564	0.714739	0.2220
422	M20 CatBoost Tuned	0.737282	0.714359	0.2221
423	M21 CatBoost Tuned	0.739480	0.715113	0.2219
424	M22 CatBoost Tuned	0.736325	0.712568	0.2224
425	M23 CatBoost Tuned	0.735849	0.712147	0.2225
426	M24 CatBoost Tuned	0.735431	0.712291	0.2226
427	M25 CatBoost Tuned	0.735819	0.714331	0.2224
428	M26 CatBoost Tuned	0.735275	0.713497	0.2223
429	M27 CatBoost Tuned	0.737269	0.713180	0.2222
430	M28 CatBoost Tuned	0.738768	0.713571	0.2221
431	M29 CatBoost Tuned	0.738677	0.713442	0.2221
432	M30 CatBoost Tuned	0.739003	0.714696	0.2220
433	M31 CatBoost Tuned	0.738665	0.714524	0.2219

\

	Validation RMSLE
403	0.2349
404	0.2324
405	0.2287
406	0.2280
407	0.2280
408	0.2280

```
409                0.2280
410                0.2280
411                0.2280
412                0.2280
413                0.2280
414                0.2279
415                0.2279
416                0.2281
417                0.2279
418                0.2279
419                0.2280
420                0.2278
421                0.2277
422                0.2278
423                0.2278
424                0.2281
425                0.2281
426                0.2281
427                0.2279
428                0.2279
429                0.2279
430                0.2279
431                0.2279
432                0.2278
433                0.2278
```

0.44 #### Interpretation

The further tuned cat boosting model does not perform as well. The training AUC increased, while the validaiton AUC did not change very much. Additionally, this model took significantly longer to run as compared to the simple cat boosting model from before. Therefore, the simple cat boosting model is optimal.

0.45 ## Explainable Boosting Machine

0.45.1 Simple EBM

[32]:
```
for group_name, features in models.items():
    start_time = time.time()  # Timer start

    # Adjusted EBM pipeline without SimpleImputer for numerical data
    ebm = ExplainableBoostingClassifier(random_state=20240325)

    # fit
    ebm.fit(X_train[features], y_train)

    # Predict probabilities
    train_prob = ebm.predict_proba(X_train[features])[:, 1]
```

```python
    val_prob = ebm.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    new_row = pd.DataFrame([[f"{group_name} EBM", train_auc, val_auc,␣
 ↪train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
 ↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 7.88 seconds
Completed M2 in 5.56 seconds
Completed M3 in 8.20 seconds
Completed M4 in 8.54 seconds
Completed M5 in 10.48 seconds
Completed M6 in 16.10 seconds
Completed M7 in 17.86 seconds
Completed M8 in 18.70 seconds
Completed M9 in 22.03 seconds
Completed M10 in 29.12 seconds
Completed M11 in 32.21 seconds
Completed M12 in 15.36 seconds
Completed M13 in 23.41 seconds
Completed M14 in 23.31 seconds
Completed M15 in 27.13 seconds
Completed M16 in 28.41 seconds
Completed M17 in 28.71 seconds
Completed M18 in 45.57 seconds
Completed M19 in 46.40 seconds
Completed M20 in 58.07 seconds
Completed M21 in 71.17 seconds
Completed M22 in 20.55 seconds
Completed M23 in 24.68 seconds
Completed M24 in 30.64 seconds
Completed M25 in 37.38 seconds
```

```
Completed M26 in 38.50 seconds
Completed M27 in 51.10 seconds
Completed M28 in 60.19 seconds
Completed M29 in 62.76 seconds
Completed M30 in 80.82 seconds
Completed M31 in 82.06 seconds
```

[32]:

	Model	Training AUC	Validation AUC	Training RMSLE	Validation RMSLE
434	M1 EBM	0.628037	0.599450	0.2246	0.2300
435	M2 EBM	0.687042	0.658695	0.2211	0.2276
436	M3 EBM	0.727854	0.710408	0.2182	0.2242
437	M4 EBM	0.743061	0.718112	0.2165	0.2240
438	M5 EBM	0.744391	0.718935	0.2165	0.2239
439	M6 EBM	0.761945	0.724416	0.2143	0.2235
440	M7 EBM	0.765401	0.727386	0.2137	0.2232
441	M8 EBM	0.766168	0.726259	0.2137	0.2233
442	M9 EBM	0.762717	0.728033	0.2141	0.2232
443	M10 EBM	0.766798	0.726993	0.2136	0.2231
444	M11 EBM	0.766312	0.726887	0.2136	0.2232
445	M12 EBM	0.760207	0.727088	0.2145	0.2231
446	M13 EBM	0.760188	0.725832	0.2144	0.2232
447	M14 EBM	0.761601	0.726991	0.2143	0.2232
448	M15 EBM	0.760804	0.726864	0.2144	0.2231
449	M16 EBM	0.761924	0.725496	0.2142	0.2233
450	M17 EBM	0.771508	0.724410	0.2131	0.2235
451	M18 EBM	0.783207	0.727065	0.2114	0.2233
452	M19 EBM	0.790839	0.726561	0.2102	0.2234
453	M20 EBM	0.783635	0.726916	0.2114	0.2233
454	M21 EBM	0.783100	0.725566	0.2115	0.2234
455	M22 EBM	0.765821	0.726432	0.2137	0.2234
456	M23 EBM	0.762217	0.729273	0.2141	0.2230
457	M24 EBM	0.760277	0.726308	0.2144	0.2233
458	M25 EBM	0.759795	0.726340	0.2145	0.2231
459	M26 EBM	0.756982	0.725016	0.2149	0.2233
460	M27 EBM	0.762477	0.724901	0.2144	0.2234
461	M28 EBM	0.763246	0.724333	0.2144	0.2234
462	M29 EBM	0.758858	0.725428	0.2150	0.2231
463	M30 EBM	0.779160	0.725136	0.2121	0.2234
464	M31 EBM	0.781066	0.726712	0.2118	0.2233

0.46 #### Interpretation

This model is the best performing model consistently across each of the variable group models. Although this model takes longer than the simple cat boosting model, the performance of this model out performs other models with the validation AUC. The permutation importance was conducted on this model, specifically M7, as this was one of the best performing models out of variable groups.

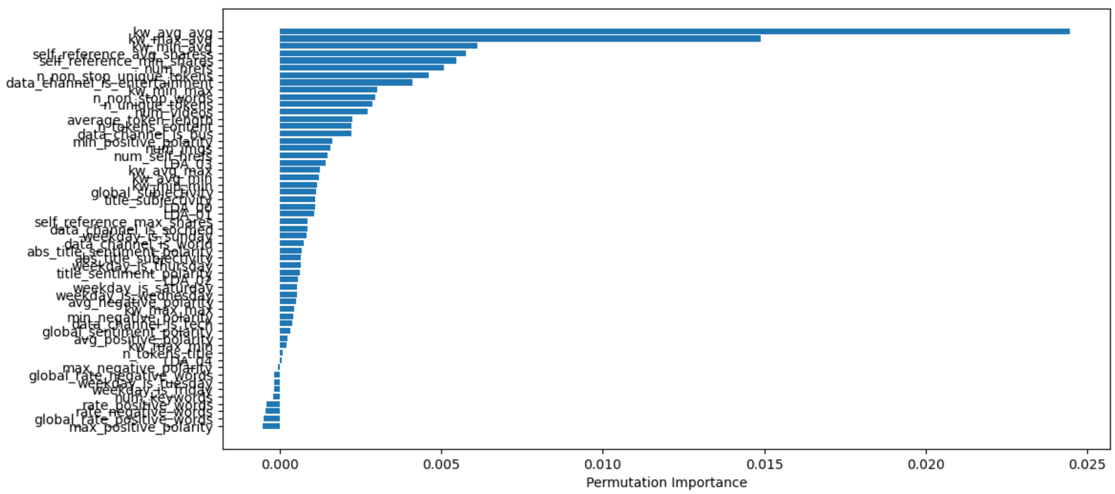
0.46.1 Permutation Importance

[33]:
```
ebm = ExplainableBoostingClassifier(random_state=20240325)
ebm.fit(X_train[models['M7']], y_train)

# computing the permutation based feature importance
perm_importance = permutation_importance(ebm, X_val[models['M7']], y_val,␣
↪n_repeats=10, random_state=42, scoring='roc_auc')

# retrieving and displaying feature importances
feature_names = np.array(models['M7'])
sorted_idx = perm_importance.importances_mean.argsort()

plt.figure(figsize=(12, 6))
plt.barh(feature_names[sorted_idx], perm_importance.
↪importances_mean[sorted_idx])
plt.xlabel("Permutation Importance")
plt.show()
```



[34]:
```
feature_names = np.array(models['M7'])

# identifying features with positive permutation importance values
positive_importance_features = feature_names[perm_importance.importances_mean >␣
↪0]

print("Features with positive permutation importance:")
for feature in positive_importance_features:
    print(feature)

# create a variable group with these features
```

```python
perm_importance_positive = positive_importance_features.tolist()

print("Variable group with positive permutation importance:")
print(perm_importance_positive)
```

```
Features with positive permutation importance:
n_tokens_title
n_tokens_content
n_unique_tokens
n_non_stop_words
n_non_stop_unique_tokens
average_token_length
num_hrefs
num_self_hrefs
num_imgs
num_videos
global_subjectivity
global_sentiment_polarity
kw_min_min
kw_max_min
kw_avg_min
kw_min_max
kw_max_max
kw_avg_max
kw_min_avg
kw_max_avg
kw_avg_avg
self_reference_min_shares
self_reference_max_shares
self_reference_avg_sharess
weekday_is_wednesday
weekday_is_thursday
weekday_is_saturday
weekday_is_sunday
data_channel_is_entertainment
data_channel_is_bus
data_channel_is_socmed
data_channel_is_tech
data_channel_is_world
LDA_00
LDA_01
LDA_02
LDA_03
LDA_04
avg_positive_polarity
min_positive_polarity
avg_negative_polarity
```

```
min_negative_polarity
title_subjectivity
title_sentiment_polarity
abs_title_subjectivity
abs_title_sentiment_polarity
Variable group with positive permutation importance:
['n_tokens_title', 'n_tokens_content', 'n_unique_tokens', 'n_non_stop_words',
'n_non_stop_unique_tokens', 'average_token_length', 'num_hrefs',
'num_self_hrefs', 'num_imgs', 'num_videos', 'global_subjectivity',
'global_sentiment_polarity', 'kw_min_min', 'kw_max_min', 'kw_avg_min',
'kw_min_max', 'kw_max_max', 'kw_avg_max', 'kw_min_avg', 'kw_max_avg',
'kw_avg_avg', 'self_reference_min_shares', 'self_reference_max_shares',
'self_reference_avg_sharess', 'weekday_is_wednesday', 'weekday_is_thursday',
'weekday_is_saturday', 'weekday_is_sunday', 'data_channel_is_entertainment',
'data_channel_is_bus', 'data_channel_is_socmed', 'data_channel_is_tech',
'data_channel_is_world', 'LDA_00', 'LDA_01', 'LDA_02', 'LDA_03', 'LDA_04',
'avg_positive_polarity', 'min_positive_polarity', 'avg_negative_polarity',
'min_negative_polarity', 'title_subjectivity', 'title_sentiment_polarity',
'abs_title_subjectivity', 'abs_title_sentiment_polarity']
```

0.47 ### Adjusted EBM 1

[35]:
```
for group_name, features in models.items():
    start_time = time.time()  # Timer start

    # Adjusted EBM pipeline without SimpleImputer for numerical data
    ebm_adjusted = ExplainableBoostingClassifier(
        random_state=20240325,
        learning_rate=0.01, # adjusted learning rate
        max_bins=256, # set max bins
        interactions=10, # set interactions
        early_stopping_rounds=50 # set early stopping roungs
    )

    # fit
    ebm_adjusted.fit(X_train[features], y_train)

    # Predict probabilities
    train_prob = ebm_adjusted.predict_proba(X_train[features])[:, 1]
    val_prob = ebm_adjusted.predict_proba(X_val[features])[:, 1]

    # Calculate AUC
    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    # Calculate RMSLE
    train_rmsle = calculateRMSLE(train_prob, y_train)
```

```
    val_rmsle = calculateRMSLE(val_prob, y_val)

    # Append results
    new_row = pd.DataFrame([[f"{group_name} EBM Adjusted 1", train_auc,␣
↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 3.86 seconds
Completed M2 in 4.43 seconds
Completed M3 in 5.70 seconds
Completed M4 in 6.81 seconds
Completed M5 in 6.95 seconds
Completed M6 in 12.10 seconds
Completed M7 in 12.17 seconds
Completed M8 in 12.84 seconds
Completed M9 in 13.69 seconds
Completed M10 in 19.48 seconds
Completed M11 in 19.88 seconds
Completed M12 in 10.35 seconds
Completed M13 in 15.02 seconds
Completed M14 in 15.56 seconds
Completed M15 in 17.55 seconds
Completed M16 in 18.07 seconds
Completed M17 in 18.12 seconds
Completed M18 in 27.47 seconds
Completed M19 in 27.29 seconds
Completed M20 in 34.66 seconds
Completed M21 in 42.02 seconds
Completed M22 in 13.70 seconds
Completed M23 in 16.18 seconds
Completed M24 in 19.84 seconds
Completed M25 in 24.07 seconds
Completed M26 in 24.39 seconds
Completed M27 in 32.04 seconds
Completed M28 in 37.78 seconds
Completed M29 in 38.65 seconds
Completed M30 in 49.82 seconds
Completed M31 in 51.07 seconds
```

[35]:
```
                  Model  Training AUC  Validation AUC  Training RMSLE  \
465   M1 EBM Adjusted 1      0.636023        0.607193          0.2242
466   M2 EBM Adjusted 1      0.678342        0.658737          0.2218
467   M3 EBM Adjusted 1      0.718354        0.709401          0.2193
468   M4 EBM Adjusted 1      0.728440        0.718318          0.2185
469   M5 EBM Adjusted 1      0.729174        0.719371          0.2184
470   M6 EBM Adjusted 1      0.741476        0.724784          0.2172
471   M7 EBM Adjusted 1      0.742880        0.725028          0.2170
472   M8 EBM Adjusted 1      0.743572        0.725698          0.2169
473   M9 EBM Adjusted 1      0.738593        0.726296          0.2175
474  M10 EBM Adjusted 1      0.739305        0.727329          0.2174
475  M11 EBM Adjusted 1      0.738411        0.726542          0.2175
476  M12 EBM Adjusted 1      0.741744        0.724843          0.2171
477  M13 EBM Adjusted 1      0.738455        0.725163          0.2175
478  M14 EBM Adjusted 1      0.740413        0.726406          0.2173
479  M15 EBM Adjusted 1      0.736548        0.725262          0.2178
480  M16 EBM Adjusted 1      0.737079        0.724997          0.2177
481  M17 EBM Adjusted 1      0.746843        0.726236          0.2167
482  M18 EBM Adjusted 1      0.755229        0.727098          0.2155
483  M19 EBM Adjusted 1      0.753422        0.728563          0.2156
484  M20 EBM Adjusted 1      0.748417        0.727667          0.2164
485  M21 EBM Adjusted 1      0.745826        0.727762          0.2167
486  M22 EBM Adjusted 1      0.741926        0.726765          0.2171
487  M23 EBM Adjusted 1      0.736410        0.725763          0.2178
488  M24 EBM Adjusted 1      0.736663        0.724825          0.2179
489  M25 EBM Adjusted 1      0.737436        0.724272          0.2177
490  M26 EBM Adjusted 1      0.733850        0.724753          0.2181
491  M27 EBM Adjusted 1      0.735912        0.724925          0.2180
492  M28 EBM Adjusted 1      0.737813        0.724208          0.2177
493  M29 EBM Adjusted 1      0.736539        0.724705          0.2178
494  M30 EBM Adjusted 1      0.743963        0.726811          0.2170
495  M31 EBM Adjusted 1      0.742422        0.726985          0.2172

     Validation RMSLE
465            0.2297
466            0.2277
467            0.2243
468            0.2239
469            0.2238
470            0.2235
471            0.2233
472            0.2233
473            0.2232
474            0.2231
475            0.2231
476            0.2234
477            0.2232
```

```
478              0.2232
479              0.2232
480              0.2233
481              0.2232
482              0.2231
483              0.2230
484              0.2230
485              0.2230
486              0.2232
487              0.2232
488              0.2234
489              0.2233
490              0.2233
491              0.2234
492              0.2233
493              0.2232
494              0.2230
495              0.2230
```

0.48 #### Interpretation

The adjusted EBM included the learning rate, max bins, interactions, and early stopping rounds to attempt to improve the validation AUC and reduce the overfitting. The overfitting was reduced, but these modifications did not necesraily increase the validation AUC, they stayed relatively the same. However, the processing time was reduced by a decent amount. So, this model would overall be a better model than the simple EBM, since it has less overfitting, similar validation AUC scores, and lower processing time.

0.49 ### Adjusted EBM 2

[36]:
```
for group_name, features in models.items():
    start_time = time.time()

    ebm_more_adjusted = ExplainableBoostingClassifier(
        random_state=20240325,
        learning_rate=0.005,  # Slightly lower learning rate for more␣
 ↪fine-grained adjustments
        max_bins=512,  # Increased number of bins for potentially capturing␣
 ↪more detail
        interactions=15,  # Allowing for more interactions
        early_stopping_rounds=100,  # More patience on early stopping to allow␣
 ↪more rounds for convergence
        n_jobs=-1  # Utilize all CPU cores for faster training
    )

    ebm_more_adjusted.fit(X_train[features], y_train)
```

```
    train_prob = ebm_more_adjusted.predict_proba(X_train[features])[:, 1]
    val_prob = ebm_more_adjusted.predict_proba(X_val[features])[:, 1]

    train_auc = roc_auc_score(y_train, train_prob)
    val_auc = roc_auc_score(y_val, val_prob)

    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    new_row = pd.DataFrame([[f"{group_name} EBM Adjusted 2", train_auc,␣
 ↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
 ↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {group_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 7.39 seconds
Completed M2 in 7.72 seconds
Completed M3 in 9.87 seconds
Completed M4 in 10.86 seconds
Completed M5 in 11.57 seconds
Completed M6 in 17.13 seconds
Completed M7 in 18.03 seconds
Completed M8 in 19.27 seconds
Completed M9 in 21.18 seconds
Completed M10 in 27.25 seconds
Completed M11 in 29.29 seconds
Completed M12 in 15.10 seconds
Completed M13 in 22.38 seconds
Completed M14 in 23.96 seconds
Completed M15 in 25.75 seconds
Completed M16 in 26.29 seconds
Completed M17 in 26.70 seconds
Completed M18 in 42.31 seconds
Completed M19 in 42.00 seconds
Completed M20 in 51.85 seconds
Completed M21 in 62.51 seconds
Completed M22 in 20.08 seconds
Completed M23 in 23.93 seconds
Completed M24 in 28.01 seconds
Completed M25 in 33.66 seconds
Completed M26 in 35.50 seconds
Completed M27 in 44.24 seconds
```

```
Completed M28 in 53.32 seconds
Completed M29 in 55.19 seconds
Completed M30 in 73.72 seconds
Completed M31 in 73.84 seconds
```

[36]:

	Model	Training AUC	Validation AUC	Training RMSLE
496	M1 EBM Adjusted 2	0.640790	0.604144	0.2240
497	M2 EBM Adjusted 2	0.685208	0.658450	0.2213
498	M3 EBM Adjusted 2	0.725316	0.712219	0.2187
499	M4 EBM Adjusted 2	0.735612	0.719000	0.2175
500	M5 EBM Adjusted 2	0.736297	0.719678	0.2176
501	M6 EBM Adjusted 2	0.745007	0.722964	0.2167
502	M7 EBM Adjusted 2	0.746603	0.725448	0.2165
503	M8 EBM Adjusted 2	0.746996	0.725894	0.2164
504	M9 EBM Adjusted 2	0.743411	0.725140	0.2170
505	M10 EBM Adjusted 2	0.740707	0.724105	0.2173
506	M11 EBM Adjusted 2	0.738092	0.725196	0.2175
507	M12 EBM Adjusted 2	0.742293	0.723971	0.2170
508	M13 EBM Adjusted 2	0.745445	0.724357	0.2167
509	M14 EBM Adjusted 2	0.745211	0.725595	0.2167
510	M15 EBM Adjusted 2	0.742294	0.724800	0.2171
511	M16 EBM Adjusted 2	0.743081	0.724533	0.2170
512	M17 EBM Adjusted 2	0.749817	0.726238	0.2162
513	M18 EBM Adjusted 2	0.759832	0.728308	0.2149
514	M19 EBM Adjusted 2	0.756544	0.729734	0.2153
515	M20 EBM Adjusted 2	0.753729	0.728196	0.2156
516	M21 EBM Adjusted 2	0.750283	0.726772	0.2161
517	M22 EBM Adjusted 2	0.747202	0.726052	0.2164
518	M23 EBM Adjusted 2	0.740450	0.724853	0.2174
519	M24 EBM Adjusted 2	0.742782	0.724551	0.2171
520	M25 EBM Adjusted 2	0.736484	0.725186	0.2178
521	M26 EBM Adjusted 2	0.734731	0.724442	0.2180
522	M27 EBM Adjusted 2	0.734562	0.724694	0.2181
523	M28 EBM Adjusted 2	0.736742	0.724625	0.2178
524	M29 EBM Adjusted 2	0.736133	0.724560	0.2179
525	M30 EBM Adjusted 2	0.747995	0.727965	0.2164
526	M31 EBM Adjusted 2	0.746505	0.726678	0.2166

\

	Validation RMSLE
496	0.2297
497	0.2277
498	0.2240
499	0.2238
500	0.2237
501	0.2235
502	0.2233
503	0.2233

```
504             0.2233
505             0.2233
506             0.2232
507             0.2233
508             0.2233
509             0.2232
510             0.2232
511             0.2233
512             0.2232
513             0.2231
514             0.2230
515             0.2230
516             0.2229
517             0.2233
518             0.2233
519             0.2233
520             0.2232
521             0.2233
522             0.2233
523             0.2232
524             0.2232
525             0.2228
526             0.2230
```

0.50 #### Interpretation

This model includes a slightly lower learning rate for more fine grained adjustments, an increased number of bins to attempt to capture more details, more interactions, more patience on early stopping to allow for more rounds of convergence, and utilizes all CPU cores to obtain faster training. This did not improve the processing time as compared to the simple EBM. There is less of an overfitting issues on this model as compared to the overfitting, but it does not necesraily improve the model as compared to the adjusted EBM 1 model, which runs much faster and has very similar results. Thus, the best model so far is the EBM 1 Adjusted.

0.51 ## Neural Network Models

Specific neural network models required that the data was transformed using the scaler or reshaped. This is done here so that the data transformations are easily visiable and able to be used throughout the NN models.

[37]:
```
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_val_scaled = scaler.transform(X_val)

n_features = X_train.shape[1]

# Reshape your data accordingly
X_train_reshaped = X_train_scaled.reshape((-1, n_features, 1))
```

```
X_val_reshaped = X_val_scaled.reshape((-1, n_features, 1))
```

0.52 ### Simple Neural Network Model 1

[38]:
```
for model_name, features in models.items():
    start_time = time.time()  # Timer start

    # Define the model
    model = Sequential([
        Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
        Dense(1, activation='sigmoid')
    ])

    # Compile the model
    model.compile(optimizer=Adam(), loss='binary_crossentropy',␣
↪metrics=[AUC(name='auc')])

    model.fit(X_train_scaled, y_train, epochs=100, batch_size=32, verbose=0,
              validation_data=(X_val_scaled, y_val),
              callbacks=[EarlyStopping(monitor='val_auc', patience=3,␣
↪restore_best_weights=True, mode='max')])

    _, train_auc = model.evaluate(X_train_scaled, y_train, verbose=0)
    _, val_auc = model.evaluate(X_val_scaled, y_val, verbose=0)

    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    new_row = pd.DataFrame([[f"{group_name} NN Simple", train_auc, val_auc,␣
↪train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()  # End timer
    print(f"Completed {model_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 7.06 seconds
Completed M2 in 8.11 seconds
Completed M3 in 7.84 seconds
Completed M4 in 6.69 seconds
Completed M5 in 8.90 seconds
Completed M6 in 7.41 seconds
Completed M7 in 7.84 seconds
Completed M8 in 7.43 seconds
```

```
Completed M9 in 7.79 seconds
Completed M10 in 6.70 seconds
Completed M11 in 9.24 seconds
Completed M12 in 8.11 seconds
Completed M13 in 6.24 seconds
Completed M14 in 8.25 seconds
Completed M15 in 6.54 seconds
Completed M16 in 8.50 seconds
Completed M17 in 6.86 seconds
Completed M18 in 4.98 seconds
Completed M19 in 4.65 seconds
Completed M20 in 6.30 seconds
Completed M21 in 5.61 seconds
Completed M22 in 6.75 seconds
Completed M23 in 5.55 seconds
Completed M24 in 9.07 seconds
Completed M25 in 10.98 seconds
Completed M26 in 8.35 seconds
Completed M27 in 8.71 seconds
Completed M28 in 8.41 seconds
Completed M29 in 9.44 seconds
Completed M30 in 6.08 seconds
Completed M31 in 5.09 seconds
```

[38]:

	Model	Training AUC	Validation AUC	Training RMSLE \
527	M31 NN Simple	0.741335	0.697379	0.2166
528	M31 NN Simple	0.762277	0.697792	0.2166
529	M31 NN Simple	0.763364	0.697828	0.2166
530	M31 NN Simple	0.748440	0.696561	0.2166
531	M31 NN Simple	0.763079	0.701149	0.2166
532	M31 NN Simple	0.759928	0.701576	0.2166
533	M31 NN Simple	0.753629	0.691433	0.2166
534	M31 NN Simple	0.759079	0.703278	0.2166
535	M31 NN Simple	0.756915	0.699650	0.2166
536	M31 NN Simple	0.747009	0.702391	0.2166
537	M31 NN Simple	0.760823	0.702124	0.2166
538	M31 NN Simple	0.764183	0.695668	0.2166
539	M31 NN Simple	0.748486	0.701846	0.2166
540	M31 NN Simple	0.764888	0.697487	0.2166
541	M31 NN Simple	0.740686	0.697459	0.2166
542	M31 NN Simple	0.764389	0.696239	0.2166
543	M31 NN Simple	0.745908	0.695658	0.2166
544	M31 NN Simple	0.713361	0.698802	0.2166
545	M31 NN Simple	0.714093	0.698813	0.2166
546	M31 NN Simple	0.727177	0.696039	0.2166
547	M31 NN Simple	0.735637	0.699231	0.2166
548	M31 NN Simple	0.754121	0.699695	0.2166

549	M31 NN Simple	0.736498	0.700157	0.2166
550	M31 NN Simple	0.769709	0.699743	0.2166
551	M31 NN Simple	0.775640	0.698962	0.2166
552	M31 NN Simple	0.762922	0.694514	0.2166
553	M31 NN Simple	0.768229	0.699622	0.2166
554	M31 NN Simple	0.764222	0.697586	0.2166
555	M31 NN Simple	0.766511	0.700379	0.2166
556	M31 NN Simple	0.744096	0.691332	0.2166
557	M31 NN Simple	0.722249	0.702998	0.2166

	Validation RMSLE
527	0.223
528	0.223
529	0.223
530	0.223
531	0.223
532	0.223
533	0.223
534	0.223
535	0.223
536	0.223
537	0.223
538	0.223
539	0.223
540	0.223
541	0.223
542	0.223
543	0.223
544	0.223
545	0.223
546	0.223
547	0.223
548	0.223
549	0.223
550	0.223
551	0.223
552	0.223
553	0.223
554	0.223
555	0.223
556	0.223
557	0.223

0.53 #### Interpretation

This simple neural network model performs quite well, but does not perform better than the adjust EBM 1. This model does perform quite fast compared to other models, but the validation AUC scores are not competitive enough with the other models. This model could be further improved, so modifications and adjustments will be made to improve the validation AUC.

0.54 ### Simple Neural Network Model 2

[39]:
```
for model_name, features in models.items():
    start_time = time.time()

    model = Sequential([
        Dense(32, activation='relu', input_shape=(X_train_scaled.shape[1],)),
        Dropout(0.5),
        Dense(16, activation='relu'),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer=Adam(learning_rate=0.001),␣
↪loss='binary_crossentropy', metrics=[AUC(name='auc')])

    model.fit(X_train_scaled, y_train, epochs=100, batch_size=32, verbose=0,
              validation_data=(X_val_scaled, y_val),
              callbacks=[EarlyStopping(monitor='val_auc', patience=5,␣
↪restore_best_weights=True, mode='max')])

    _, train_auc = model.evaluate(X_train_scaled, y_train, verbose=0)
    _, val_auc = model.evaluate(X_val_scaled, y_val, verbose=0)

    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    new_row = pd.DataFrame([[f"{group_name} NN Simple 2", train_auc, val_auc,␣
↪train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {model_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 21.17 seconds
Completed M2 in 14.99 seconds
Completed M3 in 19.28 seconds
Completed M4 in 28.84 seconds
Completed M5 in 16.66 seconds
Completed M6 in 22.48 seconds
```

```
Completed M7 in 20.77 seconds
Completed M8 in 13.06 seconds
Completed M9 in 10.00 seconds
Completed M10 in 16.34 seconds
Completed M11 in 15.49 seconds
Completed M12 in 21.39 seconds
Completed M13 in 22.12 seconds
Completed M14 in 14.68 seconds
Completed M15 in 15.50 seconds
Completed M16 in 13.02 seconds
Completed M17 in 16.72 seconds
Completed M18 in 14.25 seconds
Completed M19 in 20.13 seconds
Completed M20 in 14.43 seconds
Completed M21 in 15.43 seconds
Completed M22 in 19.43 seconds
Completed M23 in 15.39 seconds
Completed M24 in 22.35 seconds
Completed M25 in 19.67 seconds
Completed M26 in 22.62 seconds
Completed M27 in 17.53 seconds
Completed M28 in 11.37 seconds
Completed M29 in 18.73 seconds
Completed M30 in 19.35 seconds
Completed M31 in 14.61 seconds
```

[39]:

```
               Model  Training AUC  Validation AUC  Training RMSLE  \
558  M31 NN Simple 2      0.735007        0.711569          0.2166
559  M31 NN Simple 2      0.721606        0.709027          0.2166
560  M31 NN Simple 2      0.730091        0.712359          0.2166
561  M31 NN Simple 2      0.744425        0.713225          0.2166
562  M31 NN Simple 2      0.722596        0.710438          0.2166
563  M31 NN Simple 2      0.734413        0.709325          0.2166
564  M31 NN Simple 2      0.734427        0.713802          0.2166
565  M31 NN Simple 2      0.719356        0.713082          0.2166
566  M31 NN Simple 2      0.714103        0.711539          0.2166
567  M31 NN Simple 2      0.725969        0.711156          0.2166
568  M31 NN Simple 2      0.724496        0.711650          0.2166
569  M31 NN Simple 2      0.734949        0.712868          0.2166
570  M31 NN Simple 2      0.735455        0.713970          0.2166
571  M31 NN Simple 2      0.720712        0.710623          0.2166
572  M31 NN Simple 2      0.725633        0.712650          0.2166
573  M31 NN Simple 2      0.717147        0.711825          0.2166
574  M31 NN Simple 2      0.723081        0.711703          0.2166
575  M31 NN Simple 2      0.720874        0.712989          0.2166
576  M31 NN Simple 2      0.729694        0.712246          0.2166
577  M31 NN Simple 2      0.721147        0.710628          0.2166
```

578	M31 NN Simple 2	0.722792	0.711507	0.2166
579	M31 NN Simple 2	0.729305	0.712246	0.2166
580	M31 NN Simple 2	0.724951	0.710409	0.2166
581	M31 NN Simple 2	0.735535	0.712052	0.2166
582	M31 NN Simple 2	0.730657	0.714916	0.2166
583	M31 NN Simple 2	0.736531	0.715483	0.2166
584	M31 NN Simple 2	0.725283	0.709696	0.2166
585	M31 NN Simple 2	0.715883	0.709546	0.2166
586	M31 NN Simple 2	0.730486	0.715034	0.2166
587	M31 NN Simple 2	0.731919	0.709702	0.2166
588	M31 NN Simple 2	0.723552	0.709553	0.2166

	Validation RMSLE
558	0.223
559	0.223
560	0.223
561	0.223
562	0.223
563	0.223
564	0.223
565	0.223
566	0.223
567	0.223
568	0.223
569	0.223
570	0.223
571	0.223
572	0.223
573	0.223
574	0.223
575	0.223
576	0.223
577	0.223
578	0.223
579	0.223
580	0.223
581	0.223
582	0.223
583	0.223
584	0.223
585	0.223
586	0.223
587	0.223
588	0.223

0.55 #### Interpretation

This second model adds a dense layer as well as two dropout layers to avoid overfitting. The training time on this model is increased by a signficant amount, but does show signs of improving the validation AUC while maintaining insignificant overfitting. This model does perform better than the first simple neural network model, but does not perform better than the adjusted EBM 1.

0.56 ### Simple Neural Network Model 3

[40]:
```
for model_name, features in models.items():
    start_time = time.time()

    model = Sequential([
        Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
        Dropout(0.3),
        Dense(32, activation='relu'),
        Dropout(0.3),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer=Adam(learning_rate=0.0005),␣
↪loss='binary_crossentropy', metrics=[AUC(name='auc')])

    es = EarlyStopping(monitor='val_auc', patience=10,␣
↪restore_best_weights=True, mode='max')
    model.fit(X_train_scaled, y_train, epochs=150, batch_size=64, verbose=0,
              validation_data=(X_val_scaled, y_val),
              callbacks=[es])

    train_pred = model.predict(X_train_scaled).flatten()
    val_pred = model.predict(X_val_scaled).flatten()

    _, train_auc = model.evaluate(X_train_scaled, y_train, verbose=0)
    _, val_auc = model.evaluate(X_val_scaled, y_val, verbose=0)

    train_rmsle = calculateRMSLE(y_train, np.clip(train_pred, 0, None))  #␣
↪clipping predictions to ensure non-negative values
    val_rmsle = calculateRMSLE(y_val, np.clip(val_pred, 0, None))

    new_row = pd.DataFrame([[f"{model_name} NN Simple 3", train_auc, val_auc,␣
↪train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
```

```python
    print(f"Completed {model_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
744/744              1s 876us/step
186/186              0s 744us/step
Completed M1 in 20.57 seconds
744/744              1s 927us/step
186/186              0s 770us/step
Completed M2 in 18.99 seconds
744/744              1s 841us/step
186/186              0s 726us/step
Completed M3 in 13.40 seconds
744/744              1s 967us/step
186/186              0s 852us/step
Completed M4 in 20.52 seconds
744/744              1s 1ms/step
186/186              0s 849us/step
Completed M5 in 18.80 seconds
744/744              1s 854us/step
186/186              0s 799us/step
Completed M6 in 19.23 seconds
744/744              1s 948us/step
186/186              0s 731us/step
Completed M7 in 17.98 seconds
744/744              1s 860us/step
186/186              0s 760us/step
Completed M8 in 16.23 seconds
744/744              1s 904us/step
186/186              0s 762us/step
Completed M9 in 13.84 seconds
744/744              1s 873us/step
186/186              0s 758us/step
Completed M10 in 16.96 seconds
744/744              1s 923us/step
186/186              0s 773us/step
Completed M11 in 14.72 seconds
744/744              1s 969us/step
186/186              0s 854us/step
Completed M12 in 13.14 seconds
744/744              1s 974us/step
186/186              0s 874us/step
Completed M13 in 15.90 seconds
744/744              1s 980us/step
186/186              0s 819us/step
Completed M14 in 16.22 seconds
744/744              1s 1ms/step
```

```
186/186             0s 748us/step
Completed M15 in 17.53 seconds
744/744             1s 896us/step
186/186             0s 756us/step
Completed M16 in 20.48 seconds
744/744             1s 963us/step
186/186             0s 753us/step
Completed M17 in 14.79 seconds
744/744             1s 919us/step
186/186             0s 846us/step
Completed M18 in 24.28 seconds
744/744             1s 862us/step
186/186             0s 718us/step
Completed M19 in 25.68 seconds
744/744             1s 976us/step
186/186             0s 811us/step
Completed M20 in 18.39 seconds
744/744             1s 1ms/step
186/186             0s 752us/step
Completed M21 in 16.96 seconds
744/744             1s 892us/step
186/186             0s 867us/step
Completed M22 in 18.35 seconds
744/744             1s 903us/step
186/186             0s 834us/step
Completed M23 in 21.93 seconds
744/744             1s 905us/step
186/186             0s 796us/step
Completed M24 in 18.63 seconds
744/744             1s 885us/step
186/186             0s 718us/step
Completed M25 in 18.14 seconds
744/744             1s 902us/step
186/186             0s 827us/step
Completed M26 in 19.24 seconds
744/744             1s 898us/step
186/186             0s 759us/step
Completed M27 in 16.34 seconds
744/744             1s 964us/step
186/186             0s 876us/step
Completed M28 in 19.44 seconds
744/744             1s 898us/step
186/186             0s 774us/step
Completed M29 in 20.75 seconds
744/744             1s 903us/step
186/186             0s 913us/step
Completed M30 in 16.58 seconds
744/744             1s 871us/step
```

```
186/186            0s 726us/step
Completed M31 in 15.09 seconds
```

[40]:

	Model	Training AUC	Validation AUC	Training RMSLE
589	M1 NN Simple 3	0.773596	0.707178	0.2166
590	M2 NN Simple 3	0.762032	0.710935	0.2173
591	M3 NN Simple 3	0.737172	0.711257	0.2203
592	M4 NN Simple 3	0.764650	0.713795	0.2151
593	M5 NN Simple 3	0.761089	0.710670	0.2185
594	M6 NN Simple 3	0.764435	0.715658	0.2166
595	M7 NN Simple 3	0.755129	0.708435	0.2165
596	M8 NN Simple 3	0.753159	0.709823	0.2168
597	M9 NN Simple 3	0.738479	0.711350	0.2236
598	M10 NN Simple 3	0.758296	0.708929	0.2187
599	M11 NN Simple 3	0.742851	0.709935	0.2187
600	M12 NN Simple 3	0.740842	0.707439	0.2250
601	M13 NN Simple 3	0.754575	0.711732	0.2195
602	M14 NN Simple 3	0.755348	0.711463	0.2197
603	M15 NN Simple 3	0.759631	0.712694	0.2162
604	M16 NN Simple 3	0.770100	0.710377	0.2165
605	M17 NN Simple 3	0.742443	0.709034	0.2208
606	M18 NN Simple 3	0.778322	0.708297	0.2147
607	M19 NN Simple 3	0.787028	0.708725	0.2148
608	M20 NN Simple 3	0.766381	0.707720	0.2193
609	M21 NN Simple 3	0.752942	0.707485	0.2198
610	M22 NN Simple 3	0.762211	0.709287	0.2166
611	M23 NN Simple 3	0.777090	0.709640	0.2167
612	M24 NN Simple 3	0.764877	0.713710	0.2165
613	M25 NN Simple 3	0.761254	0.711644	0.2178
614	M26 NN Simple 3	0.765765	0.712541	0.2164
615	M27 NN Simple 3	0.754383	0.711643	0.2174
616	M28 NN Simple 3	0.765894	0.712757	0.2150
617	M29 NN Simple 3	0.774746	0.711561	0.2173
618	M30 NN Simple 3	0.751126	0.710251	0.2185
619	M31 NN Simple 3	0.749497	0.711788	0.2184

\

	Validation RMSLE
589	0.2271
590	0.2264
591	0.2264
592	0.2253
593	0.2273
594	0.2261
595	0.2254
596	0.2253
597	0.2289
598	0.2269

```
599               0.2257
600               0.2311
601               0.2276
602               0.2272
603               0.2248
604               0.2269
605               0.2270
606               0.2262
607               0.2279
608               0.2281
609               0.2284
610               0.2259
611               0.2281
612               0.2257
613               0.2266
614               0.2258
615               0.2250
616               0.2250
617               0.2279
618               0.2263
619               0.2261
```

0.57 #### Interpretation

This third neural network model does not significantly improve the scores as compared to the second model. An additional dense layer was added along with a decrease in the learning rate, but the processing time is the same while the overfitting is actually increased. Additionally, the number of epochs was raised from 100 to 150 for a more indepth look. So this model is not an ideal model, and the best model so far amongst the neural network models is the second simple neural network model.

0.58 ### Complex Neural Network Model

[41]:
```
for model_name, features in models.items():
    start_time = time.time()

    model = Sequential([
        Dense(128, activation='relu', input_shape=(X_train_scaled.shape[1],)),
        BatchNormalization(),
        Dropout(0.5),
        Dense(64, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),
        Dense(32, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),
        Dense(1, activation='sigmoid')
```

```
    ])

    model.compile(optimizer=Adam(learning_rate=0.001),␣
↪loss='binary_crossentropy', metrics=[AUC(name='auc')])

    model.fit(X_train_scaled, y_train, epochs=100, batch_size=32, verbose=0,
              validation_data=(X_val_scaled, y_val),
              callbacks=[EarlyStopping(monitor='val_auc', patience=5,␣
↪restore_best_weights=True, mode='max')])

    _, train_auc = model.evaluate(X_train_scaled, y_train, verbose=0)
    _, val_auc = model.evaluate(X_val_scaled, y_val, verbose=0)

    train_rmsle = calculateRMSLE(train_prob, y_train)
    val_rmsle = calculateRMSLE(val_prob, y_val)

    new_row = pd.DataFrame([[f"{model_name} NN Complex", train_auc, val_auc,␣
↪train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {model_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
Completed M1 in 24.89 seconds
Completed M2 in 30.25 seconds
Completed M3 in 31.14 seconds
Completed M4 in 23.07 seconds
Completed M5 in 23.19 seconds
Completed M6 in 27.37 seconds
Completed M7 in 28.28 seconds
Completed M8 in 37.69 seconds
Completed M9 in 33.52 seconds
Completed M10 in 30.38 seconds
Completed M11 in 28.78 seconds
Completed M12 in 21.76 seconds
Completed M13 in 32.14 seconds
Completed M14 in 24.58 seconds
Completed M15 in 31.40 seconds
Completed M16 in 23.17 seconds
Completed M17 in 28.93 seconds
Completed M18 in 23.60 seconds
Completed M19 in 29.49 seconds
Completed M20 in 33.59 seconds
```

```
Completed M21 in 28.65 seconds
Completed M22 in 29.43 seconds
Completed M23 in 29.17 seconds
Completed M24 in 23.39 seconds
Completed M25 in 27.52 seconds
Completed M26 in 27.00 seconds
Completed M27 in 31.97 seconds
Completed M28 in 28.92 seconds
Completed M29 in 34.23 seconds
Completed M30 in 26.95 seconds
Completed M31 in 27.46 seconds
```

[41]:

```
              Model  Training AUC  Validation AUC  Training RMSLE  \
620  M31 NN Complex      0.736321        0.711864          0.2166
621  M31 NN Complex      0.742054        0.715358          0.2166
622  M31 NN Complex      0.743683        0.717296          0.2166
623  M31 NN Complex      0.727441        0.709844          0.2166
624  M31 NN Complex      0.728448        0.714545          0.2166
625  M31 NN Complex      0.738683        0.714576          0.2166
626  M31 NN Complex      0.739094        0.712156          0.2166
627  M31 NN Complex      0.754559        0.714179          0.2166
628  M31 NN Complex      0.747513        0.715055          0.2166
629  M31 NN Complex      0.744133        0.720031          0.2166
630  M31 NN Complex      0.735812        0.713895          0.2166
631  M31 NN Complex      0.728398        0.713818          0.2166
632  M31 NN Complex      0.744699        0.713467          0.2166
633  M31 NN Complex      0.731408        0.714155          0.2166
634  M31 NN Complex      0.743502        0.716064          0.2166
635  M31 NN Complex      0.728712        0.712847          0.2166
636  M31 NN Complex      0.738614        0.714640          0.2166
637  M31 NN Complex      0.731331        0.712403          0.2166
638  M31 NN Complex      0.739741        0.714656          0.2166
639  M31 NN Complex      0.745247        0.715128          0.2166
640  M31 NN Complex      0.739494        0.717039          0.2166
641  M31 NN Complex      0.740445        0.714874          0.2166
642  M31 NN Complex      0.739989        0.710696          0.2166
643  M31 NN Complex      0.730830        0.714468          0.2166
644  M31 NN Complex      0.738327        0.715159          0.2166
645  M31 NN Complex      0.737683        0.713959          0.2166
646  M31 NN Complex      0.744693        0.715496          0.2166
647  M31 NN Complex      0.738888        0.713192          0.2166
648  M31 NN Complex      0.748208        0.718108          0.2166
649  M31 NN Complex      0.733689        0.714462          0.2166
650  M31 NN Complex      0.736184        0.713266          0.2166

     Validation RMSLE
620             0.223
```

```
621                 0.223
622                 0.223
623                 0.223
624                 0.223
625                 0.223
626                 0.223
627                 0.223
628                 0.223
629                 0.223
630                 0.223
631                 0.223
632                 0.223
633                 0.223
634                 0.223
635                 0.223
636                 0.223
637                 0.223
638                 0.223
639                 0.223
640                 0.223
641                 0.223
642                 0.223
643                 0.223
644                 0.223
645                 0.223
646                 0.223
647                 0.223
648                 0.223
649                 0.223
650                 0.223
```

0.59 #### Interpretation

This more complex neural network includes batch normalization layers along with additional dropout layers. The overfitting is reduced in this model as compared to the third simple neural network model, and performs similarly to the second neural network model. The processing time is increased in the more complex model here, so it is less ideal as compared to the second simple NN. Additionally, this model does not out perform the adjusted EBM 1.

0.60 ### Conv1D Adjusted Neural Network 1

[42]:
```
for model_name, features in models.items():
    start_time = time.time()

    model = Sequential([
        # Applying Conv1D on the reshaped data; treating each feature as a␣
↪timestep
```

```
        Conv1D(filters=32, kernel_size=1, activation='relu',␣
↪input_shape=(n_features, 1)),
        MaxPooling1D(pool_size=2, strides=2),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.3),
        Dense(64, activation='relu'),
        Dropout(0.3),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer=Adam(learning_rate=0.0001),
                  loss='binary_crossentropy', metrics=[AUC(name='auc')])

    es = EarlyStopping(monitor='val_auc', patience=15,␣
↪restore_best_weights=True, mode='max')
    model.fit(X_train_reshaped, y_train, epochs=200, batch_size=32, verbose=0,
              validation_data=(X_val_reshaped, y_val),
              callbacks=[es])

    _, train_auc = model.evaluate(X_train_reshaped, y_train, verbose=0)
    _, val_auc = model.evaluate(X_val_reshaped, y_val, verbose=0)

    # Prediction and RMSLE calculation need correct predictions
    train_pred = model.predict(X_train_reshaped).flatten()
    val_pred = model.predict(X_val_reshaped).flatten()

    train_rmsle = calculateRMSLE(y_train, np.clip(train_pred, 0, None))
    val_rmsle = calculateRMSLE(y_val, np.clip(val_pred, 0, None))

    new_row = pd.DataFrame([[f"{model_name} NN Conv1D Adjusted", train_auc,␣
↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {model_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M1 in 80.89 seconds
744/744              1s 2ms/step
186/186              0s 2ms/step
Completed M2 in 90.04 seconds
```

```
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M3 in 85.91 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M4 in 99.06 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M5 in 114.49 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M6 in 64.48 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M7 in 96.15 seconds
744/744              1s 2ms/step
186/186              0s 1ms/step
Completed M8 in 71.11 seconds
744/744              1s 2ms/step
186/186              0s 1ms/step
Completed M9 in 79.46 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M10 in 82.62 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M11 in 101.87 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M12 in 84.08 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M13 in 87.52 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M14 in 109.49 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M15 in 82.62 seconds
744/744              1s 2ms/step
186/186              0s 1ms/step
Completed M16 in 63.56 seconds
744/744              1s 1ms/step
186/186              0s 1ms/step
Completed M17 in 98.61 seconds
744/744              1s 2ms/step
186/186              0s 1ms/step
Completed M18 in 74.86 seconds
```

```
744/744             1s 2ms/step
186/186             0s 1ms/step
Completed M19 in 94.26 seconds
744/744             1s 2ms/step
186/186             0s 1ms/step
Completed M20 in 64.36 seconds
744/744             1s 1ms/step
186/186             0s 2ms/step
Completed M21 in 106.53 seconds
744/744             1s 1ms/step
186/186             0s 1ms/step
Completed M22 in 103.11 seconds
744/744             1s 1ms/step
186/186             0s 1ms/step
Completed M23 in 101.14 seconds
744/744             1s 1ms/step
186/186             0s 1ms/step
Completed M24 in 90.51 seconds
744/744             2s 1ms/step
186/186             0s 1ms/step
Completed M25 in 82.11 seconds
744/744             1s 2ms/step
186/186             0s 2ms/step
Completed M26 in 78.79 seconds
744/744             1s 1ms/step
186/186             0s 1ms/step
Completed M27 in 80.63 seconds
744/744             1s 2ms/step
186/186             0s 2ms/step
Completed M28 in 105.85 seconds
744/744             1s 2ms/step
186/186             0s 2ms/step
Completed M29 in 80.71 seconds
744/744             1s 1ms/step
186/186             0s 1ms/step
Completed M30 in 84.54 seconds
744/744             1s 2ms/step
186/186             0s 2ms/step
Completed M31 in 89.75 seconds
```

[42]:

```
                       Model  Training AUC  Validation AUC  Training RMSLE  \
651   M1 NN Conv1D Adjusted      0.736287        0.711109          0.2167
652   M2 NN Conv1D Adjusted      0.743762        0.713524          0.2220
653   M3 NN Conv1D Adjusted      0.746032        0.710604          0.2183
654   M4 NN Conv1D Adjusted      0.755695        0.710115          0.2132
655   M5 NN Conv1D Adjusted      0.769245        0.713502          0.2139
656   M6 NN Conv1D Adjusted      0.722870        0.712042          0.2209
```

657	M7 NN Conv1D Adjusted	0.756500	0.710696	0.2155
658	M8 NN Conv1D Adjusted	0.730564	0.710325	0.2168
659	M9 NN Conv1D Adjusted	0.744752	0.712093	0.2164
660	M10 NN Conv1D Adjusted	0.741162	0.711302	0.2184
661	M11 NN Conv1D Adjusted	0.763622	0.711633	0.2149
662	M12 NN Conv1D Adjusted	0.748451	0.711741	0.2181
663	M13 NN Conv1D Adjusted	0.750646	0.712623	0.2160
664	M14 NN Conv1D Adjusted	0.758349	0.708382	0.2132
665	M15 NN Conv1D Adjusted	0.745220	0.712818	0.2186
666	M16 NN Conv1D Adjusted	0.728950	0.710767	0.2219
667	M17 NN Conv1D Adjusted	0.757627	0.710231	0.2136
668	M18 NN Conv1D Adjusted	0.736233	0.710957	0.2167
669	M19 NN Conv1D Adjusted	0.750675	0.712858	0.2160
670	M20 NN Conv1D Adjusted	0.725361	0.709774	0.2208
671	M21 NN Conv1D Adjusted	0.766348	0.709069	0.2136
672	M22 NN Conv1D Adjusted	0.760813	0.708625	0.2160
673	M23 NN Conv1D Adjusted	0.766132	0.710065	0.2131
674	M24 NN Conv1D Adjusted	0.750844	0.710358	0.2164
675	M25 NN Conv1D Adjusted	0.747387	0.710534	0.2170
676	M26 NN Conv1D Adjusted	0.742208	0.709871	0.2197
677	M27 NN Conv1D Adjusted	0.735802	0.710878	0.2162
678	M28 NN Conv1D Adjusted	0.761499	0.712150	0.2143
679	M29 NN Conv1D Adjusted	0.739853	0.711530	0.2197
680	M30 NN Conv1D Adjusted	0.745729	0.710258	0.2176
681	M31 NN Conv1D Adjusted	0.744512	0.710516	0.2160

	Validation RMSLE
651	0.2242
652	0.2294
653	0.2265
654	0.2232
655	0.2257
656	0.2259
657	0.2254
658	0.2234
659	0.2251
660	0.2261
661	0.2260
662	0.2264
663	0.2255
664	0.2241
665	0.2262
666	0.2277
667	0.2240
668	0.2240
669	0.2251
670	0.2261

```
671             0.2258
672             0.2268
673             0.2248
674             0.2254
675             0.2257
676             0.2269
677             0.2236
678             0.2249
679             0.2271
680             0.2255
681             0.2243
```

0.61 #### Interpretation

This neural networks includes a convolutional layer, along with max pooling, multiple dense layers, and an increased number of epochs for a deeper look. The performance of this model is similar to other NN models, but since it is significantly more complex and takes drastically longer to process, this model is not ideal since the validation AUC scores are not altered. Additionally, there is a bit of an overfitting increase as compared to the second simple NN. As a result of this, the second simple NN is preferred still.

0.62 ### Conv1D Adjusted Neural Network 2

[43]:
```
for model_name, features in models.items():
    start_time = time.time()

    model = Sequential([
        Conv1D(filters=64, kernel_size=1, activation='relu',␣
 ↪input_shape=(n_features, 1)),
        MaxPooling1D(pool_size=2),
        Conv1D(filters=64, kernel_size=1, activation='relu'),  # Additional␣
 ↪Conv layer
        MaxPooling1D(pool_size=2),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.4),  # Slightly increased dropout
        Dense(64, activation='relu'),
        Dropout(0.4),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer=Adam(learning_rate=0.0005),  # Increased learning␣
 ↪rate
                  loss='binary_crossentropy', metrics=[AUC(name='auc')])

    es = EarlyStopping(monitor='val_auc', patience=10,␣
 ↪restore_best_weights=True, mode='max')  # Adjusted patience
```

```python
    model.fit(X_train_reshaped, y_train, epochs=100, batch_size=64, verbose=0,␣
↪# Reduced epochs, increased batch size
              validation_data=(X_val_reshaped, y_val),
              callbacks=[es])

    _, train_auc = model.evaluate(X_train_reshaped, y_train, verbose=0)
    _, val_auc = model.evaluate(X_val_reshaped, y_val, verbose=0)

    train_pred = model.predict(X_train_reshaped).flatten()
    val_pred = model.predict(X_val_reshaped).flatten()

    train_rmsle = calculateRMSLE(y_train, np.clip(train_pred, 0, None))
    val_rmsle = calculateRMSLE(y_val, np.clip(val_pred, 0, None))

    new_row = pd.DataFrame([[f"{model_name} NN Conv1D Optimized 2", train_auc,␣
↪val_auc, train_rmsle, val_rmsle]],
                           columns=['Model', 'Training AUC', 'Validation AUC',␣
↪'Training RMSLE', 'Validation RMSLE'])
    results_df = pd.concat([results_df, new_row], ignore_index=True)

    end_time = time.time()
    print(f"Completed {model_name} in {end_time - start_time:.2f} seconds")

results_df.tail(31)
```

```
744/744               2s 2ms/step
186/186               0s 2ms/step
Completed M1 in 73.77 seconds
744/744               2s 2ms/step
186/186               0s 2ms/step
Completed M2 in 78.44 seconds
744/744               2s 2ms/step
186/186               0s 2ms/step
Completed M3 in 72.46 seconds
744/744               2s 2ms/step
186/186               0s 2ms/step
Completed M4 in 77.57 seconds
744/744               2s 2ms/step
186/186               0s 2ms/step
Completed M5 in 80.59 seconds
744/744               2s 2ms/step
186/186               0s 2ms/step
Completed M6 in 69.26 seconds
744/744               2s 2ms/step
186/186               0s 2ms/step
Completed M7 in 73.79 seconds
744/744               2s 2ms/step
```

```
186/186             0s 2ms/step
Completed M8 in 77.46 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M9 in 80.90 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M10 in 80.43 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M11 in 83.20 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M12 in 68.57 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M13 in 55.52 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M14 in 71.44 seconds
744/744             1s 2ms/step
186/186             0s 2ms/step
Completed M15 in 57.33 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M16 in 62.23 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M17 in 60.96 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M18 in 70.59 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M19 in 55.46 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M20 in 65.83 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M21 in 72.91 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M22 in 84.75 seconds
744/744             2s 2ms/step
186/186             0s 2ms/step
Completed M23 in 75.36 seconds
744/744             2s 2ms/step
```

```
186/186              0s 2ms/step
Completed M24 in 73.09 seconds
744/744              2s 2ms/step
186/186              0s 2ms/step
Completed M25 in 71.03 seconds
744/744              2s 2ms/step
186/186              0s 2ms/step
Completed M26 in 66.63 seconds
744/744              2s 2ms/step
186/186              0s 2ms/step
Completed M27 in 73.51 seconds
744/744              2s 2ms/step
186/186              0s 2ms/step
Completed M28 in 79.45 seconds
744/744              2s 2ms/step
186/186              0s 2ms/step
Completed M29 in 62.14 seconds
744/744              1s 2ms/step
186/186              0s 2ms/step
Completed M30 in 83.21 seconds
744/744              2s 2ms/step
186/186              0s 2ms/step
Completed M31 in 77.67 seconds
```

[43]:

	Model	Training AUC	Validation AUC	Training RMSLE \
682	M1 NN Conv1D Optimized 2	0.741977	0.712537	0.2164
683	M2 NN Conv1D Optimized 2	0.741016	0.712113	0.2155
684	M3 NN Conv1D Optimized 2	0.735827	0.716828	0.2167
685	M4 NN Conv1D Optimized 2	0.746041	0.712080	0.2159
686	M5 NN Conv1D Optimized 2	0.747499	0.715631	0.2159
687	M6 NN Conv1D Optimized 2	0.735628	0.712862	0.2171
688	M7 NN Conv1D Optimized 2	0.739474	0.713740	0.2173
689	M8 NN Conv1D Optimized 2	0.744471	0.711743	0.2175
690	M9 NN Conv1D Optimized 2	0.744813	0.716533	0.2163
691	M10 NN Conv1D Optimized 2	0.748764	0.714428	0.2146
692	M11 NN Conv1D Optimized 2	0.749263	0.713469	0.2167
693	M12 NN Conv1D Optimized 2	0.736619	0.714565	0.2179
694	M13 NN Conv1D Optimized 2	0.725337	0.713589	0.2182
695	M14 NN Conv1D Optimized 2	0.739692	0.715245	0.2166
696	M15 NN Conv1D Optimized 2	0.724454	0.711854	0.2173
697	M16 NN Conv1D Optimized 2	0.728835	0.712735	0.2183
698	M17 NN Conv1D Optimized 2	0.727925	0.714761	0.2183
699	M18 NN Conv1D Optimized 2	0.740541	0.713188	0.2177
700	M19 NN Conv1D Optimized 2	0.725578	0.713094	0.2183
701	M20 NN Conv1D Optimized 2	0.736986	0.714197	0.2163
702	M21 NN Conv1D Optimized 2	0.740667	0.714013	0.2191
703	M22 NN Conv1D Optimized 2	0.749898	0.713328	0.2162

704	M23 NN Conv1D Optimized 2	0.745875	0.713304	0.2165
705	M24 NN Conv1D Optimized 2	0.737323	0.716395	0.2184
706	M25 NN Conv1D Optimized 2	0.740056	0.714612	0.2170
707	M26 NN Conv1D Optimized 2	0.734910	0.714037	0.2186
708	M27 NN Conv1D Optimized 2	0.741309	0.713575	0.2187
709	M28 NN Conv1D Optimized 2	0.751094	0.713018	0.2150
710	M29 NN Conv1D Optimized 2	0.728173	0.714781	0.2203
711	M30 NN Conv1D Optimized 2	0.751206	0.713651	0.2137
712	M31 NN Conv1D Optimized 2	0.745493	0.714612	0.2153

	Validation RMSLE
682	0.2236
683	0.2233
684	0.2228
685	0.2241
686	0.2238
687	0.2234
688	0.2241
689	0.2244
690	0.2235
691	0.2229
692	0.2248
693	0.2243
694	0.2233
695	0.2235
696	0.2230
697	0.2237
698	0.2236
699	0.2245
700	0.2236
701	0.2229
702	0.2254
703	0.2244
704	0.2239
705	0.2245
706	0.2238
707	0.2245
708	0.2255
709	0.2238
710	0.2255
711	0.2235
712	0.2235

0.63 #### Interpretation

This neural network has an additional convolutional layer, slightly increased dropout, increased learning rate, adjusted patient, reduced epochs, and increased batch size. This model does run faster as compared to the last model, likely due to the decreased number of epochs. However, it does not show a significant difference in performance. It is still a slow running model, and does not have an enhanced validation AUC. Thus, the second simple NN is still preferred. Nevertheless, the adjusted EBM 1 performs the best out of all of the models based on analyzing the output here. An analysis over the dataframe of scores will be conducted to better determine the best models.

0.64 # Model Selection

Three primary methods for organizing the dataframe of scores is conducted to try and best rank the models. Since there are a total of 712 models to consider, this is a very difficult process to do just by eye.

1) The first method takes the difference in AUC scores between the training and validation sets to consider overfitting. Additionally, the complexity of the variable group is considered, with M1 being the most simple model and M31 being the most complex model. It is important to take into consideration the complexity of the model in order to select the best models. Finally, the dataframe is sorted by the highest validation, lowest AUC difference, and lowest complexity.

2) The second method takes into consideration the lowest RMSLE scores and highest AUC scores. This is for comparison purposes to the other rankings, as RMSLE is being considered secondarily in this analysis with such a wide array of different types of models.

3) The third model takes into consideration both RMSLE and AUC scores, but creates a combined score, applying a greater weight to the AUC score since it is more important to take into consideration.

0.65 ## Model Validation AUC, AUC Difference and COmplexity Ranking

[70]:
```
# adding a 'Difference AUC' column to measure overfitting
results_df['Difference AUC'] = abs(results_df['Training AUC'] -␣
↪results_df['Validation AUC'])

# adding a 'Complexity' column based on the model name
results_df['Complexity'] = results_df['Model'].apply(lambda x: int(x.
↪split()[0][1:]))

# sorting by the Validation AUC (desc), then by Difference AUC (asc), then by␣
↪Complexity (asc)
sorted_results_df = results_df.sort_values(by=['Validation AUC', 'Difference␣
↪AUC', 'Complexity'], ascending=[False, True, True])

# taking a look at the top 40 models
top_40_models = sorted_results_df.head(40)
top_40_models
```

[70]:
```
                 Model  Training AUC  Validation AUC  Training RMSLE  \
514  M19 EBM Adjusted 2      0.756544        0.729734          0.2153
456             M23 EBM      0.762217        0.729273          0.2141
```

483	M19 EBM Adjusted 1	0.753422	0.728563	0.2156
513	M18 EBM Adjusted 2	0.759832	0.728308	0.2149
515	M20 EBM Adjusted 2	0.753729	0.728196	0.2156
442	M9 EBM	0.762717	0.728033	0.2141
525	M30 EBM Adjusted 2	0.747995	0.727965	0.2164
485	M21 EBM Adjusted 1	0.745826	0.727762	0.2167
484	M20 EBM Adjusted 1	0.748417	0.727667	0.2164
440	M7 EBM	0.765401	0.727386	0.2137
474	M10 EBM Adjusted 1	0.739305	0.727329	0.2174
482	M18 EBM Adjusted 1	0.755229	0.727098	0.2155
445	M12 EBM	0.760207	0.727088	0.2145
451	M18 EBM	0.783207	0.727065	0.2114
443	M10 EBM	0.766798	0.726993	0.2136
447	M14 EBM	0.761601	0.726991	0.2143
495	M31 EBM Adjusted 1	0.742422	0.726985	0.2172
453	M20 EBM	0.783635	0.726916	0.2114
444	M11 EBM	0.766312	0.726887	0.2136
448	M15 EBM	0.760804	0.726864	0.2144
494	M30 EBM Adjusted 1	0.743963	0.726811	0.2170
516	M21 EBM Adjusted 2	0.750283	0.726772	0.2161
486	M22 EBM Adjusted 1	0.741926	0.726765	0.2171
464	M31 EBM	0.781066	0.726712	0.2118
526	M31 EBM Adjusted 2	0.746505	0.726678	0.2166
452	M19 EBM	0.790839	0.726561	0.2102
475	M11 EBM Adjusted 1	0.738411	0.726542	0.2175
455	M22 EBM	0.765821	0.726432	0.2137
478	M14 EBM Adjusted 1	0.740413	0.726406	0.2173
458	M25 EBM	0.759795	0.726340	0.2145
457	M24 EBM	0.760277	0.726308	0.2144
473	M9 EBM Adjusted 1	0.738593	0.726296	0.2175
441	M8 EBM	0.766168	0.726259	0.2137
512	M17 EBM Adjusted 2	0.749817	0.726238	0.2162
481	M17 EBM Adjusted 1	0.746843	0.726236	0.2167
517	M22 EBM Adjusted 2	0.747202	0.726052	0.2164
503	M8 EBM Adjusted 2	0.746996	0.725894	0.2164
446	M13 EBM	0.760188	0.725832	0.2144
487	M23 EBM Adjusted 1	0.736410	0.725763	0.2178
472	M8 EBM Adjusted 1	0.743572	0.725698	0.2169

	Validation RMSLE	Difference AUC	Complexity	Normalized RMSLE \
514	0.2230	0.026810	19	0.539256
456	0.2230	0.032944	23	0.539256
483	0.2230	0.024858	19	0.539256
513	0.2231	0.031524	18	0.539050
515	0.2230	0.025534	20	0.539256
442	0.2232	0.034684	9	0.538843
525	0.2228	0.020030	30	0.539669

485	0.2230	0.018064	21	0.539256
484	0.2230	0.020750	20	0.539256
440	0.2232	0.038015	7	0.538843
474	0.2231	0.011977	10	0.539050
482	0.2231	0.028131	18	0.539050
445	0.2231	0.033119	12	0.539050
451	0.2233	0.056142	18	0.538636
443	0.2231	0.039804	10	0.539050
447	0.2232	0.034610	14	0.538843
495	0.2230	0.015437	31	0.539256
453	0.2233	0.056718	20	0.538636
444	0.2232	0.039425	11	0.538843
448	0.2231	0.033939	15	0.539050
494	0.2230	0.017152	30	0.539256
516	0.2229	0.023511	21	0.539463
486	0.2232	0.015160	22	0.538843
464	0.2233	0.054354	31	0.538636
526	0.2230	0.019827	31	0.539256
452	0.2234	0.064278	19	0.538430
475	0.2231	0.011869	11	0.539050
455	0.2234	0.039389	22	0.538430
478	0.2232	0.014007	14	0.538843
458	0.2231	0.033455	25	0.539050
457	0.2233	0.033969	24	0.538636
473	0.2232	0.012297	9	0.538843
441	0.2233	0.039908	8	0.538636
512	0.2232	0.023579	17	0.538843
481	0.2232	0.020607	17	0.538843
517	0.2233	0.021150	22	0.538636
503	0.2233	0.021101	8	0.538636
446	0.2232	0.034355	13	0.538843
487	0.2232	0.010646	23	0.538843
472	0.2233	0.017874	8	0.538636

	Combined Score
514	0.672591
456	0.672268
483	0.671771
513	0.671530
515	0.671514
442	0.671276
525	0.671476
485	0.671210
484	0.671144
440	0.670823
474	0.670845
482	0.670683

```
445         0.670677
451         0.670536
443         0.670610
447         0.670547
495         0.670666
453         0.670432
444         0.670474
448         0.670520
494         0.670545
516         0.670579
486         0.670389
464         0.670289
526         0.670452
452         0.670122
475         0.670294
455         0.670031
478         0.670137
458         0.670153
457         0.670006
473         0.670060
441         0.669972
512         0.670019
481         0.670018
517         0.669827
503         0.669717
446         0.669736
487         0.669687
472         0.669579
```

0.66 ## Model RMSLE and AUC Ranking

[45]:
```
# sorting models by Validation RMSLE (ascending), then by Validation AUC␣
↪(descending) for a focus on prediction accuracy
sorted_by_rmsle_df = results_df.sort_values(by=['Validation RMSLE', 'Validation␣
↪AUC'], ascending=[True, False])

# taking a look at the top 40 models focused on RMSLE
top_40_models_rmsle = sorted_by_rmsle_df.head(40)
print("Top 40 Models Sorted by RMSLE:")
top_40_models_rmsle
```

```
Top 20 Models Sorted by RMSLE:
```

[45]:

	Model	Training AUC	Validation AUC	Training RMSLE \
525	M30 EBM Adjusted 2	0.747995	0.727965	0.2164
684	M3 NN Conv1D Optimized 2	0.735827	0.716828	0.2167
516	M21 EBM Adjusted 2	0.750283	0.726772	0.2161

```
691  M10 NN Conv1D Optimized 2       0.748764         0.714428             0.2146
701  M20 NN Conv1D Optimized 2       0.736986         0.714197             0.2163
514         M19 EBM Adjusted 2       0.756544         0.729734             0.2153
456                    M23 EBM       0.762217         0.729273             0.2141
483         M19 EBM Adjusted 1       0.753422         0.728563             0.2156
515         M20 EBM Adjusted 2       0.753729         0.728196             0.2156
485         M21 EBM Adjusted 1       0.745826         0.727762             0.2167
484         M20 EBM Adjusted 1       0.748417         0.727667             0.2164
495         M31 EBM Adjusted 1       0.742422         0.726985             0.2172
494         M30 EBM Adjusted 1       0.743963         0.726811             0.2170
526         M31 EBM Adjusted 2       0.746505         0.726678             0.2166
629             M31 NN Complex       0.744133         0.720031             0.2166
648             M31 NN Complex       0.748208         0.718108             0.2166
622             M31 NN Complex       0.743683         0.717296             0.2166
640             M31 NN Complex       0.739494         0.717039             0.2166
634             M31 NN Complex       0.743502         0.716064             0.2166
646             M31 NN Complex       0.744693         0.715496             0.2166

     Validation RMSLE  Difference AUC  Complexity
525            0.2228        0.020030          30
684            0.2228        0.018998           3
516            0.2229        0.023511          21
691            0.2229        0.034336          10
701            0.2229        0.022789          20
514            0.2230        0.026810          19
456            0.2230        0.032944          23
483            0.2230        0.024858          19
515            0.2230        0.025534          20
485            0.2230        0.018064          21
484            0.2230        0.020750          20
495            0.2230        0.015437          31
494            0.2230        0.017152          30
526            0.2230        0.019827          31
629            0.2230        0.024102          31
648            0.2230        0.030100          31
622            0.2230        0.026387          31
640            0.2230        0.022455          31
634            0.2230        0.027439          31
646            0.2230        0.029197          31
```

0.67 ## Model RMSLE and AUC Combined Score Ranking

[71]:
```
# normalizing the RMSLE
max_rmsle = results_df['Validation RMSLE'].max()
results_df['Normalized RMSLE'] = 1 - (results_df['Validation RMSLE'] /
 ↪max_rmsle)
```

```python
# simple combined score (example: 70% weight on AUC, 30% weight on Normalized␣
↪RMSLE)
results_df['Combined Score'] = 0.7 * results_df['Validation AUC'] + 0.3 *␣
↪results_df['Normalized RMSLE']

# sorting by combined score (descending)
sorted_by_combined_score_df = results_df.sort_values(by='Combined Score',␣
↪ascending=False)

# taking a look at the top 40 models based on the combined score
top_40_models_combined = sorted_by_combined_score_df.head(40)
print("Top 40 Models Sorted by Combined Score (AUC & RMSLE):")
top_40_models_combined
```

Top 20 Models Sorted by Combined Score (AUC & RMSLE):

[71]:

	Model	Training AUC	Validation AUC	Training RMSLE	\
514	M19 EBM Adjusted 2	0.756544	0.729734	0.2153	
456	M23 EBM	0.762217	0.729273	0.2141	
483	M19 EBM Adjusted 1	0.753422	0.728563	0.2156	
513	M18 EBM Adjusted 2	0.759832	0.728308	0.2149	
515	M20 EBM Adjusted 2	0.753729	0.728196	0.2156	
525	M30 EBM Adjusted 2	0.747995	0.727965	0.2164	
442	M9 EBM	0.762717	0.728033	0.2141	
485	M21 EBM Adjusted 1	0.745826	0.727762	0.2167	
484	M20 EBM Adjusted 1	0.748417	0.727667	0.2164	
474	M10 EBM Adjusted 1	0.739305	0.727329	0.2174	
440	M7 EBM	0.765401	0.727386	0.2137	
482	M18 EBM Adjusted 1	0.755229	0.727098	0.2155	
445	M12 EBM	0.760207	0.727088	0.2145	
495	M31 EBM Adjusted 1	0.742422	0.726985	0.2172	
443	M10 EBM	0.766798	0.726993	0.2136	
516	M21 EBM Adjusted 2	0.750283	0.726772	0.2161	
447	M14 EBM	0.761601	0.726991	0.2143	
494	M30 EBM Adjusted 1	0.743963	0.726811	0.2170	
451	M18 EBM	0.783207	0.727065	0.2114	
448	M15 EBM	0.760804	0.726864	0.2144	
444	M11 EBM	0.766312	0.726887	0.2136	
526	M31 EBM Adjusted 2	0.746505	0.726678	0.2166	
453	M20 EBM	0.783635	0.726916	0.2114	
486	M22 EBM Adjusted 1	0.741926	0.726765	0.2171	
475	M11 EBM Adjusted 1	0.738411	0.726542	0.2175	
464	M31 EBM	0.781066	0.726712	0.2118	
458	M25 EBM	0.759795	0.726340	0.2145	
478	M14 EBM Adjusted 1	0.740413	0.726406	0.2173	
452	M19 EBM	0.790839	0.726561	0.2102	
473	M9 EBM Adjusted 1	0.738593	0.726296	0.2175	

455	M22 EBM	0.765821	0.726432	0.2137
512	M17 EBM Adjusted 2	0.749817	0.726238	0.2162
481	M17 EBM Adjusted 1	0.746843	0.726236	0.2167
457	M24 EBM	0.760277	0.726308	0.2144
441	M8 EBM	0.766168	0.726259	0.2137
517	M22 EBM Adjusted 2	0.747202	0.726052	0.2164
446	M13 EBM	0.760188	0.725832	0.2144
503	M8 EBM Adjusted 2	0.746996	0.725894	0.2164
487	M23 EBM Adjusted 1	0.736410	0.725763	0.2178
472	M8 EBM Adjusted 1	0.743572	0.725698	0.2169

	Validation RMSLE	Difference AUC	Complexity	Normalized RMSLE \
514	0.2230	0.026810	19	0.539256
456	0.2230	0.032944	23	0.539256
483	0.2230	0.024858	19	0.539256
513	0.2231	0.031524	18	0.539050
515	0.2230	0.025534	20	0.539256
525	0.2228	0.020030	30	0.539669
442	0.2232	0.034684	9	0.538843
485	0.2230	0.018064	21	0.539256
484	0.2230	0.020750	20	0.539256
474	0.2231	0.011977	10	0.539050
440	0.2232	0.038015	7	0.538843
482	0.2231	0.028131	18	0.539050
445	0.2231	0.033119	12	0.539050
495	0.2230	0.015437	31	0.539256
443	0.2231	0.039804	10	0.539050
516	0.2229	0.023511	21	0.539463
447	0.2232	0.034610	14	0.538843
494	0.2230	0.017152	30	0.539256
451	0.2233	0.056142	18	0.538636
448	0.2231	0.033939	15	0.539050
444	0.2232	0.039425	11	0.538843
526	0.2230	0.019827	31	0.539256
453	0.2233	0.056718	20	0.538636
486	0.2232	0.015160	22	0.538843
475	0.2231	0.011869	11	0.539050
464	0.2233	0.054354	31	0.538636
458	0.2231	0.033455	25	0.539050
478	0.2232	0.014007	14	0.538843
452	0.2234	0.064278	19	0.538430
473	0.2232	0.012297	9	0.538843
455	0.2234	0.039389	22	0.538430
512	0.2232	0.023579	17	0.538843
481	0.2232	0.020607	17	0.538843
457	0.2233	0.033969	24	0.538636
441	0.2233	0.039908	8	0.538636

517	0.2233	0.021150	22	0.538636
446	0.2232	0.034355	13	0.538843
503	0.2233	0.021101	8	0.538636
487	0.2232	0.010646	23	0.538843
472	0.2233	0.017874	8	0.538636

	Combined Score
514	0.672591
456	0.672268
483	0.671771
513	0.671530
515	0.671514
525	0.671476
442	0.671276
485	0.671210
484	0.671144
474	0.670845
440	0.670823
482	0.670683
445	0.670677
495	0.670666
443	0.670610
516	0.670579
447	0.670547
494	0.670545
451	0.670536
448	0.670520
444	0.670474
526	0.670452
453	0.670432
486	0.670389
475	0.670294
464	0.670289
458	0.670153
478	0.670137
452	0.670122
473	0.670060
455	0.670031
512	0.670019
481	0.670018
457	0.670006
441	0.669972
517	0.669827
446	0.669736
503	0.669717
487	0.669687
472	0.669579

0.68 ## Model Considerations

As the EBM Adjusted and EBM models perform the best in both the first and third rankings, these will be considered. The RMSLE normal ranking shows different models as compared to the others rankings that put the emphasis on the AUC score. M19 EBM Adjusted 2, M23 EBM, and M19 EBM Adjusted 1 perform the best out of these rankings.

0.69 # Test Set Prediction

[47]:
```
def prediction_folder(day):
    folder_path = f'Predictions/Day_{day}'
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)
```

0.70 ### Prediction Functions

Since some models are repeated frequently, it will clean up the code to utilize functions.

0.71 #### Simple EBM Prediction Function

[48]:
```
def simple_ebm_prediction(model, day):
    features = models[model]

    # Simple EBM Model
    ebm = ExplainableBoostingClassifier(random_state=20240325)
    ebm.fit(X_train[features], y_train)

    X_test = test_data[features]

    # Predicting with the model
    test_data['score'] = ebm.predict_proba(X_test)[:, 1]

    # Saving the required predictions
    test_data[['article_id', 'score']].to_csv(f'Predictions/Day_{day}/
 ↪{model}_ebm_predictions.csv', index=False)
```

0.72 #### Adjusted EBM 1 Prediction Function

[49]:
```
def ebm_adjusted_1_prediction(model, day):
    features = models[model]

    # Adjusted EBM Model 1
    ebm_adjusted_1 = ExplainableBoostingClassifier(
        random_state=20240325,
        learning_rate=0.01,
        max_bins=256,
        interactions=10,
```

```
        early_stopping_rounds=50
    )
    ebm_adjusted_1.fit(X_train[features], y_train)

    X_test = test_data[features]

    # Predicting with the model
    test_data['score'] = ebm_adjusted_1.predict_proba(X_test)[:, 1]

    # Saving the required predictions
    test_data[['article_id', 'score']].to_csv(f'Predictions/Day_{day}/
↪{model}_ebm_adjusted_1_predictions.csv', index=False)
```

0.73 #### Adjusted EBM 2 Prediction Function

[50]:
```
def ebm_adjusted_2_prediction(model, day):
    features = models[model]

    # Adjusted EBM Model 2
    ebm_adjusted_2 = ExplainableBoostingClassifier(
        random_state=20240325,
        learning_rate=0.005,
        max_bins=512,
        interactions=15,
        early_stopping_rounds=100,
        n_jobs=-1
    )
    ebm_adjusted_2.fit(X_train[features], y_train)

    X_test = test_data[features]

    # Predicting with the model
    test_data['score'] = ebm_adjusted_2.predict_proba(X_test)[:, 1]

    # Saving the required predictions
    test_data[['article_id', 'score']].to_csv(f'Predictions/Day_{day}/
↪{model}_ebm_adjusted_2_predictions.csv', index=False)
```

0.74 ## Day 1 Predictions

All of the predictions from day one came from the simple EBM model

[51]:
```
prediction_folder('1')
```

[52]:
```
simple_ebm_prediction('M9', '1')
```

[53]:
```
simple_ebm_prediction('M7', '1')
```

[54]:
```
simple_ebm_prediction('M10', '1')
```

[55]:
```
simple_ebm_prediction('M6', '1')
```

[56]:
```
simple_ebm_prediction('M12', '1')
```

0.75 ## Day 2 Predictions

[57]:
```
prediction_folder('2')
```

[58]:
```
simple_ebm_prediction('M11', '2')
```

[59]:
```
ebm_adjusted_1_prediction('M10', '2')
```

[60]:
```
ebm_adjusted_1_prediction('M11', '2')
```

[61]:
```
ebm_adjusted_1_prediction('M12', '2')
```

[62]:
```
ebm_adjusted_1_prediction('M9', '2')
```

0.76 ## Day 3 Predictions

[63]:
```
prediction_folder('3')
```

[64]:
```
ebm_adjusted_1_prediction('M19', '3')
```

[65]:
```
ebm_adjusted_2_prediction('M18', '3')
```

[66]:
```
ebm_adjusted_2_prediction('M19', '3')
```

[67]:
```
ebm_adjusted_2_prediction('M20', '3')
```

[68]:
```
simple_ebm_prediction('M18', '3')
```

0.77 ## Day 4 Predictions

[69]:
```
prediction_folder('4')
```

[72]:
```
simple_ebm_prediction('M23', '4')
```

[73]:
```
simple_ebm_prediction('M19', '4')
```

[74]:
```
simple_ebm_prediction('M31', '4')
```

[75]:
```
simple_ebm_prediction('M24', '4')
```

[76]:
```
simple_ebm_prediction('M13', '4')
```

0.78 ## Day 5 Predictions

[77]:
```
prediction_folder('5')
```

[78]:
```
simple_ebm_prediction('M22', '5')
```

[79]:
```
ebm_adjusted_2_prediction('M30', '5')
```

[80]:
```
ebm_adjusted_1_prediction('M18', '5')
```

[81]:
```
ebm_adjusted_1_prediction('M31', '5')
```

[82]:
```
ebm_adjusted_2_prediction('M21', '5')
```

0.79 ## Day 6 Predictions

[83]:
```
prediction_folder('6')
```

[84]:
```
simple_ebm_prediction('M14', '6')
```

[85]:
```
simple_ebm_prediction('M15', '6')
```

[86]:
```
simple_ebm_prediction('M20', '6')
```

[87]:
```
simple_ebm_prediction('M25', '6')
```

[88]:
```
ebm_adjusted_1_prediction('M30', '6')
```

0.80 ## Day 7 Predictions

[89]:
```
prediction_folder('7')
```

[90]:
```
ebm_adjusted_2_prediction('M31', '7')
```

[91]:
```
ebm_adjusted_1_prediction('M22', '7')
```

[92]:
```
ebm_adjusted_1_prediction('M14', '7')
```

[ ]:
```

```