kaggle-assignment-zsofia-katona

April 16, 2024

#

Classifying social media post popularity

Created by Zsófia Rebeka Katona

Data Science 2 - Kaggle competition

0.1 ## Introduction

The goal of this challenge is to predict which articles of the popularity dataset are labelled as popular. To achieve this, I will build and experiment with different predictive models from simple linear models to convolutional neural networks. The aim is to find the model that performs the best in predicting the popularity of articles. The data comes from the website mashable.com as of the beginning of 2015. The dataset used in the competition can be found in the UCI repository.

0.2 ## Data import

[1]:
```
# Importing required libraries
import datetime
import os
import time

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import sklearn
import tensorflow as tf
import xgboost as xgb
import warnings


from keras.callbacks import EarlyStopping, ReduceLROnPlateau
from keras.layers import BatchNormalization, Conv1D, Dense, Dropout, Flatten,␣
  ↪GlobalAveragePooling2D, Input, MaxPooling1D, Rescaling, Reshape
from keras.models import Model, Sequential, clone_model
from keras.optimizers import Adam
from keras.utils import set_random_seed, to_categorical
from sklearn import tree
```

```
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.impute import KNNImputer, SimpleImputer
from sklearn.linear_model import LassoCV, LinearRegression, LogisticRegression,␣
    ↪Ridge, ElasticNet
from sklearn.metrics import PrecisionRecallDisplay, RocCurveDisplay, auc,␣
    ↪confusion_matrix, precision_recall_curve, precision_score, recall_score,␣
    ↪roc_auc_score, roc_curve
from sklearn.model_selection import GridSearchCV, KFold, RandomizedSearchCV,␣
    ↪cross_validate, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer, MinMaxScaler,␣
    ↪OneHotEncoder, PolynomialFeatures, StandardScaler
from sklearn.tree import DecisionTreeRegressor
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from xgboost import XGBRegressor
```

[2]:

```
# Importing the training and the test set
current_dir = os.getcwd()
train_df = pd.read_csv("train.csv")
test_df = pd.read_csv("test.csv")

# Checking the attributes of the sets
print(f"The shape of the train set is: {train_df.shape}.")
print(f"The shape of the test set is {test_df.shape}.")
print("The data types of the train set:")
train_df.info()
```

```
The shape of the train set is: (29733, 61).
The shape of the test set is (9911, 60).
The data types of the train set:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 29733 entries, 0 to 29732
Data columns (total 61 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   timedelta                      29733 non-null  int64
 1   n_tokens_title                 29733 non-null  int64
 2   n_tokens_content               29733 non-null  int64
 3   n_unique_tokens                29733 non-null  float64
 4   n_non_stop_words               29733 non-null  float64
 5   n_non_stop_unique_tokens       29733 non-null  float64
 6   num_hrefs                      29733 non-null  int64
 7   num_self_hrefs                 29733 non-null  int64
 8   num_imgs                       29733 non-null  int64
 9   num_videos                     29733 non-null  int64
```

```
10  average_token_length          29733 non-null  float64
11  num_keywords                  29733 non-null  int64
12  data_channel_is_lifestyle     29733 non-null  int64
13  data_channel_is_entertainment 29733 non-null  int64
14  data_channel_is_bus           29733 non-null  int64
15  data_channel_is_socmed        29733 non-null  int64
16  data_channel_is_tech          29733 non-null  int64
17  data_channel_is_world         29733 non-null  int64
18  kw_min_min                    29733 non-null  int64
19  kw_max_min                    29733 non-null  float64
20  kw_avg_min                    29733 non-null  float64
21  kw_min_max                    29733 non-null  int64
22  kw_max_max                    29733 non-null  int64
23  kw_avg_max                    29733 non-null  float64
24  kw_min_avg                    29733 non-null  float64
25  kw_max_avg                    29733 non-null  float64
26  kw_avg_avg                    29733 non-null  float64
27  self_reference_min_shares     29733 non-null  float64
28  self_reference_max_shares     29733 non-null  float64
29  self_reference_avg_sharess    29733 non-null  float64
30  weekday_is_monday             29733 non-null  int64
31  weekday_is_tuesday            29733 non-null  int64
32  weekday_is_wednesday          29733 non-null  int64
33  weekday_is_thursday           29733 non-null  int64
34  weekday_is_friday             29733 non-null  int64
35  weekday_is_saturday           29733 non-null  int64
36  weekday_is_sunday             29733 non-null  int64
37  is_weekend                    29733 non-null  int64
38  LDA_00                        29733 non-null  float64
39  LDA_01                        29733 non-null  float64
40  LDA_02                        29733 non-null  float64
41  LDA_03                        29733 non-null  float64
42  LDA_04                        29733 non-null  float64
43  global_subjectivity           29733 non-null  float64
44  global_sentiment_polarity     29733 non-null  float64
45  global_rate_positive_words    29733 non-null  float64
46  global_rate_negative_words    29733 non-null  float64
47  rate_positive_words           29733 non-null  float64
48  rate_negative_words           29733 non-null  float64
49  avg_positive_polarity         29733 non-null  float64
50  min_positive_polarity         29733 non-null  float64
51  max_positive_polarity         29733 non-null  float64
52  avg_negative_polarity         29733 non-null  float64
53  min_negative_polarity         29733 non-null  float64
54  max_negative_polarity         29733 non-null  float64
55  title_subjectivity            29733 non-null  float64
56  title_sentiment_polarity      29733 non-null  float64
57  abs_title_subjectivity        29733 non-null  float64
```

```
 58  abs_title_sentiment_polarity   29733 non-null  float64
 59  is_popular                     29733 non-null  int64
 60  article_id                     29733 non-null  int64
dtypes: float64(34), int64(27)
memory usage: 13.8 MB
```

0.3 ## Exploratory Data Analysis

[3]:
```
# Checking the dataset
train_df.head(10)
```

[3]:
```
   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
5        568               7               126         0.723577
6        318              12              1422         0.367994
7        582               6              1102         0.451287
8        269               9                 0         0.000000
9        567               7                94         0.755319

   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
5               1.0                  0.774194          3               3
6               1.0                  0.469256         28              28
7               1.0                  0.642089          7               3
8               0.0                  0.000000          0               0
9               1.0                  0.812500          8               6

   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
5         1           0  …               0.285714               0.000000
6        26           0  …               0.700000              -0.234167
7         1           0  …               0.800000              -0.151630
8         5           0  …               0.000000               0.000000
9         0          11  …               1.000000              -0.183333
```

```
   min_negative_polarity  max_negative_polarity  title_subjectivity  \
0                   -0.4              -0.100000            0.000000
1                   -1.0              -0.100000            0.000000
2                   -0.2              -0.050000            0.000000
3                   -0.8              -0.400000            0.000000
4                   -1.0              -0.050000            0.000000
5                    0.0               0.000000            0.454545
6                   -0.5              -0.050000            1.000000
7                   -0.4              -0.050000            0.800000
8                    0.0               0.000000            0.500000
9                   -0.2              -0.166667            0.000000

   title_sentiment_polarity  abs_title_subjectivity  \
0                  0.000000                0.500000
1                  0.000000                0.500000
2                  0.000000                0.500000
3                  0.000000                0.500000
4                  0.000000                0.500000
5                  0.136364                0.045455
6                  0.100000                0.500000
7                  0.400000                0.300000
8                  0.500000                0.000000
9                  0.000000                0.500000

   abs_title_sentiment_polarity  is_popular  article_id
0                      0.000000           0           1
1                      0.000000           0           3
2                      0.000000           0           5
3                      0.000000           0           6
4                      0.000000           0           7
5                      0.136364           0           8
6                      0.100000           0           9
7                      0.400000           1          11
8                      0.500000           0          12
9                      0.000000           0          14

[10 rows x 61 columns]
```

0.4 ### Data cleaning

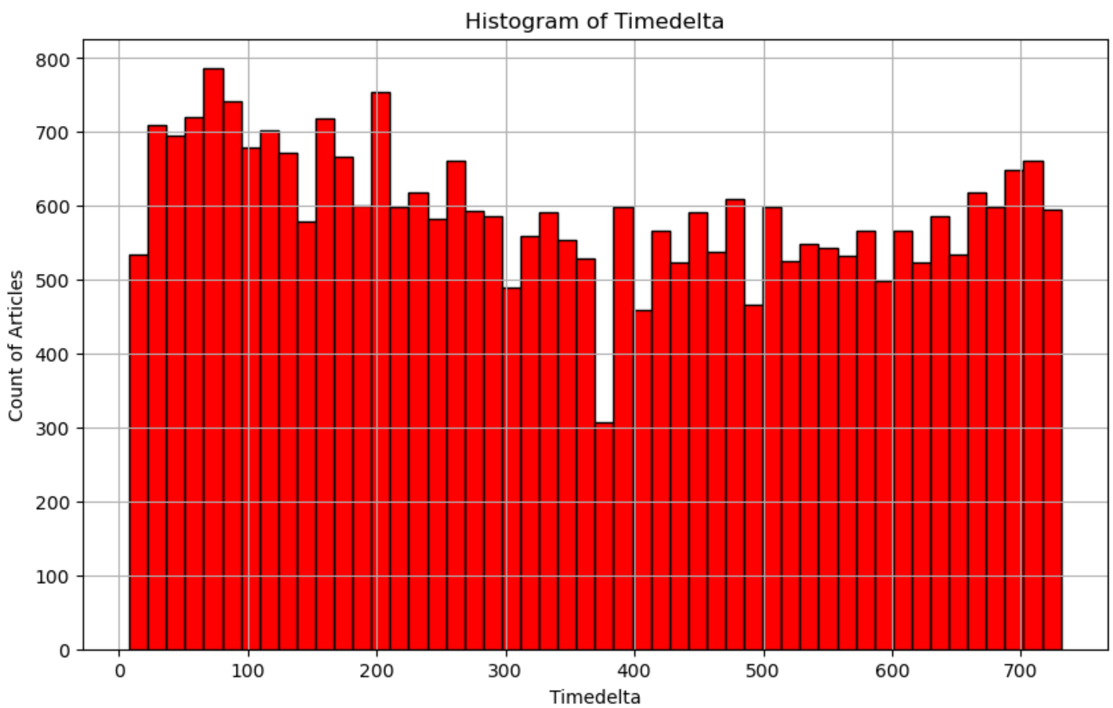
[4]:
```
# Filtering for missing values
train_df.isnull().sum().sum()
```

[4]: 0

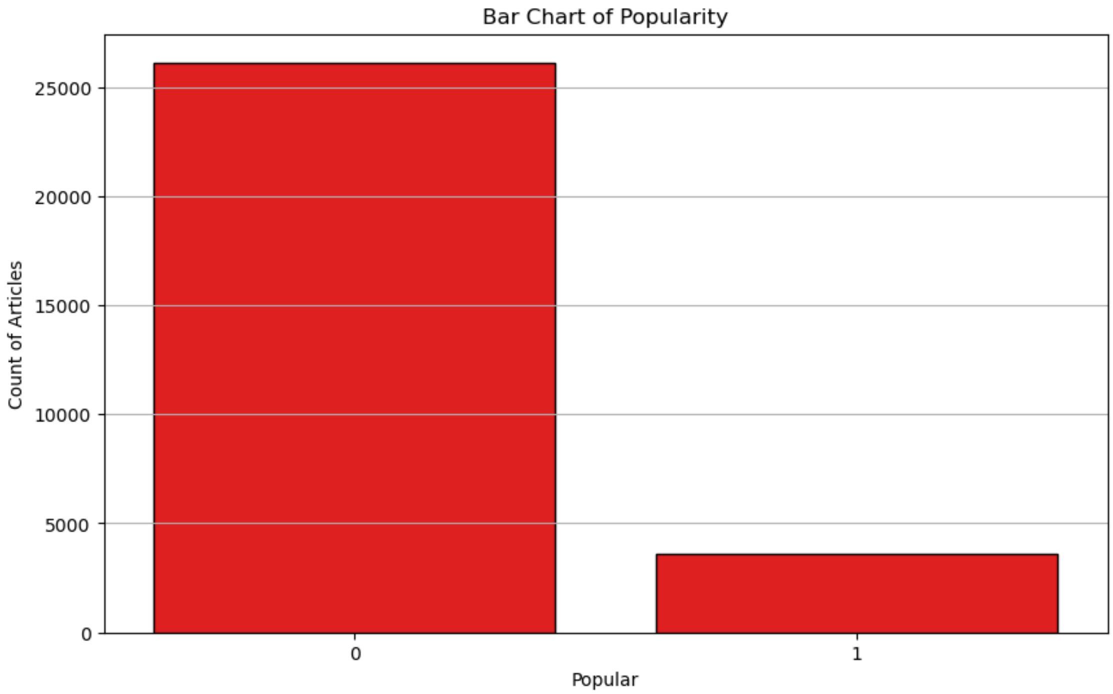
[5]:
```
# Filtering for NA values
train_df.isna().sum().sum()
```

[5]: 0

[6]:
```
# Plotting the distribution of days elapsed between the article publication and
↪the dataset acquisition
plt.figure(figsize=(10, 6))
plt.hist(train_df['timedelta'], bins=50, color='red', edgecolor='black')
plt.xlabel('Timedelta')
plt.ylabel('Count of Articles')
plt.title('Histogram of Timedelta')
plt.grid(True)
plt.show()
```



[7]:
```
# Plotting the distribution of popularity
plt.figure(figsize=(10, 6))
sns.countplot(x='is_popular', data=train_df, color='red', edgecolor='black')
plt.xlabel('Popular')
plt.ylabel('Count of Articles')
plt.title('Bar Chart of Popularity')
plt.grid(True, axis='y')
plt.show()
```



[8]:
```
# Checking the number of popular and not popular articles
train_df.value_counts("is_popular")
```

[8]:
```
is_popular
0    26116
1     3617
dtype: int64
```

We can observe that there are significantly more unpopular articles than popular ones. Namely, 12% of the total articles are popular.

Train-test split

[9]:
```
# Dropping the target variable from the training set
features = train_df.drop(columns=["is_popular"])
label = train_df["is_popular"]

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

# Splitting the fata
X_train, X_test, y_train, y_test = train_test_split(features, label,␣
↪test_size=0.2, random_state=prng)
```

Feature engineering

```
[11]: # Adding the total number of media elements in each post (links, videos, images)
train_df['total_multimedia'] = train_df['num_hrefs'] +␣
 ↪train_df['num_self_hrefs'] + train_df['num_imgs'] + train_df['num_videos']

# Combining 'easy-to-consume' topics to represent the dominant topic of the␣
 ↪article
train_df['easy_to_consume'] = train_df['data_channel_is_socmed'] +␣
 ↪train_df['data_channel_is_entertainment'] +␣
 ↪train_df['data_channel_is_lifestyle']

# Combining more specific topics
train_df['hard_to_consume'] = train_df['data_channel_is_bus'] +␣
 ↪train_df['data_channel_is_tech'] + train_df['data_channel_is_world']

# Selecting features for feature engineering
features_fe = train_df[['total_multimedia', 'easy_to_consume',␣
 ↪'hard_to_consume']]

# Labelling the count column
label = train_df["is_popular"]

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

# Splitting the feature engineered training set and test set again
X_train_fe, X_test_fe, y_train, y_test = train_test_split(features_fe, label,␣
 ↪test_size=0.2, random_state=prng)
```

0.5 ## Predictive models

```
[12]: # Defining the loss function (root mean squared logarithmic error)
def calculateRMSLE(prediction, y_obs):
    """
    Calculate the Root Mean Squared Logarithmic Error (RMSLE)
    between the predicted values and the observed values.

    Parameters:
    - prediction: array-like, the predicted values.
    - y_obs: array-like, the observed (true) values.

    Returns:
    - float: the RMSLE value rounded to four decimal places.
    """
    return round(np.sqrt(
        np.mean(
                (
```

```
                    np.log(np.where(prediction < 0, 0, prediction) + 1) -
                    np.log(y_obs + 1)
                )**2
        )
    ), 4)
```

0.5.1 Model 1: Linear models

[13]:
```
# Setting the pipeline for the linear model
model1 = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('num', StandardScaler(), X_train.columns)
    ])),
    ('regressor', LinearRegression())
])

# Fitting the model
model1.fit(X_train, y_train)

# Calculating the train error
train_error_lin = calculateRMSLE(model1.predict(X_train), y_train)

# Calculating the test error
test_error_lin = calculateRMSLE(model1.predict(X_test), y_test)

# Displaying the results
model_results = pd.DataFrame({"Model": ["Model 1 (Linear)"], "Train Error":␣
 ↪[train_error_lin], "Test Error": [test_error_lin]})
model_results
```

[13]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237

To start out, I ensembled a simple linear model consisting of a ColumnTransformer and a StandardScaler. This also serves a benchmark to compare how other models perform and a starting point to understand the dataset and set initial expectations. While the error scores are very close to each other, the test error being slightly higher implies overfitting.

0.5.2 Model 2: Logistic model

[14]:
```
# Defining the pipeline for the logistic regression model
model2 = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('num', StandardScaler(), X_train.columns)
    ])),
    ('classifier', LogisticRegression(random_state=prng))
])
```

```
# Fitting the model
model2.fit(X_train, y_train)

# Calculating the train error
train_error_model2 = calculateRMSLE(model2.predict(X_train), y_train)
# Calculating the test error
test_error_model2 = calculateRMSLE(model2.predict(X_test), y_test)

# Displaying the results
new_results_model2 = pd.DataFrame({'Model': ['Model 2 (Logistic)'],
                            'Train Error': [train_error_model2],
                            'Test Error': [test_error_model2]})
model_results = pd.concat([model_results, new_results_model2],␣
 ↪ignore_index=True)
model_results
```

[14]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435

The logistic model performed somewhat worse than the simple linear model. The model still seems to be overfitting, however, the pipeline still consists of a ColumnTransformer and a StandardScaler to prevent overfitting. This implies that maybe the model needs some regularization. (The regularized models can be found towards the end of the analysis.)

0.5.3 Model 3: Flexible linear with polynomial features

[15]:

```
# Creating a the flexible linar pipeline with 2 degree polynomials
steps = [
    ("scaling", StandardScaler()),
    ("2_degree_poly", PolynomialFeatures(degree=2, include_bias=False)),
    ("ols", LinearRegression())
]
model3 = Pipeline(steps)

# Fitting the flexible pipe on the training data
model3.fit(X_train, y_train)

# Calculating the train error
train_error_model3 = calculateRMSLE(model3.predict(X_train), y_train)

# Calculating the test error
test_error_model3 = calculateRMSLE(model3.predict(X_test), y_test)

# Displaying the results
new_results_model3 = pd.DataFrame({'Model': ['Model 3 (Polynomial linear)'],
```

```
                              'Train Error': [train_error_model3],
                              'Test Error': [test_error_model3]})
model_results = pd.concat([model_results, new_results_model3],␣
 ↪ignore_index=True)
model_results
```

[15]:

```
                      Model  Train Error  Test Error
0            Model 1 (Linear)       0.2222      0.2237
1          Model 2 (Logistic)       0.2419      0.2435
2  Model 3 (Polynomial linear)       0.2097      0.5708
```

By implementing a flexible model, I wanted to further improve accuracy. While there was a potential for overfitting with a flexible model, it was less prone to underfitting than simpler models, ensuring to capture underlying patterns. As expected, the model did show signs of overfitting. The test error was considerably higher than the training error. Due to computational intensity limitations, I could only compute the 2-degree poly model.

0.5.4 Model 4: Decision Trees

[16]:

```
# Creating the decision tree pipeline
steps = [
    ("tree", tree.DecisionTreeRegressor(max_depth=5, random_state=prng))
]
model4 = Pipeline(steps)

# Fitting the training data
model4.fit(X_train, y_train)

# Calculating the train error
train_error_model4 = calculateRMSLE(model4.predict(X_train), y_train)

# Calculating the test error
test_error_model4 = calculateRMSLE(model4.predict(X_test), y_test)

# Displaying the results
new_results_model4 = pd.DataFrame({'Model': ['Model 4 (Decision tree)'],
                              'Train Error': [train_error_model4],
                              'Test Error': [test_error_model4]})
model_results = pd.concat([model_results, new_results_model4],␣
 ↪ignore_index=True)
model_results
```

[16]:

```
                      Model  Train Error  Test Error
0            Model 1 (Linear)       0.2222      0.2237
1          Model 2 (Logistic)       0.2419      0.2435
2  Model 3 (Polynomial linear)       0.2097      0.5708
3     Model 4 (Decision tree)       0.2205      0.2265
```

My first ensemble model performed better than all of the simple linear models and to avoid overfitting and a better generalization, I set the maximum depth to a moderate number. On the other hand, it was still overfitted to the training data. The test error for the simple linear model was lower, implying that the simple linear model would potentially perform better on an unseen dataset, than the decision tree.

0.5.5 Model 5: Improved decision tree

[17]:
```
# Creating the improved decision tree pipeline
steps = [
    ("tree", tree.DecisionTreeRegressor(max_depth=10, random_state=prng))
]
model5 = Pipeline(steps)

# Fitting the training data
model5.fit(X_train, y_train)

# Calculating the train error
train_error_model5 = calculateRMSLE(model5.predict(X_train), y_train)

# Calculating the test error
test_error_model5 = calculateRMSLE(model5.predict(X_test), y_test)

# Displaying the results
new_results_model5 = pd.DataFrame({'Model': ['Model 5 (Improved Decision␣
 ↪tree)'],
                            'Train Error': [train_error_model5],
                            'Test Error': [test_error_model5]})
model_results = pd.concat([model_results, new_results_model5],␣
 ↪ignore_index=True)
model_results
```

[17]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489

To improve accuracy, I experimented with a deeper decision tree by setting the maximum depth to 10. As expected, the model became overfitted. The training error improved, but the test error increased compared to the simpler decision tree model.

0.5.6 Model 6: RandomForest

[18]:
```
# Creating a pipeline for RandomForest
steps = [
    ("rf", RandomForestRegressor(n_estimators=100, random_state=prng))
]
model6 = Pipeline(steps)

# Fitting the training data
model6.fit(X_train, y_train)

# Calculating the train error
train_error_model6 = calculateRMSLE(model6.predict(X_train), y_train)

# Calculating the test error
test_error_model6 = calculateRMSLE(model6.predict(X_test), y_test)

# Displaying the results
new_results_model6 = pd.DataFrame({'Model': ['Model 6 (Random Forest)'],
                                   'Train Error': [train_error_model6],
                                   'Test Error': [test_error_model6]})
model_results = pd.concat([model_results, new_results_model6],␣
 ↪ignore_index=True)
model_results
```

[18]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298

I set the RandomForest estimator to 100 to make the model more robust and to compute a more reliable prediction, but still wanted to achieve a balance by avoiding overfitting. The training error considerably decreased compared to the previous models, and the test error overperformed almost all other models, it still overfitted to the training data.

0.5.7 Model 7: Improved RandomForest

using cross-validation

[19]:
```
# Setting a timer to measure the running time
start_time = time.time()

# Define the number of folds
k_folds = 5
```

```
# Create a KFold cross-validation splitter
kf = KFold(n_splits=k_folds, shuffle=True, random_state=prng)

# Define preprocessing steps
preprocessor = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# Defiing the RandomForest pipeline with 100 estimators
model7 = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', RandomForestRegressor(n_estimators=100, random_state=prng))
])

# Fitting the model
model7.fit(X_train, y_train)

# Defining a custom scorer
def rmsle_scorer(estimator, X, y):
    prediction = estimator.predict(X)
    return calculateRMSLE(prediction, y)

# Performing cross-validation
cv_scores = cross_validate(model7, X_train, y_train, cv=kf,␣
 ↪scoring=rmsle_scorer)

# Calculating the error on training set
train_error_model7 = calculateRMSLE(model7.predict(X_train), y_train)

# Calculating the error on test set
test_error_model7 = calculateRMSLE(model7.predict(X_test), y_test)

# Displaying the results and the run time
new_results_model7 = pd.DataFrame({'Model': ['Model 7 (Improved Random␣
 ↪Forest)'],
                            'Train Error': [train_error_model7],
                            'Test Error': [test_error_model7]})
model_results = pd.concat([model_results, new_results_model7],␣
 ↪ignore_index=True)

end_time = time.time()
execution_time = end_time - start_time
print("Execution time: {:.2f} seconds".format(execution_time))
model_results
```

```
Execution time: 549.82 seconds
```

[19]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302

To improve the RandomForest model, I used cross-validation to optimize the hyperparameter tuning and reduce overfitting. Even with the addition of `SimpleImputer` and `StandardScaler`, the training error score remained the same, while the model performed worse on the test set. Using the median in imputing ensured the model's robustness and potentially reduced further overfitting.

0.5.8 Model 8: Explainable Gradient Boosting

[20]:

```python
# Defining the pipeline for Gradient Boosting
model8 = Pipeline([
    ('xgb_model', xgb.XGBRegressor(enable_categorical=True))
])

# Fitting the pipeline
model8.fit(X_train, y_train)

# Calculating the training error
train_error_model8 = calculateRMSLE(model8.predict(X_train), y_train)

# Calculating the test error
test_error_model8 = calculateRMSLE(model8.predict(X_test), y_test)

# Displaying the results
new_results_model8 = pd.DataFrame({'Model': ['Model 8 (Gradient Boosting)'],
                            'Train Error': [train_error_model8],
                            'Test Error': [test_error_model8]})
model_results = pd.concat([model_results, new_results_model8],␣
 ↪ignore_index=True)
model_results
```

[20]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377

The explainable Gradient Boosting model performed somewhat poorer than the RandomForest and was still overfitting.

0.5.9 Model 9: Improved Explainable Gradient Boosting

Adding a StandardScaler and using cross-validation to find the right parameters

[21]:
```
# Defining the pipeline for XGB
model9 = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('num', StandardScaler(), X_train.columns)
    ])),
    ('regressor', XGBRegressor(enable_categorical=True, objective='reg:
 ↪squarederror', random_state=20240419))
])

# Defining the parameter grid for hyperparameter tuning
param_grid = {
    'regressor__learning_rate': [0.01, 0.1, 0.2],
    'regressor__n_estimators': [50, 100, 200],
    'regressor__max_depth': [3, 5, 7],
}

# Initializing the GridSearchCV object
grid_search = GridSearchCV(estimator=model9, param_grid=param_grid,␣
 ↪scoring='neg_mean_squared_error', cv=5)

# Performing grid search
grid_search.fit(X_train, y_train)

# Getting the best hyperparameters
best_params = grid_search.best_params_

# Updating the model with the best hyperparameters
model9.set_params(**best_params)

# Fitting the pipeline with the best hyperparameters
model9.fit(X_train, y_train)

# Calculating the training error
train_error_model9 = calculateRMSLE(model9.predict(X_train), y_train)

# Calculating the test error
test_error_model9 = calculateRMSLE(model9.predict(X_test), y_test)

# Displaying the results
new_results_model9 = pd.DataFrame({'Model': ['Model 9 (Improved Gradient␣
 ↪Boosting)'],
```

```
                              'Train Error': [train_error_model9],
                              'Test Error': [test_error_model9]})
model_results = pd.concat([model_results, new_results_model9],␣
 ↪ignore_index=True)
model_results
```

[21]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220

To improve my GradientBoosting model, I defined a set of learning rates, estimators and different depths, and applied cross-validation to find the best parameters. With the addition of `StandardScaler` and CV, this model yielded the lowest test error, also performing the best on the unseen test dataset. While this model is still overfitting, the difference between the training and test error are considerably lower than in other models.

0.6 ## Neural networks

0.6.1 Model 10

Simple fully connected layer network with dropout

[22]:
```
# Setting the simple network
model10 = Sequential([
    Rescaling(1./255, input_shape=(60,)),
    Flatten(),
    Dense(256, activation='relu'),
    # Adding a Dropout of 0.5
    Dropout(0.5),
    # Output shape adjusted to the binary target variable
    Dense(1, activation='sigmoid')
])

# Compiling the model
model10.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])
print(model10.summary())
```

```
C:\Users\Zsófi\AppData\Roaming\Python\Python311\site-
packages\keras\src\layers\preprocessing\tf_data_layer.py:18: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
```

```
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(**kwargs)
```

Model: "sequential"

```
 Layer (type)                          Output Shape                          ␣
↪Param #

 rescaling (Rescaling)                 (None, 60)                               ␣
↪  0

 flatten (Flatten)                     (None, 60)                               ␣
↪  0

 dense (Dense)                         (None, 256)                          ␣
↪15,616

 dropout (Dropout)                     (None, 256)                              ␣
↪  0

 dense_1 (Dense)                       (None, 1)                                ␣
↪257
```

Total params: 15,873 (62.00 KB)

Trainable params: 15,873 (62.00 KB)

Non-trainable params: 0 (0.00 B)

None

[23]:
```
# Defining early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='accuracy', patience=5,␣
 ↪restore_best_weights=True)

# Training the model
history10 = model10.fit(X_train, y_train, epochs=25, validation_data=(X_test,␣
 ↪y_test), callbacks=[early_stopping])
```

```
Epoch 1/25
744/744                   2s 2ms/step -
accuracy: 0.7850 - loss: 24.6533 - val_accuracy: 0.8699 - val_loss: 0.5318
```

```
Epoch 2/25
744/744              1s 2ms/step -
accuracy: 0.8439 - loss: 0.6073 - val_accuracy: 0.8762 - val_loss: 0.4903
Epoch 3/25
744/744              1s 2ms/step -
accuracy: 0.8683 - loss: 0.5083 - val_accuracy: 0.8764 - val_loss: 0.3938
Epoch 4/25
744/744              1s 2ms/step -
accuracy: 0.8757 - loss: 0.4147 - val_accuracy: 0.8767 - val_loss: 0.3795
Epoch 5/25
744/744              1s 2ms/step -
accuracy: 0.8778 - loss: 0.3865 - val_accuracy: 0.8767 - val_loss: 0.3770
Epoch 6/25
744/744              1s 2ms/step -
accuracy: 0.8780 - loss: 0.3829 - val_accuracy: 0.8769 - val_loss: 0.3650
Epoch 7/25
744/744              1s 2ms/step -
accuracy: 0.8793 - loss: 0.3730 - val_accuracy: 0.8774 - val_loss: 0.3629
Epoch 8/25
744/744              1s 2ms/step -
accuracy: 0.8793 - loss: 0.3679 - val_accuracy: 0.8774 - val_loss: 0.3763
Epoch 9/25
744/744              1s 2ms/step -
accuracy: 0.8724 - loss: 0.3828 - val_accuracy: 0.8771 - val_loss: 0.3623
Epoch 10/25
744/744              1s 2ms/step -
accuracy: 0.8794 - loss: 0.3645 - val_accuracy: 0.8772 - val_loss: 0.3619
Epoch 11/25
744/744              1s 2ms/step -
accuracy: 0.8757 - loss: 0.3718 - val_accuracy: 0.8774 - val_loss: 0.3607
Epoch 12/25
744/744              2s 2ms/step -
accuracy: 0.8796 - loss: 0.3678 - val_accuracy: 0.8774 - val_loss: 0.3694
Epoch 13/25
744/744              1s 2ms/step -
accuracy: 0.8779 - loss: 0.3725 - val_accuracy: 0.8774 - val_loss: 0.3700
Epoch 14/25
744/744              1s 2ms/step -
accuracy: 0.8795 - loss: 0.3664 - val_accuracy: 0.8774 - val_loss: 0.3684
Epoch 15/25
744/744              1s 2ms/step -
accuracy: 0.8787 - loss: 0.3726 - val_accuracy: 0.8769 - val_loss: 0.3732
Epoch 16/25
744/744              1s 2ms/step -
accuracy: 0.8780 - loss: 0.3759 - val_accuracy: 0.8774 - val_loss: 0.3739
Epoch 17/25
744/744              1s 2ms/step -
accuracy: 0.8800 - loss: 0.3686 - val_accuracy: 0.8774 - val_loss: 0.3734
```

```
Epoch 18/25
744/744              2s 2ms/step -
accuracy: 0.8785 - loss: 0.3703 - val_accuracy: 0.8766 - val_loss: 0.3745
Epoch 19/25
744/744              2s 3ms/step -
accuracy: 0.8739 - loss: 0.3801 - val_accuracy: 0.8774 - val_loss: 0.3755
Epoch 20/25
744/744              2s 3ms/step -
accuracy: 0.8745 - loss: 0.3782 - val_accuracy: 0.8774 - val_loss: 0.3735
```

[24]:

```python
# Defining a function to evaluate the models in terms of RMSLE and creating a␣
↪df with the results
def evaluate_model(model, X_train, y_train, X_test, y_test, model_name):
    # Predictions on the training set
    y_train_pred = model.predict(X_train).flatten()

    # Predictions on the test set
    y_test_pred = model.predict(X_test).flatten()

    # Calculate RMSLE for the training set
    train_rmsle = calculateRMSLE(y_train_pred, y_train)

    # Calculate RMSLE for the test set
    test_rmsle = calculateRMSLE(y_test_pred, y_test)

    # Displaying the results
    new_results = pd.DataFrame({'Model': [model_name],
                                'Train Error': [train_rmsle],
                                'Test Error': [test_rmsle]})
    return new_results

# Evaluating the model
new_results_model10 = evaluate_model(model10, X_train, y_train, X_test, y_test,␣
↪'Model 10 (Simple network with dropouts)')
model_results = pd.concat([model_results, new_results_model10],␣
↪ignore_index=True)
model_results
```

```
744/744              1s 1ms/step
186/186              0s 2ms/step
```

[24]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489

```
5                   Model 6 (Random Forest)       0.0804       0.2298
6          Model 7 (Improved Random Forest)       0.0804       0.2302
7                Model 8 (Gradient Boosting)      0.1322       0.2377
8     Model 9 (Improved Gradient Boosting)        0.2140       0.2220
9  Model 10 (Simple network with dropouts)        0.2292       0.2301
```

For ease of comparison, as the first neutral network model, I set a simple network of fully connected layers and one dropout layer. The binary classification problem required a sigmoid function, therefore I adjusted it to the binary target variable. This model performed similar to the RandomForest in terms of test error, but was still overfitting.

0.6.2 Model 11

Convolutional neural network with dropout

[25]:
```
# Setting the CNN network
model11 = Sequential([
    Rescaling(1./255, input_shape=(60, 1)),
    Conv1D(32, 3, activation='relu'),
    MaxPooling1D(2),
    Conv1D(64, 3, activation='relu'),
    MaxPooling1D(2),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

# Compiling the model
model11.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])
print(model11.summary())
```

```
C:\Users\Zsófi\AppData\Roaming\Python\Python311\site-
packages\keras\src\layers\preprocessing\tf_data_layer.py:18: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(**kwargs)
```

```
Model: "sequential_1"


 Layer (type)                       Output Shape                          ␣
↪Param #

 rescaling_1 (Rescaling)            (None, 60, 1)                             ␣
↪  0
```

```
conv1d (Conv1D)                    (None, 58, 32)
↪128

max_pooling1d (MaxPooling1D)       (None, 29, 32)
↪  0

conv1d_1 (Conv1D)                  (None, 27, 64)
↪6,208

max_pooling1d_1 (MaxPooling1D)     (None, 13, 64)
↪  0

flatten_1 (Flatten)                (None, 832)
↪  0

dense_2 (Dense)                    (None, 256)
↪213,248

dropout_1 (Dropout)                (None, 256)
↪  0

dense_3 (Dense)                    (None, 1)
↪257
```

Total params: 219,841 (858.75 KB)

Trainable params: 219,841 (858.75 KB)

Non-trainable params: 0 (0.00 B)

None

[26]:
```
# Defining early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='accuracy', patience=5,␣
↪restore_best_weights=True)

# Training the model
history11 = model11.fit(X_train, y_train, epochs=25, validation_data=(X_test,␣
↪y_test), callbacks=[early_stopping])
```

```
Epoch 1/25
744/744                7s 7ms/step -
accuracy: 0.8262 - loss: 5.7522 - val_accuracy: 0.8774 - val_loss: 0.3934
```

```
Epoch 2/25
744/744              5s 6ms/step -
accuracy: 0.8738 - loss: 0.4037 - val_accuracy: 0.8767 - val_loss: 0.3697
Epoch 3/25
744/744              5s 6ms/step -
accuracy: 0.8795 - loss: 0.3841 - val_accuracy: 0.8756 - val_loss: 0.3710
Epoch 4/25
744/744              5s 6ms/step -
accuracy: 0.8803 - loss: 0.3786 - val_accuracy: 0.8774 - val_loss: 0.4275
Epoch 5/25
744/744              5s 6ms/step -
accuracy: 0.8754 - loss: 0.3824 - val_accuracy: 0.8771 - val_loss: 0.3693
Epoch 6/25
744/744              5s 7ms/step -
accuracy: 0.8777 - loss: 0.3716 - val_accuracy: 0.8774 - val_loss: 0.3655
Epoch 7/25
744/744              6s 7ms/step -
accuracy: 0.8775 - loss: 0.3727 - val_accuracy: 0.8774 - val_loss: 0.3657
Epoch 8/25
744/744              9s 6ms/step -
accuracy: 0.8791 - loss: 0.3656 - val_accuracy: 0.8774 - val_loss: 0.3596
Epoch 9/25
744/744              5s 6ms/step -
accuracy: 0.8787 - loss: 0.3625 - val_accuracy: 0.8774 - val_loss: 0.3592
```

[27]:
```python
# Evaluating the model
new_results_model11 = evaluate_model(model11, X_train, y_train, X_test, y_test,
 ↪'Model 11 (Convolutional neural network with dropout)')
model_results = pd.concat([model_results, new_results_model11],
 ↪ignore_index=True)
model_results
```

```
744/744              2s 2ms/step
186/186              0s 2ms/step
```

[27]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220
9	Model 10 (Simple network with dropouts)	0.2292	0.2301
10	Model 11 (Convolutional neural network with dr…	0.2292	0.2305

Adding convolutional layers and pooling might seem unconventional at first, as CNNs are mostly associated with image data, however, experimenting with these additional layers for regularization didn't yield positive results for the model's performance. Both error scores increased, making it a weaker performer compared to the simple network.

0.6.3 Model 12

Simple fully connected layer with increased depth

[28]:
```
# Setting the network
model12 = Sequential([
    Rescaling(1./255, input_shape=(60,)),
    Flatten(),
    Dropout(0.5),
    # Adding more layers to increase depth
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(128, activation = 'relu'),
    Dropout(0.5),
    Dense(64, activation = 'relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

# Compiling the model
model12.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])
print(model12.summary())
```

```
C:\Users\Zsófi\AppData\Roaming\Python\Python311\site-
packages\keras\src\layers\preprocessing\tf_data_layer.py:18: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(**kwargs)
```

```
Model: "sequential_2"


 Layer (type)                      Output Shape                           ␣
↪Param #

 rescaling_2 (Rescaling)           (None, 60)                               ␣
↪   0

 flatten_2 (Flatten)               (None, 60)                               ␣
↪   0
```

```
dropout_2 (Dropout)             (None, 60)                        ␣
↪  0

dense_4 (Dense)                 (None, 256)                     ␣
↪15,616

dropout_3 (Dropout)             (None, 256)                       ␣
↪  0

dense_5 (Dense)                 (None, 128)                     ␣
↪32,896

dropout_4 (Dropout)             (None, 128)                       ␣
↪  0

dense_6 (Dense)                 (None, 64)                       ␣
↪8,256

dropout_5 (Dropout)             (None, 64)                        ␣
↪  0

dense_7 (Dense)                 (None, 1)                         ␣
↪ 65
```

Total params: 56,833 (222.00 KB)

Trainable params: 56,833 (222.00 KB)

Non-trainable params: 0 (0.00 B)

None

[29]:

```
# Defining early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='accuracy', patience=5,␣
↪restore_best_weights=True)

# Training the model
history12 = model12.fit(X_train, y_train, epochs=25, validation_data=(X_test,␣
↪y_test), callbacks=[early_stopping])
```

```
Epoch 1/25
744/744               3s 3ms/step -
accuracy: 0.7722 - loss: 28.7793 - val_accuracy: 0.8774 - val_loss: 0.4841
Epoch 2/25
```

```
744/744              2s 3ms/step -
accuracy: 0.8426 - loss: 1.3002 - val_accuracy: 0.8774 - val_loss: 0.4153
Epoch 3/25
744/744              2s 2ms/step -
accuracy: 0.8716 - loss: 0.5305 - val_accuracy: 0.8774 - val_loss: 0.3918
Epoch 4/25
744/744              2s 3ms/step -
accuracy: 0.8770 - loss: 0.4444 - val_accuracy: 0.8774 - val_loss: 0.3764
Epoch 5/25
744/744              2s 2ms/step -
accuracy: 0.8791 - loss: 0.3907 - val_accuracy: 0.8774 - val_loss: 0.3729
Epoch 6/25
744/744              2s 2ms/step -
accuracy: 0.8753 - loss: 0.4006 - val_accuracy: 0.8774 - val_loss: 0.3721
Epoch 7/25
744/744              3s 2ms/step -
accuracy: 0.8767 - loss: 0.3851 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 8/25
744/744              2s 2ms/step -
accuracy: 0.8767 - loss: 0.3781 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 9/25
744/744              2s 3ms/step -
accuracy: 0.8794 - loss: 0.3709 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 10/25
744/744              2s 2ms/step -
accuracy: 0.8748 - loss: 0.3793 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 11/25
744/744              2s 2ms/step -
accuracy: 0.8785 - loss: 0.3703 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 12/25
744/744              2s 2ms/step -
accuracy: 0.8775 - loss: 0.3736 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 13/25
744/744              2s 2ms/step -
accuracy: 0.8771 - loss: 0.3747 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 14/25
744/744              2s 2ms/step -
accuracy: 0.8769 - loss: 0.3736 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 15/25
744/744              3s 2ms/step -
accuracy: 0.8807 - loss: 0.3668 - val_accuracy: 0.8774 - val_loss: 0.3720
Epoch 16/25
744/744              2s 2ms/step -
accuracy: 0.8824 - loss: 0.3642 - val_accuracy: 0.8774 - val_loss: 0.3720
```

[30]:

```
# Evaluating the model
```

```
new_results_model12 = evaluate_model(model12, X_train, y_train, X_test, y_test,␣
↪'Model 12 (Increased width with dropouts)')
model_results = pd.concat([model_results, new_results_model12],␣
↪ignore_index=True)
model_results
```

```
744/744              1s 1ms/step
186/186              0s 1ms/step
```

[30]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220
9	Model 10 (Simple network with dropouts)	0.2292	0.2301
10	Model 11 (Convolutional neural network with dr…	0.2292	0.2305
11	Model 12 (Increased width with dropouts)	0.2285	0.2293

In Model 12, I increased the depth of the simple network as deeper networks can learn more discriminative features that can be decisive in binary classification predictions. By dropping `MaxPooling` and `Conv1D`, lower error scores were achieved, further proving that complex CNN models are more suited to image data or other, more dimensional datasets. Model 12 performed very similar to the simple network with fully connected layers, implying that increasing the depth of the network barely improved the predictions. There was only a slight improvement in the test error.

0.6.4 Model 13

Simple fully connected neural network with increased dropout, and an additional learning rate for adam

[31]:

```
# Setting the network
model13 = Sequential([
    Flatten(input_shape=(60,)),
    Dense(64, activation='relu'),
    # Increasing the Dropout value
    Dropout(0.6),
    # Adding BatchNormalization
    BatchNormalization(),
    Dense(32, activation='relu'),
    Dropout(0.6),
    BatchNormalization(),
    Dense(1, activation='sigmoid')
])
```

```
# Defining the optimizer with a lower learning rate
optimizer = Adam(learning_rate=0.01)

# Compiling the model
model13.compile(loss='binary_crossentropy', optimizer=optimizer,␣
 ↪metrics=['accuracy'])
print(model13.summary())
```

```
C:\Users\Zsófi\AppData\Roaming\Python\Python311\site-
packages\keras\src\layers\reshaping\flatten.py:37: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

```
Model: "sequential_3"



 Layer (type)                          Output Shape                           ␣
↪Param #

 flatten_3 (Flatten)                   (None, 60)                               ␣
↪   0

 dense_8 (Dense)                       (None, 64)                             ␣
↪3,904

 dropout_6 (Dropout)                   (None, 64)                               ␣
↪   0

 batch_normalization                   (None, 64)                               ␣
↪256
 (BatchNormalization)                                                           ␣
↪

 dense_9 (Dense)                       (None, 32)                             ␣
↪2,080

 dropout_7 (Dropout)                   (None, 32)                               ␣
↪   0

 batch_normalization_1                 (None, 32)                               ␣
↪128
 (BatchNormalization)                                                           ␣
↪
```

```
 dense_10 (Dense)                     (None, 1)                                    ␣
↪ 33
```

```
 Total params: 6,401 (25.00 KB)

 Trainable params: 6,209 (24.25 KB)

 Non-trainable params: 192 (768.00 B)
```

None

[32]:
```
# Defining early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='accuracy', patience=5,␣
 ↪restore_best_weights=True)

# Training the model
history13 = model13.fit(X_train, y_train, epochs=25, validation_data=(X_test,␣
 ↪y_test), callbacks=[early_stopping])
```

```
Epoch 1/25
744/744                4s 2ms/step -
accuracy: 0.8468 - loss: 0.4202 - val_accuracy: 0.8761 - val_loss: 0.3721
Epoch 2/25
744/744                2s 2ms/step -
accuracy: 0.8855 - loss: 0.3577 - val_accuracy: 0.8749 - val_loss: 0.3768
Epoch 3/25
744/744                2s 2ms/step -
accuracy: 0.8745 - loss: 0.3780 - val_accuracy: 0.8752 - val_loss: 0.3793
Epoch 4/25
744/744                2s 2ms/step -
accuracy: 0.8783 - loss: 0.3711 - val_accuracy: 0.8746 - val_loss: 0.4300
Epoch 5/25
744/744                2s 2ms/step -
accuracy: 0.8830 - loss: 0.3611 - val_accuracy: 0.8747 - val_loss: 0.4148
Epoch 6/25
744/744                2s 2ms/step -
accuracy: 0.8821 - loss: 0.3630 - val_accuracy: 0.8747 - val_loss: 0.5482
Epoch 7/25
744/744                3s 2ms/step -
accuracy: 0.8766 - loss: 0.3741 - val_accuracy: 0.8752 - val_loss: 0.4628
```

[33]:
```
# Evaluating the model
new_results_model13 = evaluate_model(model13, X_train, y_train, X_test, y_test,␣
 ↪'Model 13 (NN with learning rate and increased dropouts)')
```

```
model_results = pd.concat([model_results, new_results_model13],␣
↪ignore_index=True)
model_results
```

```
744/744              1s 1ms/step
186/186              0s 1ms/step
```

[33]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220
9	Model 10 (Simple network with dropouts)	0.2292	0.2301
10	Model 11 (Convolutional neural network with dr…	0.2292	0.2305
11	Model 12 (Increased width with dropouts)	0.2285	0.2293
12	Model 13 (NN with learning rate and increased …	0.2317	0.2324

To further experiment with neural networks, I added a layer of `BatchNormalization` to improve generalization. By trying different learning rates, I wanted to see if the number of epochs can be decreased. A moderate learning rate, set to 0.01 did result in lower number of epochs, thus a faster learning process. Moreover, I increased the probability values of both Dropout layers. The result was that this model performed worse than the simple neural networks. However, experimenting separately with different dropout values, I have found that increasing the probabilities resulted in better test error scores. Therefore, we can assume that the BatchNormalization layer weakens the neural network's performance applied to this specific dataset.

0.7 ## Improving the best performing models

Picking my best performing linear and Gradient Boosting models and adding further improvements, such as regularization methods

0.7.1 Model 14

Logistic model with Ridge regularization

[34]:
```
# Defining the pipeline for the logistic regression model with L2 regularization
model14 = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('num', StandardScaler(), X_train.columns)
    ])),
    ('classifier', LogisticRegression(penalty='l2', C=0.0002,␣
↪random_state=prng))
])
```

```
# Fitting the model
model14.fit(X_train, y_train)

# Calculating the train error
train_error_log_l2 = calculateRMSLE(model14.predict(X_train), y_train)

# Calculating the test error
test_error_log_l2 = calculateRMSLE(model14.predict(X_test), y_test)

# Displaying the results
new_results_log_l2 = pd.DataFrame({'Model': ['Model 14 (Logistic with L2␣
↪Regularization)'],
                                   'Train Error': [train_error_log_l2],
                                   'Test Error': [test_error_log_l2]})
model_results = pd.concat([model_results, new_results_log_l2],␣
↪ignore_index=True)
model_results
```

[34]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220
9	Model 10 (Simple network with dropouts)	0.2292	0.2301
10	Model 11 (Convolutional neural network with dr…	0.2292	0.2305
11	Model 12 (Increased width with dropouts)	0.2285	0.2293
12	Model 13 (NN with learning rate and increased …	0.2317	0.2324
13	Model 14 (Logistic with L2 Regularization)	0.2415	0.2432

To further improve my models, I decided to pick the ones that performed well or consistently and apply some regularization techniques to see if they will perform better than the improved gradient boosting model. I have found that adding a Ridge regularization with a relatively low C value to the logistic model does not really improve the model as the error scores were very similar to the original model.

0.7.2 Model 15

Linear model with Ridge regularization

[35]:
```
# Setting the pipeline for the Ridge regression model
model15 = Pipeline([
    ('preprocessor', ColumnTransformer([
```

```
        ('num', StandardScaler(), X_train.columns)
    ])),
    ('regressor', Ridge(alpha=1))
])

# Fitting the model
model15.fit(X_train, y_train)

# Calculating the train error
train_error_linear_ridge = calculateRMSLE(model15.predict(X_train), y_train)

# Calculating the test error
test_error_linear_ridge = calculateRMSLE(model15.predict(X_test), y_test)

# Displaying the results
new_results_linear_ridge = pd.DataFrame({"Model": ["Model 15 (Linear model with␣
 ↪Ridge)"],
                                    "Train Error": [train_error_linear_ridge],
                                    "Test Error": [test_error_linear_ridge]})

model_results = pd.concat([model_results, new_results_linear_ridge],␣
 ↪ignore_index=True)
model_results
```

[35]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220
9	Model 10 (Simple network with dropouts)	0.2292	0.2301
10	Model 11 (Convolutional neural network with dr…	0.2292	0.2305
11	Model 12 (Increased width with dropouts)	0.2285	0.2293
12	Model 13 (NN with learning rate and increased …	0.2317	0.2324
13	Model 14 (Logistic with L2 Regularization)	0.2415	0.2432
14	Model 15 (Linear model with Ridge)	0.2222	0.2237

As the simple linear model obtained one of the lowest test error scores, and there was still a lot of options to improve it, I decided to experiment with different regularization methods. However, adding Ridge regularization didn't change the error scores, meaning that that the simple linear model was generalizing well to unseen data despite the overfitting. I was assuming that it might be worth to explore different regularization strengths so I used cross-validation to find the best alpha.

0.7.3 Model 16

Linear model with Ridge regularization, using cross-validation

[36]:
```python
# Defining the parameter grid for alpha values
param_grid = {'regressor__alpha': [0.1, 0.5, 1.0, 5.0, 10.0]}

# Initializing the GridSearchCV object
grid_search = GridSearchCV(estimator=model15, param_grid=param_grid,␣
 ↪scoring='neg_mean_squared_error', cv=5)

# Performing grid search
grid_search.fit(X_train, y_train)

# Getting the best alpha
best_alpha = grid_search.best_params_['regressor__alpha']

# Updating the model with the best alpha
model16 = model15.set_params(regressor__alpha=best_alpha)

# Fitting the model
model16.fit(X_train, y_train)

# Calculate the train and test errors
train_error_ridge_best = calculateRMSLE(model16.predict(X_train), y_train)
test_error_ridge_best = calculateRMSLE(model16.predict(X_test), y_test)

# Display the results
new_results_model16 = pd.DataFrame({"Model": ["Model 16 (Linear model with␣
 ↪Ridge - Best Alpha)"],
                                          "Train Error":␣
 ↪[train_error_ridge_best],
                                          "Test Error": [test_error_ridge_best]})
model_results = pd.concat([model_results, new_results_model16],␣
 ↪ignore_index=True)
model_results
```

[36]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220
9	Model 10 (Simple network with dropouts)	0.2292	0.2301

```
10  Model 11 (Convolutional neural network with dr…       0.2292      0.2305
11           Model 12 (Increased width with dropouts)       0.2285      0.2293
12  Model 13 (NN with learning rate and increased …       0.2317      0.2324
13         Model 14 (Logistic with L2 Regularization)       0.2415      0.2432
14                 Model 15 (Linear model with Ridge)       0.2222      0.2237
15    Model 16 (Linear model with Ridge - Best Alpha)       0.2222      0.2237
```

Even after experimenting with different lambdas, the scores remained unchanged. This indicates that the simple linear model might be insensitive to regularization or further justify that the model is already generalizing well to unseen data. However, I wanted to see if the model is actually insensitive or a different regularization technique can lower the error scores.

0.7.4 Model 17

Linear model with ElasticNet, using cross-validation

[37]:
```
# Defining ElasticNet model
elastic_net = ElasticNet()

# Defining hyperparameters grid for tuning
param_grid = {
    'alpha': [0.1, 1, 10, 25, 50],
    'l1_ratio': [0.0001, 0.001, 0.1, 0.5, 0.9]
}

# Performing grid search cross-validation to find the best hyperparameters
grid_search = GridSearchCV(elastic_net, param_grid, cv=5,␣
 ↪scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

# Getting the best hyperparameters
best_alpha = grid_search.best_params_['alpha']
best_l1_ratio = grid_search.best_params_['l1_ratio']

# Training the model
model17 = ElasticNet(alpha=best_alpha, l1_ratio=best_l1_ratio)
model17.fit(X_train, y_train)

# Calculating the train and test errors
train_error_model17 = calculateRMSLE(model17.predict(X_train), y_train)
test_error_model17 = calculateRMSLE(model17.predict(X_test), y_test)

# Display the results
new_results_model17 = pd.DataFrame({"Model": ["Model 17 (Linear model with␣
 ↪ElasticNet - Best Alpha)"],
                                        "Train Error": [train_error_model17],
                                        "Test Error": [test_error_model17]})
```

```
model_results = pd.concat([model_results, new_results_model17],␣
↪ignore_index=True)
model_results
```

[37]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220
9	Model 10 (Simple network with dropouts)	0.2292	0.2301
10	Model 11 (Convolutional neural network with dr…	0.2292	0.2305
11	Model 12 (Increased width with dropouts)	0.2285	0.2293
12	Model 13 (NN with learning rate and increased …	0.2317	0.2324
13	Model 14 (Logistic with L2 Regularization)	0.2415	0.2432
14	Model 15 (Linear model with Ridge)	0.2222	0.2237
15	Model 16 (Linear model with Ridge - Best Alpha)	0.2222	0.2237
16	Model 17 (Linear model with ElasticNet - Best …	0.2223	0.2241

I applied the combination of L1 and L2 to analyze if a different regularization technique can further improve the accuracy of the simple linear model. Unfortunately, ElasticNet brought around somewhat higher error scores. This concluded that the simple linear model may not benefit from further improvements using regularization.

0.7.5 Model 18

Feature engineered XGB

[38]:

```
# Defining the pipeline with StandardScaler, Regularization, and XGBoost model
model18 = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('num', StandardScaler(), X_train_fe.columns)
    ])),
    ('regressor', XGBRegressor(enable_categorical=True, objective='reg:
↪squarederror', random_state=20240419))
])

# Defining the parameter grid for hyperparameter tuning
param_grid = {
    'regressor__learning_rate': [0.01, 0.1, 0.2],
    'regressor__n_estimators': [50, 100, 200],
    'regressor__max_depth': [3, 5, 7],
}
```

```
# Initializing the GridSearchCV object
grid_search = GridSearchCV(estimator=model18, param_grid=param_grid,␣
↪scoring='neg_mean_squared_error', cv=5)

# Performing grid search
grid_search.fit(X_train_fe, y_train)

# Getting the best hyperparameters
best_params = grid_search.best_params_

# Updating the model with the best hyperparameters
model18.set_params(**best_params)

# Fitting the pipeline with the best hyperparameters
model18.fit(X_train_fe, y_train)

# Calculating the training error
train_error_model18 = calculateRMSLE(model18.predict(X_train_fe), y_train)

# Calculating the test error
test_error_model18 = calculateRMSLE(model18.predict(X_test_fe), y_test)

# Displaying the results
new_results_model18 = pd.DataFrame({'Model': ['Model 18 (Improved Gradient␣
↪Boosting with FE)'],
                            'Train Error': [train_error_model18],
                            'Test Error': [test_error_model18]})
model_results = pd.concat([model_results, new_results_model18],␣
↪ignore_index=True)
model_results
```

[38]:

	Model	Train Error	Test Error
0	Model 1 (Linear)	0.2222	0.2237
1	Model 2 (Logistic)	0.2419	0.2435
2	Model 3 (Polynomial linear)	0.2097	0.5708
3	Model 4 (Decision tree)	0.2205	0.2265
4	Model 5 (Improved Decision tree)	0.1925	0.2489
5	Model 6 (Random Forest)	0.0804	0.2298
6	Model 7 (Improved Random Forest)	0.0804	0.2302
7	Model 8 (Gradient Boosting)	0.1322	0.2377
8	Model 9 (Improved Gradient Boosting)	0.2140	0.2220
9	Model 10 (Simple network with dropouts)	0.2292	0.2301
10	Model 11 (Convolutional neural network with dr…	0.2292	0.2305
11	Model 12 (Increased width with dropouts)	0.2285	0.2293
12	Model 13 (NN with learning rate and increased …	0.2317	0.2324
13	Model 14 (Logistic with L2 Regularization)	0.2415	0.2432
14	Model 15 (Linear model with Ridge)	0.2222	0.2237

15	Model 16 (Linear model with Ridge - Best Alpha)	0.2222	0.2237
16	Model 17 (Linear model with ElasticNet - Best …	0.2223	0.2241
17	Model 18 (Improved Gradient Boosting with FE)	0.2247	0.2262

Considering all models, improved gradient boosting still performed the lowest test error scores, therefore I decided to fit it to my feature engineered data sets. Previously, I grouped the variables into several categories and ran them on my best models, however, they performed considerably worse than the original models. Therefore, I opted to create a few extra variables instead and include them in the feature engineered data set. However, the model performed worse both in terms of training and test errors, making the original gradient boosting model the best performing model.

0.8 ## Evaluation

Predictive models: After analyzing the final results table, I concluded that the improved Gradient Boosting has the lowest test RMSLE score. As the simple, linear models were also performing well, I tried to improve them with different regularization techniques, such as Ridge, Lasso and ElasticNet, and used cross-validation to find the best regularization parameters. For most of these improved models, the RMSLE scores were very similar. I have also found that my first linear model with Ridge regression already uses the best parameters, hence the cross-validation in the linear models have not improved the model. The flexible model was considerably overfitted to the training data. Due to the large number of predictors, the 3 and 4 degree polynomials increased the computational intensity and significantly increased the run time, therefore I could only compute the results of the 2-degree poly model. I also experimented with grouped variables, but the model obtained higher train and test error scores, therefore I opted to drop the grouped variables. Regarding the other models, RandomForest and GradientBoosting yielded the best training RMSLE results. StandardScaler improved my RandomForest model, however it largely overfitted my decision tree. Improving the Explainable Gradient Boosting by using StandardScaler and the cross-validation method yielded the best test error scores. The scores were lower than what the neural networks produced.

Neural networks: When I was experimenting with different neural networks, I have found that including MaxPooling and several convolutional layers weakens the model performance, therefore I opted to exclude them. After experimenting with several dropout values, I have noticed that as soon as I increased the dropout from 0.4 to 0.6, the training and test errors decreased. By dropping out more neurons during training, models performed better in terms of generalization and were less likely to overfit the training data. However, they still didn't achieve better scores than the improved gradient boosting model. It might be due to the reason that simpler models may perform better than complex neural networks, especially if the dataset is not large or complex enough (like images) to benefit from the additional capacity of the neural networks. Therefore, this dataset might be too simple for neural networks to perform well.

The best model: I wanted to improve my best performing Gradient Boosting model with feature engineering. As experimenting with grouped variables showed that including them weakens the performance of the model, I chose to include 3 additional columns `total_multimedia`, `easy_to_consume` and `hard_to_consume`. Both the training and test RMSLE scores were higher, implying that feature engineering is not likely to improve the Gradient Boosting model. The only limitation still was that all my models were overfitting and despite applying regularization techniques, hyperparameter tuning, cross-validation or scaling, I couldn't achive a well-fitted model.

[39]:
```
# Saving the predictions of a certain model
def save_predictions(model, test_df, model_number):
    scores = model.predict(test_df)

    # Getting the article_id column from test_df
    article_ids = test_df['article_id']

    # Creating a df for the predictions
    df = pd.DataFrame({'article_id': article_ids, 'score': scores})

    # Saving predictions to a CSV file
    df.to_csv(f'predictions_model{model_number}.csv', index=False)

save_predictions(model9, test_df, 9)
```