**MACHINE LEARNING PROJECT REPORT**

Thomas Colangelo & Jacopo Gentile

1. Load dependencies

The first step is to import all the modules we will need, such as:

* Pandas
* Matplotlib
* Seaborn
* Numpy
* Train test split, Randomized Search CV, learning curve from Sklearn
* Mean squared error, mean absolute error, r2 score from Sklearn
* Warnings
* Label Encoder, MinMax Scaler
* XGB Regressor
* Sequential and Model from Keras
* Dense and Input from Keras
* SHAP

1. Import the model dataset and the prediction dataset
2. Preprocessing
   1. Firstly, we remove the whitespaces from column names. For examples ‘ shares’ is now ‘shares’
   2. We check for null values in both datasets. There are none.
   3. Outlier removal and categorical variable encoding. We identify and store categorical variables by iterating through all the columns and storing columns with data type object as categorical, encode them with Label Encoder and then remove outliers from the dataset with a threshold of 10 z scores. We print the number of rows before and after the operations to demonstrate the effect of the preprocessing.
3. Feature Engineering

We decided to create some new features since some of the originals were not informative and hard to interpret. For example, the original dataset contains 9 features dedicated to keywords with ambiguous names and even more ambiguous values that appear completely unrelated to each other. The features we created are:

* Keyword\_avg\_score: kw\_avg\_avg renamed
* Keyword\_min\_score: average between kw\_avg\_min and kw\_max\_min
* keyword\_max\_score: average between kw\_avg\_max and kw\_min\_max
* keyword\_max\_score\_proportion: kw\_max\_max divided by its biggest most common value, 843300. We created this to make the feature more informative.
* keyword\_min\_score\_proportion: kw\_min\_min divided by its biggest most common value, 217. Before dividing we added 2 to make it positive.
* rate\_neutral\_words: 1 – (global\_rate\_positive\_words + global\_rate\_negative\_words)
* rate\_positive\_words: global\_rate\_positive\_words renamed
* rate\_negative\_words: global\_rate\_negative\_words renamed

We also dropped redundant columns:

* kw\_avg\_avg
* kw\_max\_avg
* kw\_min\_avg
* kw\_avg\_min
* kw\_max\_min
* kw\_min\_min
* kw\_avg\_max
* ke\_max\_max
* kw\_min\_max
* abs\_title\_subjectivity
* abs\_title\_sentiment\_polarity
* global\_rate\_positive\_words
* global\_rate\_negative\_words

We then print out our new columns to check their values. Overall, this helped us decrease multicollinearity and improve interpretability.

After this, we plot the correlation heatmap for the modified model dataframe and print out the top 10 correlated features to shares.

The scores for all of them are very low, except keyword\_avg\_score. This suggests that the relationships could be non-linear.

1. Train Test Split, we split the data into train and test sets, with the y train and test sets containing the shares column and the x train and test sets containing the rest of the features. We ensured consistency by setting a random state of 40 and chose a conventional training size of 0.2
2. Scaling

We scale both the model and training dataset using MinMax scaler. This scaler offered the best performance out of all the ones we tried (Robust, Standard, MaxAbs, Normalizer, QuantileTransformer, PowerTransformer). For the model dataset, we scale only the X sets (X train and X test) since we don’t want to train our model to predict scaled shares values. We also make sure to not scale categorical variables. We do this by excluding them from scaling using list comprehension and the categorical variables list we created during preprocessing.

Finally, we plot box plots for all features in our dataset to check their distributions.

We then concatenate back together the unscaled categorical variables with our scaled features using pandas. Again, we plot out variables to check that everything went according to plan. We also print mean and standard deviation of all variables.

Upon examining the data, we decide to log transform it to improve our model’s understanding of the data but most importantly our GANs ability to create convincing synthetic data. Here, unlike before, we do also log transform the target variable to ensure consistency, reduce peakedness and try normalizing it.

We plot our target variable before and after the log transformation to check the effects. Peakedness has drastically reduced, as well as skewness. Distribution also appears close to normality so we will be content with this result. Again, we print mean and standard deviation to observe the effects of the transformation.

1. Data Augmentation with GAN

During our experimentation with this dataset, we kept being able to improve our results until we hit a plateau. It seemed like there was no way to further improve our data and our model. That is when we tried changing the train test split to include more and more training data and we noticed that performance improved dramatically with the inclusion of more training data. This, however, wasn’t a completely correct approach: by using a smaller testing dataset, we ran the risk of fortuitously getting better results since maybe the smaller test dataset simply had some similar rows to the much vaster training dataset. Our model’s performance would be inconsistent at best and fraudulent at worst. I then ran into the concept of synthetic data and data augmentation. After hours of research I stumbled upon GANs, also known as Generative Adversary Networks, a class of artificial intelligence algorithms used in unsupervised machine learning, introduced by Ian Goodfellow and his colleagues in 2014.

In a GAN, two neural networks, called the generator and the discriminator, are trained simultaneously through a competitive process. The generator tries to produce data that is indistinguishable from real data, while the discriminator tries to distinguish between real data and fake data generated by the generator.

The generator and discriminator are trained iteratively, where the generator tries to improve its ability to produce realistic data by fooling the discriminator, and the discriminator tries to improve its ability to distinguish between real and fake data. This adversarial process leads to the generator producing increasingly realistic data over time.

While GANs are usually best suited to classification tasks (for example, in image classification it would produce more training data by rotating or altering the gamma of an image), I decided to try and apply it to our regression task.

We build a generator and a discriminator using neural networks and have them go to work by feeding them our training data. The results are then concatenated with our train test datasets.

One issue with GANs is that they don’t predict the target variable, instead they label it as 1 to label that row as synthetically generated. This limited the accuracy of our regressor, and I did try to implement a cGAN, also known as conditional GAN, which would predict the target variable too, but honestly, I ran out of time and GANs are already stretching the limits of my comprehension of machine learning. Undertaking cGANs right now would be a stretch. Maybe next time.

After creating our synthetic data and concatenating it with our train test dataframes, we plot the distribution of the augmented x train to check the results. They don’t look too far off from the original X train, so we’ll take that.

For safety’s sake, we print out the number of rows added to our split and filter rows where the categorical variables are above their limit of 6, just like they were in the original train dataframe.

In this case, no rows get filtered out but it’s better to be safe than sorry.

Now we plot our augmented x train alongside our original x train to again check the distribution.

We also plot our original y train and our augmented y train to check the distribution. Here we see those y = 1 values created by the GAN.

1. Regression Using XGBoost

XGBoost stands for eXtreme Gradient Boosting. It's an open-source library designed to provide a scalable and efficient implementation of gradient boosting, specifically optimized for speed and performance.

Gradient boosting is an ensemble learning technique that builds a strong predictive model by combining multiple weak learners, typically decision trees. It works by iteratively training new models to correct the errors made by previous models. XGBoost extends this concept by introducing several enhancements to the gradient boosting algorithm, resulting in faster training speed and better performance.

This was by far the best performing and fastest regressor we tried, and we tried all of them (as far as I know). This may be due to that fact the relationships between the features and the target variables are likely to be nonlinear.

We implement a parameter grid and use Random Search to find the best hyperparameter combination that minimizes MAE and print out all relevant training and test metrics. These are our results:

* Training Metrics:
* Mean Squared Error (MSE): 0.4981708359938941
* Mean Absolute Error (MAE): 0.5223132998775026
* R-squared (R2): 0.7125952563154381
* Adjusted R-squared: 0.7120846776270662
* Test Metrics:
* Mean Squared Error (MSE): 0.6302635344359566
* Mean Absolute Error (MAE): 0.5771969851278731
* R-squared (R2): 0.7459854110732913
* Adjusted R-squared: 0.7442103265944184

Our other efforts without data augmentation produced similar MAE and MSE, but much lower R2.

We then plot our residuals graph and here we see the issue with using GANs instead of cGANs. There is a distinct conglomerate of residuals that are far away from the main bulk. These are our y values created by the GAN, which are all equal to 1. Nothing, short of switching to cGAN or other, more complex data augmentation methods, can be done about that now.

Still, the mean of our residuals is -0.01320, which isn’t perfect but it’s close enough to zero to excuse our ugly residuals graph.

1. SHAP

We use SHAP values to visualize the most important features to our XGBregressor for the sake of curiosity and we notice that self\_reference\_min\_shares, keyword\_avg\_score, product\_category and topic\_quality are the most important.

When we plotted the correlation heatmap, the most correlated score was keyword\_avg\_score. This confirms our suspicion that relationships aren’t linear.

1. Predictions

Finally, we generate predictions using our prediction dataset, which has followed all the same preprocessing steps as our model dataset.

We predict using our XGBregressor, undo the log transformation and round to the nearest integer.

We plot the predicted shares variable as a boxplot. The result seems satisfactory, considering that the distribution resembles the shares variable from the model dataset, minus some huge outliers that would have been almost impossible to predict anyways.

We save our predictions to a csv file.