MoReBergers: Social Media Prediction

Mohamed Kleit
mohamed.kleit@umontreal.ca
Université de Montréal

Lucia Eve Berger
lucia.eve.berger@umontreal.ca
Université de Montréal

Alireza Razaghi
alireza.razaghi@umontreal.ca
Université de Montréal

Abstract

In this report, we present the analysis and results obtained as part of a Kaggle competition on social media prediction. We developed a model capable of accurately predicting the number of profile likes based on simulated social media profile information. We followed an iterative process with an embedded feedback-loop between the data preprocessing, model selection and model performance.

Through our process, we developed two models which provided our best performances on the private test set: an ensemble of gradient boosting models and a multi-layer Neural Network. With the former, we obtained a Root Mean Squared Logarithmic Error (RMSLE) of 1.6719 and a RMSLE of 1.6712 on the latter.

1 Introduction

In this competition, we were tasked with predicting the number of profile likes. This can be understood as a supervised regression problem. The train dataset provided included 7500 observations and 24 features (23 features, 1 label). The test dataset includes 2500 observations.

2 Data Cleaning

The first part of any machine learning or data science project is data cleaning. Data cleaning is the process of detecting and correcting (or removing) corrupt or inaccurate records from a record set, table, or database and refers to identifying incomplete, incorrect, inaccurate or irrelevant parts of the data and then replacing, modifying, or deleting the dirty or coarse data.[Wikipedia]

2.1 Addressing Missing values

As seen in Figure 4, we encountered missing values in different features. For each one, we took a different approach. Our approaches can be classified as below:

- **Numerical columns:** For *'Avg Daily Profile Clicks'* and *'Avg Daily Profile Visit Duration in seconds'* columns, we used mean imputation. We used SimpleImputer from sklearn and replaced all the missing values with the mean value of that feature.
- **Categorical columns:**
  - For *'Personal URL'* and *'Profile Cover Image Status'* columns we simply replaced missing values with 0 and also replaced all other values with 1. Thus, at the end we have a column with either 0 or 1.
  - For *'Profile Theme Color'*, *'Profile Text Color'*, *'Profile Page Color'*, *'UTC offset'*, and *'User Time Zone'*, we replaced the missing values with the most frequent value in the column.
  - For the *'Profile Category'* column, first the white spaces were replaced by NaN values and then the Nan values were replaced by *'Unknown'*.

2.2 Removed Features

After carefully investigating all the features we decided to remove few of them. First *'ID'* and *'Username'* columns are irrelevant to our target. The Location column was very messy and also we could extract a sense of location from other columns such as User Time Zone. So we dropped the Location. Finally, we decided to drop the *'Profile Image'* column because of the poor quality of the images and time consuming process of using them.

3 Exploratory Data Analysis

In this part, we explored the dataset and grouped the main characteristics.

3.1 Categorical features

For non numerical features, we examined how many unique values each contained. See Figure 5.
According to Figure 5 for listed features there were a high number of unique values. In order to use categorical features in many models they need to be converted to numerical features and for that we used feature encoding. We decided to use Frequency encoding for above features as we did not want a significant increase in the dimensionality and number of columns. See Figure 6.
For the *'Profile Category'* column, we used One-Hot Encoding as there was only few unique values.

3.2 Data distribution

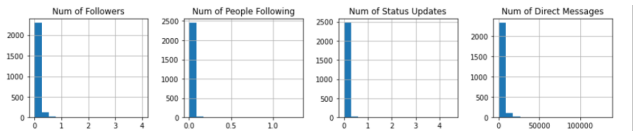
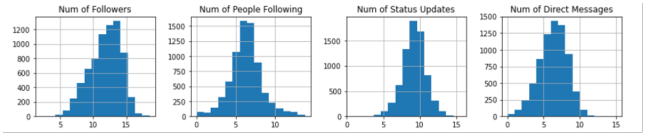


Figure 1. Distribution of Numerical Features

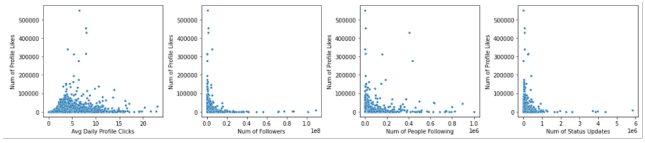
, , ,
2021.

As we can see in Figure 1, our numerical features are skewed and we used a log transform to make them more normal. See Figure 7.



3.3 Outliers

There were many outliers in the dataset. We did an analysis with a boxplot. Given the size of our train dataset (7500 observations), we did not remove all the outliers. We decided to choose 200,000 likes as a threshold and remove all the data points above it. The reason we chose this number was the fact that our data points are well condensed below 200,000 as can be seen in the next picture.



3.4 Scaling

As many machine learning models such as linear models require scaling, we decided to StandardScaler from Sklearn to standardize our data. See Figure 8.

3.5 Feature Selection

We used the Select K Best algorithm with mutual_info_regression for feature selection. Mutual_info_regression estimates mutual information for a continuous target variable. Mutual information (MI) between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency. [Sklearn]
After that we tried different number of features based on above and finally we used top 21 features for our best results.

4 Methodology

As mentioned earlier, we followed an iterative process in which we conceptually embedded a feedback-loop. Data is not a blackbox and the preprocessing can heavily impact the model performance. In this section, we present two different avenues we took in order to develop a model that performs well. A good model is a model that generalizes well on previously unseen data.

4.1 From linear models to gradient boosting

We first started by testing linear models which provided decent results. We then decided to use more complex models - here, gradient boosting. A model that is more complex should be able to capture more relationships between our features and thus lead to better results.

4.1.1 Hyper parameter tuning and cross-validation.

We optimized each model by tuning the hyper parameters using an exhaustive search over a grid of specified values. For each combination of values specified in the grid, we methodically fit the estimator and then evaluated it using 10-fold cross validation (CV). Cross-validation was used as it reduces the variability of the estimate of the test error and tends to not overestimate it too much.

We split the provided data (discarding the images) into training/test splits of ratio 80/20. CV was applied on the training set split by splitting it further into 10 different folds of approximately equal size. The first fold was treated as the validation set, which was used to evaluate the error after the model had been fit on the remaining 9 folds. We repeated this procedure 10 times and the CV results were simply the average of the error calculated at each iteration. This process was executed for every combination of values in the grid, after which we kept the estimator with the combination that led to the best CV results. Finally, we applied that estimator to the test set split in order to obtain an unbiased estimate of our test error.

4.1.2 RMSLE and log-transform.

The metric used as part of the competition to assess the performance of the models was the RMSLE, expressed as follows:

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(p_i + 1) - \log(t_i + 1))^2} \qquad (1)$$

where p_i is the predicted value for observation i and t_i its true value. Considering we are in a regression analysis setting, some of our predictions might be negative. The RMSLE, which uses the natural logarithm, is not defined for negative values. This is not a problem for the neural network (we will discuss this later) as it uses the *ReLU* activation function for the hidden layer and a linear one for the ouput layer. The *ReLU* forces all values to be positive which only leads to positive predictions. However, it constitutes a problem for all the other models for which the output space is not restricted to positive values.

We took care of this problem during training by projecting the values of our target variable onto the log-space as follows:

$$\hat{t}_i = \log(t_i + 1) \qquad (2)$$

where $\hat{t}_i$ is the log-transformed target value. We then train our model using the Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(p_i - t_i)^2} \qquad (3)$$

After the training was completed, we approximately reverse-transformed the predicted values to their original scale by

applying the natural exponent. If we wanted to perform the exact inverse, we would have needed to subtract one to the result of the exponentiation. However, this could still have led to negative predictions for values really close to 0. Moreover, an empirical analysis showed that the error induced by approximating the inverse was minimal.

4.1.3 Linear Regression. The first statistical model we tried was Linear Regression, expressed as follows:

$$y = f(X) + \epsilon \tag{4}$$

where $y \in \mathbf{R^n}$ and $X \in \mathbf{R^{n \times p}}$: y is the vector of predicted values, X is the input with n observations and p independent variables. Linear regression assumes that the true relationship between X and Y is linear and that ϵ is a mean-zero random error term. We can use a residuals plot to verify those assumptions.

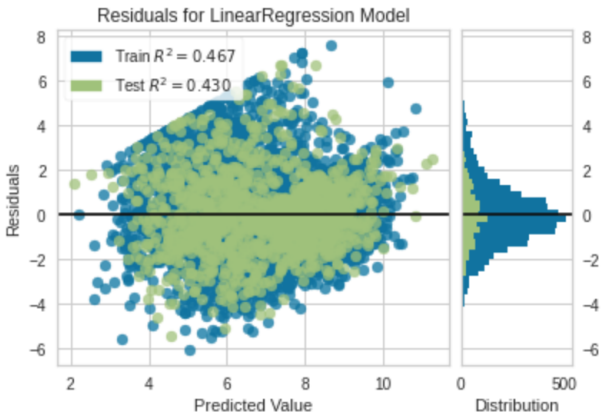


Figure 2. Residuals plot for Linear Regression

As shown in figure 2, the residuals seem to be randomly scattered around the 0 horizontal axis. Furthermore, the right part shows that the distribution of the residuals seems to follow a normal distribution with mean 0. Linear models thus seem appropriate for our training set.

4.1.4 Regularized linear regression. As we assume that X and y follow a linear relationship, exploring regularized linear models seems like the next logical step. Regularization helps in reducing variance (while introducing some bias) as it favours simpler models.

The first regularized model is the **Least Absolute Shrinkage Selector Operator (LASSO)** which assigns a penalty equal to the sum of the absolute values of the coefficients. The particularity of LASSO is that it can shrink some of the coefficients to 0 thus eliminating some features. Another method would be **Ridge** regression. We also assigned some penalty to the weights of the regression model but the penalty corresponds to the sum of the squared magnitude of the coefficients. Consequently, the coefficients can never shrink to 0 and this method does not eliminate any features. Finally, **Elastic Net** is a hybrid of both LASSO and Ridge.

4.1.5 Gradient Boosting. We then explored models that are more flexible than the ones described previously. We thus turn towards gradient boosting. Similarly to bagging, boosting is an ensemble technique in which we use many "weak" learners - decision trees in our case - and combine them using some model averaging technique. A weak learner is simply a decision tree that would perform relatively poorly on its own. However, boosting differs from bagging in the sense that the weak learners are not built independently but sequentially.

Indeed, the intuition is that every subsequent learner learns from the mistakes of the previous ones. Specifically, in the regression analysis setting, we can assume that our goal is to teach a model F that makes predictions of the form $\hat{y} = F(X)$. It does so by minimizing, in an iterative process, a loss function. At every stage $m \in (1, ..., M)$, we have a general estimator F_{m-1}. The aim is to improve this mode F_{m-1} by fitting a new weak learner h_m to the residuals of the previous learners. Therefore, at each stage m, a new estimator is added to the general model in an attempt to correct the errors made by the previous learners.

Intuitively, we can see gradient boosting as using gradient descent to solve an optimization problem with respect to some loss function.

4.1.6 XGBoost and Light Gradient Boosting Machine (LightGBM). We used two specific implementations of gradient boosting: XGBoost and LightGBM. They mainly differ in the way the decision trees are growing (see figure 3). XGBoost applies level-wise tree growth which consists in maintaining balanced trees. On the other hand, LightGBM applies leaf-wise tree growth which splits the leaf that reduces the loss the most. It makes it a more flexible model prone to overfitting.

Figure 3. Difference between level-wise and leaf-wise tree growth [1]

4.1.7 Voting Regression. Finally, building on the idea that combining multiple estimators and performing a model-averaging technique usually reduces overall variance, we combine XGBoost and LightGBM using a Voting Regression model. This model averages the individual predictions from each of the base models to form the final prediction. We present and discuss the results in the "Results and discussion" section.

4.2 Neural Network

We experimented with a deep-learning approach, using variants of a small Neural-Network.

4.2.1 Architecture. The parameters we examined were the number of neurons per layer (the width) and the number of layers (depth). In our design, we calculated the number of neurons as function of input size. As we experimented with the count of features, this formula represented how many inputs to pass between the layers. $((features * constant))$

We experimented with the depth of our network too. With more layers, we realized we were able to capture more aspects and relationships between the data points.

4.2.2 Design and Hyperparameters. We observed high proclivity of the network to overfit. To address this trend, we added two dropout layers and trained on a learning rate schedule. Dropout layers drop a certain percentage of the weights randomly. Effectively, they make the network less sensitive to the specific weights of neurons. This was constructed to decrease memorization between the layers of our network. We trained on a slower learning rate, reducing our model's response to error.

We constructed two structures. These two varied in the number of neurons in their input and hidden layer. The larger of the two is depicted in 11, with a total of 7,700 trainable parameters. The smaller model contained 2400 trainable parameters.

5 Results and discussion

5.1 Linear Models

	Linear Reg.	Lasso	Ridge	ElasticNet
CV RMSLE	1.6311	1.6979	1.6912	1.6919
Public score	1.8725	1.8733	1.8728	1.8728
Private score	1.7408	1.7405	1.7411	1.7411

Table 1. Cross-validation and test results

Table 1 demonstrates that all three models provide very similar results. Lasso, Ridge and ElasticNet did not provide any improvement on Linear Regression. This can be explained in a simple way: linear regression model is too simple for our training data. This means that it did not overfit and our regularized models, whose aim is to reduce overfiting, were not valuable. They optimized regularized models converged towards a simple linear regression model. ElasticNet being a hybrid of Lasso and Ridge, it converged towards the regularized model that gave better results which was Ridge.

5.2 Gradient boosting and voting regression

	XGBoost	LightGBM	Voting Reg.
CV RMSLE	1.5408	1.5413	1.5303
Public score	1.7895	1.7723	1.7674
Private score	1.6785	1.6748	1.6719

Table 2. Cross-validation and test results for ensemble

Table 2 shows that gradient boosting performed better than linear models. XGBoost and LightGBM being more flexible models, they did better capture the inherent relationships between the features. Linear models usually work well in practice but require extensive feature engineering. Our feature engineering led to more than decent scores with linear models. However, considering the nature of gradient boosting algorithms, we reached better scores with the same feature engineering. As we can see in the 2, using a voting regression combining XGBoost and LightGBM did not provide much improvement. Voting regression usually helps in reducing variance. However, since our models did not present high variance at first, the improvement was minimal.

5.3 Note on cross-validation results

As can be seen in table 1 and table 2, the CV RMSLE tends to underestimate the generalization error. This seems to be contradictory with the definition of CV we previously gave - CV error usually overestimates the test error. However, this observation is explained by the fact that our models are cross-validated using the RMSE rather than the RMSLE. Indeed, we log-transform our target values before fitting our models. The RMSE is thus evaluated using the predictions on the log-space which explains why we see lower CV error.

5.4 Neural Network Approach

In validating our Neural Networks, we used cross-validation. We captured the full spread of data and utilize all of our limited data. Our best result on the private dataset was a RMSLE of 1.6712.

However, we recognize shortcomings with this approach. As observed on table 3, our approach underestimated the model's performance. We overestimated the training and validation error. Additionally, we recognize the limited scale of this approach. With a larger dataset or model, the training time would be cumbersome and computationally-intensive.

	5-layer: 7000	5-layer: 1200
CV RMSLE	1.8601	1.8490
Public score	1.80226	1.80696
Private score	1.6712	1.67770

Table 3. Cross-validation and test results for the 5-layer Neural-Network

6 Conclusion

In completing this project, we realized the value of data exploration and feature extraction. While there are different strategies to develop a well-generalizing model, the core element was the data. For example, applying a logarithmic transform on the data increased our performance. We see great value in understanding our input well before passing it to the next stage of a Machine or Deep learning pipeline.

7 Acknowledgments

Shoutout to sklearn and keras!

References

[1] Author keitakurita. 2019. LightGBM and XGBoost Explained: Machine Learning Explained. https://mlexplained.com/2018/01/05/lightgbm-and-xgboost-explained/

A Appendix A

```
df_train.isnull().sum()
```

```
Id                                          0
User Name                                   0
Personal URL                             4244
Profile Cover Image Status                 90
Profile Verification Status                 0
Profile Text Color                         66
Profile Page Color                         78
Profile Theme Color                        72
Is Profile View Size Customized?            0
UTC Offset                                486
Location                                 1461
Location Public Visibility                  0
User Language                               0
Profile Creation Timestamp                  0
User Time Zone                            486
Num of Followers                            0
Num of People Following                     0
Num of Status Updates                       0
Num of Direct Messages                      0
Profile Category                            0
Avg Daily Profile Visit Duration in seconds    77
Avg Daily Profile Clicks                   76
Profile Image                               0
Num of Profile Likes                        0
dtype: int64
```

Figure 4. Addressing Missing Values

```
print("Number of unique values for Profile Text Color", df_train['Profile Text Color'].value_counts().nunique())
print("Number of unique values for Profile Page Color", df_train['Profile Page Color'].value_counts().nunique())
print("Number of unique values for Profile Theme Color", df_train['Profile Theme Color'].value_counts().nunique())
print("Number of unique values for UTC Offset", df_train['UTC Offset'].value_counts().nunique())
print("Number of unique values for User Time Zone", df_train['User Time Zone'].value_counts().nunique())
```

```
Number of unique values for Profile Text Color 39
Number of unique values for Profile Page Color 38
Number of unique values for Profile Theme Color 40
Number of unique values for UTC Offset 26
Number of unique values for User Time Zone 57
```

Figure 5. Categorical Unique Values

```
#Define function for frequency encoding
def encode_frequency_feature(X, feature):
    encode = X.groupby(feature).size() / len(X)
    X[feature] = X[feature].apply(lambda x: encode[x])
    return X
```

Figure 6. Frequency Encoding

```
#Log transform numerical feats (Train)
for i in num_features.columns:
  df_train[i] = np.log1p(df_train[i])
```

Figure 7. Log Function

```
# #Standardization - Test
df_test_stand = df_test.copy()
for i in x_col:

    # fit on training data column
    scale = StandardScaler().fit(df_test_stand[[i]])

    # transform the training data column
    df_test_stand[i] = scale.transform(df_test_stand[[i]])
```

Figure 8. Standardize Segment

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
from sklearn.feature_selection import mutual_info_regression
# define feature selection
fs = SelectKBest(score_func=mutual_info_regression, k='all')
# apply feature selection
X_selected = fs.fit_transform(df_train_stand[x_col], df_train_stand[y_col].values.ravel())
names = df_train_stand[x_col].columns.values[fs.get_support()]
scores = fs.scores_[fs.get_support()]
names_scores = list(zip(names, scores))
ns_df = pd.DataFrame(data = names_scores, columns=['Feat_names', 'F_Scores'])
#Sort the dataframe for better visualization
ns_df_sorted = ns_df.sort_values(['F_Scores', 'Feat_names'], ascending = [False, True])
print(ns_df_sorted)
```

Figure 9. Select K best features

	Feat_names	F_Scores
15	Personal URL	0.243225
13	Num of People Following	0.130305
14	Num of Status Updates	0.125881
0	Avg Daily Profile Clicks	0.087741
12	Num of Followers	0.054449
11	Num of Direct Messages	0.054006
18	Profile Category_unknown	0.048131
10	Location Public Visibility	0.033808
26	UTC Offset	0.032918
28	User Time Zone	0.030993
27	User Language	0.029975
25	Profile Verification Status_Verified	0.024349
20	Profile Creation Timestamp	0.024255
22	Profile Text Color	0.017946
9	Is Profile View Size Customized?	0.015323
16	Profile Category_celebrity	0.012521
24	Profile Verification Status_Pending	0.010634
2	Avg_seconds_binned_(18.03, 20.691]	0.008303
21	Profile Page Color	0.008200
17	Profile Category_government	0.005214
23	Profile Theme Color	0.004883
5	Avg_seconds_binned_(25.48, 28.132]	0.003550
19	Profile Cover Image Status	0.002205
1	Avg_seconds_binned_(14.795, 18.03]	0.000000
3	Avg_seconds_binned_(20.691, 23.205]	0.000000
4	Avg_seconds_binned_(23.205, 25.48]	0.000000
6	Avg_seconds_binned_(28.132, 31.091]	0.000000
7	Avg_seconds_binned_(31.091, 34.595]	0.000000
8	Avg_seconds_binned_(34.595, 48.885]	0.000000

Figure 10. Select K best features output

```
Model: "sequential_22"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_59 (Dense)             (None, 45)                720
_________________________________________________________________
dropout_27 (Dropout)         (None, 45)                0
_________________________________________________________________
dense_60 (Dense)             (None, 150)               6900
_________________________________________________________________
dropout_28 (Dropout)         (None, 150)               0
_________________________________________________________________
dense_61 (Dense)             (None, 1)                 151
=================================================================
Total params: 7,771
Trainable params: 7,771
Non-trainable params: 0
_________________________________________________________________
```

Figure 11. Larger Neural Network