Retweet Prediction Using common Regression Models (Tree-based Models and Neural Network)
Kaggle team : Les hackers de l'extrême

Lao Quentin
École Polytechnique
Palaiseau, France
quentin.lao@polytechnique.edu

Hellegouarch Paul
École Polytechnique
Palaiseau, France
paul.hellegouarch@polytechnique.edu

Vastel Yann
Institut Polytechnique de Paris
Palaiseau, France
yann.vastel@polytechnique.edu

1. Introduction

Making predictions for future outcomes based on real-world problems is one of many modern-day machine learning and artificial intelligence objectives. In this data challenge we will implement a method to predict as good as possible the number of retweets a tweet will get. A data based such as Tweeter is a wealth of information. Therefore, the challenge focuses on the specific dataset of the 2022 French presidential election.

Two types of features are available for this challenge: tweet-related information (such as text or the number of hashtags in the tweet) or user-related information (such as the number of followers of the user).

At first, we spent a certain amount of time on features engineering with the selection and extraction of features before focusing on our model choice and its amelioration (especially by choosing the best parameters). Finally we compared our model with a neural network.

The loss function that we tried to minimize as much as possible is the **Mean Absolute Error**.

2. Features Engineering

Features Engineering is fundamental in Machine Learning as features represent how our models perceive data. This part could definitely improve our models' performance and let us reap better results.

Before focusing on features extraction, we had a quick look at the two data files *train.csv* and *evaluation.csv*. On Excel, with the use of some data filters, we tried to have a first impression on which features were useful to the challenge. We quickly realized that some features had a moderate impact on the mean of the number of retweets while others were extremely impacting this mean.

2.1. Raw features

The raw data provided by the challenge in the two files *train.csv* and *evaluation.csv* contains some basic information related to the tweets. Some of these data can be used as features. For instance, we will be using :

- tweet's features : the number of likes (*favorites_count*) and the date posted (*timestamp*)
- author's data : the number of followers (*followers_count*), the number of friends (*friends_count*), the number of statuses (*statuses_count*) and whether the account is verified (*verified*).

In practice, this piece of information can be extracted directly from Twitter's API with not any additional changes.

2.2. Hand crafted features

Based the raw data provided by the challenge, one may craft additional features that are considered more relevant. Moreover, some of these data are not functional yet as they are not in a numerical form such as the text and the lists of hashtags, mentions and URLs. Our goal is to extract relevant information from the latter. We then generate the following features :

- Number of URLs and hashtags : the simplest way to deal with the lists of URLs and hashtags is to take their length. These features are quite relevant as hashtags increase the visibility of a tweet and URLs usually incite people to share the tweet.

- Some relations with followers, friends and favorites count : we introduce the followers-friends ratio, the product followers-favorites and the product friends-favorites respectively defined by followers/(friends+1), followers*favorites and friends*favorites. The intuition behind this is that if the author has way more followers than his number of friends, then he probably represents an important account of interest for his followers. For instance, a media or a celebrity usually have an enormous number of followers compared to the number of friends and thus is more likely to be retweeted.

- The weekday and the hour : to avoid the issue of category features (that is for instance having a weekday feature equal to 1 for Monday, 2 for Tuesday, and so on), we pass the actual weekday into a sinus function. Thus, we keep the periodic property of weekdays which would not be the case if Monday = 1 and Sunday = 7; that makes the algorithm perceives Monday and Sunday as very distant days. We apply a similar method to define the hour feature.

- 10 dimensions of text embedding : we propose to transform each Twitter text into a 10-dimension vector based on Word2vec from the library Gensim. The way we do that is to first vectorize the tweet texts with TF-IDF in order to calculate the IDF score of each token. Secondly, we aggregate all the tweet texts and pretrain a 10-dimension Word2vec model on it. To calculate the final text embedding vector of a tweet, we average the Word2vec vectors of each word of the tweet's text weighted by its IDF score.

2.3. Other ideas

Besides the previous features, we also tried many other features and transformations that won't be kept in the final model but are still very interesting for future projects.

2.3.1 Additional handcrafted features

An interesting idea was to consider sentiment (positive or negative) and emotion (anger, fear, so on) of tweet texts, based on NRC Word-Emotion Association Lexicon (aka EmoLex)[1]. This lexicon contains thousands of words each labelled with a multi-hot encoding list of emotions. Nevertheless, this feature did not provide us better performance.
We also try another method for the text embedding, Doc2Vec with different embedding sizes 16, 32, 64. For using hashtag, building a graph with each nodes is a tweet and two nodes are connected by an edge if the two tweets have one hashtag in common. To extract features vector from this graph we use DeepWalk method with different embedding size.

[1]Created By: Dr. Saif M. Mohammad, Dr. Peter Turney, https://saifmohammad.com/WebPages/NRC-Emotion-Lexicon.htm

2.4. Features selection

Among the raw data provided by the challenge, we choose not to consider all of them. As a matter of fact, *mentions* are empty list for every tweets and cannot bring any additional information. *TweedID* is useless as well; tweet's IDs are random numbers to identify tweets which could confuse our algorithm.

3. Chosen model : Random Forest

3.1. Model choice

The most used models for this kind of task are tree-based models and neural networks. Among tree-based models, we have the choice between the two common ones: Random Forest and Gradient Boost Regressor. All of these models are provided by the Python library SCIKITLEARN. The way we proceed was first to focus on these tree-based models and tackle the neural network approach later on.

We had to choose between a Random Forest and a Gradient Boost Regressor. The main difference between the two is how trees are built. As random forests build their trees independently, random forests look stronger against overfitting. Indeed, Gradient Boost Regressor builds its trees one by one: each tree is built so as to correct the prediction error of the previous one. Thus, Gradient Boost Regressors look less robust and more sensitive to noise. The random forest model then fits better in our situation with real-world data.

3.2. Hyperparameter tuning

The principal purpose of hyperparameter tuning is to find the right combination of parameters to maximize our algorithm's performance. To evaluate the quality of a set of hyperparameters, we will pass our algorithm through a K-Fold cross-validation test with 3 folds. Another important point when playing with hyperparameters is overfitting. According to what combination we choose, we will more or less carry risks of overfitting. Here are the main hyperparameters in a random forest model we played with :

- *n_estimators*: corresponds to the number of trees in the forest. Random Forest is based on weak individual trees, that is to say, a single tree has a high risk to overfit. The more trees we have, the less likely our algorithm is to overfit. However, having many trees requires high computational power.

- *max_features*: determines how many features each tree is randomly assigned.

Type	Names	Description	Dim
Tweet Raw Features	*favorites_count*, *timestamp*	Number of likes, date posted	2
User Raw Features	*followers_count*, *friends_count*, *statuses_count*, *verified*	Number of followers, number of friends, number of statuses, verified user	4
URLs and hashtags	*#urls*, *#hashtags*	Number of URLs, number of hashtags	2
Followers-Friends-Favorites	*followers_over_friends*, *followers_times_favorites*, *friends_times_favorites*	$fof = \frac{followers}{friends+1}$, $ft1 = followers \times favorites$, $ft2 = friends \times favorites$	3
Weekday and hour	*weekday*, *hour*	$weekday = \sin(\frac{2\pi}{3600\times 24\times 7} timestamp)$, $hour = \sin(\frac{2\pi}{3600\times 24} timestamp)$	2
Text embedding	*text_emb*	10-dimension vector to represent the text	10
TOTAL			**23**

Table 1: Chosen features for our model

- *max_depth*: represents the maximum number of levels in a tree. Having a too-high max depth makes the decision trees more complex, thus they lose capacities of generalizing and increase the risk of overfitting.

- *min_samples_leaf*: is the minimum number of samples required to be at a leaf node. We have to keep this value bigger as possible at least above 1. If a leaf contains few tweets, this might lead to overfitting.

- *min_samples_split*: corresponds to the minimum number of samples required to split an internal node)

Consequently, hyperparameter tuning is about balancing the performance of our algorithm (using K-Fold cross-validation), with our computational capacities and with the risk of overfitting. To find a quite good hyperparameter combination, we will proceed to a random search (available in SCIKITLEARN): one provides a range for each hyperparameter and during an arbitrary number of iterations, a set of hyperparameters is drawn randomly tested with a K-Fold cross-validation. During all our tests, we will use the Mean Absolute Error (MAE) as metric. The algorithm with the best set will be trained on the whole train dataset and saved. We ran a random search on 75 iterations with 3 folds each. Our results are exposed in table 2. Another approach would be doing a grid search, that is to test every possible combination but if we take the ranges in table 2, we get $10 \times 3 \times 10 \times 3 \times 3 = 2700$ possibilities. Despite this approach would give us the best set of hyperparameters, it is impossible to compute it. For instance, it lasts 20 hours for a powerful gaming computer to compute our 75 iterations.

All the result values in table 2 look to respect the overfitting constraints we fix before except the max depth. The random search max depth gave us the value 80 which is actually quite high. We set the max depth to 20 instead. Actually, we sacrifice a little bit of the MAE score to reduce the risk of overfitting. In our experiments, we lose on average 0.1 MAE score when setting the max depth to 20 instead of 80.

Hyperparameter	Ranges	Result
n_estimators	[100, 200, ..., 1000]	500
max_features	['log2', 'sqrt', 1.0]	1.0
max_depth	[10, 20, ..., 100]	80
min_samples_leaf	[1,2,4]	2
min_samples_split	[2,5,10]	2

Table 2: Chosen features for our model

3.3. Results

Our final Random forest model reaches with MAE of 5.6 in cross-validation which is not bad. Actually, our algorithm guesses **relatively** right. Indeed, in most of the cases, the tweets have a low number of retweets which allows our algorithm to guess correctly. However, it looks to "fail" on tweets with high numbers of retweets as we can see by printing out the biggest errors :

```
--> Err : 2473 (pred:18829;answer:21303)
--> Err : 1525 (pred:16489;answer:18015)
--> Err : 987 (pred:9164;answer:8177)
--> Err : 720 (pred:9251;answer:9972)
--> Err : 635 (pred:12197;answer:11562)
```

The top error is indeed a relative error of 11% and occurs only on one tweet. The relative error decreases rapidly at 5% with the 5th biggest absolute error. Because these tweets have high numbers of retweets, they have important weights in the final MAE score.

4. Comparison with a Neural Network

Let's now compare our Random Forest model to a Neural Network. We will first start with a basic Neural Network (a classic Multi-Layer Perceptron known as MLP) and try to make it more complex.

4.1. Classic Multi-Layer Perceptron

4.1.1 Architecture and features adjustment

Let's start our comparison with a classic MLP. We will choose the architecture illustrated in figure 1a. The input layer is the same as with the Random Forest, so has a size of 23. The MLP contains 2 hidden dense layers of size 512 and 128. The output layer is a dense layer of size 1 and is compared to the target. Between each fully connected layer, we placed a ReLU function as activation. We took the MAE as loss function and Adam as optimizer with its default learning rate ($lr = 0.001$). We implemented our MLP with the Python library Tensorflow.

For some Machine Learning models, not doing any transformations on our features may lead to terrible performances. For instance, without scaling the features, a Neural Network needs an enormous amount of epochs to converge. Scaling a feature is to modify all the values so that its distributions become reduced and centred[2].

Another interesting transformation is the log-transformation which is simply applying the function $\log(. + 1)$. We used this transformation on the target *retweets_count* to deal with very sparse tweets with extremely high numbers of retweets.

4.1.2 Results

We train the model for 200 epochs with a batch size of 32 and a validation split of 0.2. We evaluate our model through a 3-Folds cross-validation and get a MAE of 9.3 which is quite bad. Several possibilities to improve the performance of our model: make the neural network more complex by adding new layers or increase our features.

4.2. Neural Network with embedding layers

4.3. Hashtags embedding

To increase the performance of our model, we increase the feature number. A great advantage of Neural Networks is their capability to take high-dimensional vectors as input. We propose to dig deeper into hashtags. Hashtags seem to be quite relevant. We can imagine that some trend hashtags attract more users and increase the probability of being retweeted. To deal with hashtags, we created a special new n dimensional feature that will no enter the neural network by the same entrance as the other features. This new vector will be the input of an embedding layer. The way we proceed to calculate this feature is as follows:

1. We choose an arbitrary integer f
2. Let V_f be the number of unique hashtags present in the whole training dataset that appear at least f times.
3. We create an index dictionary for the vocabulary V_f,

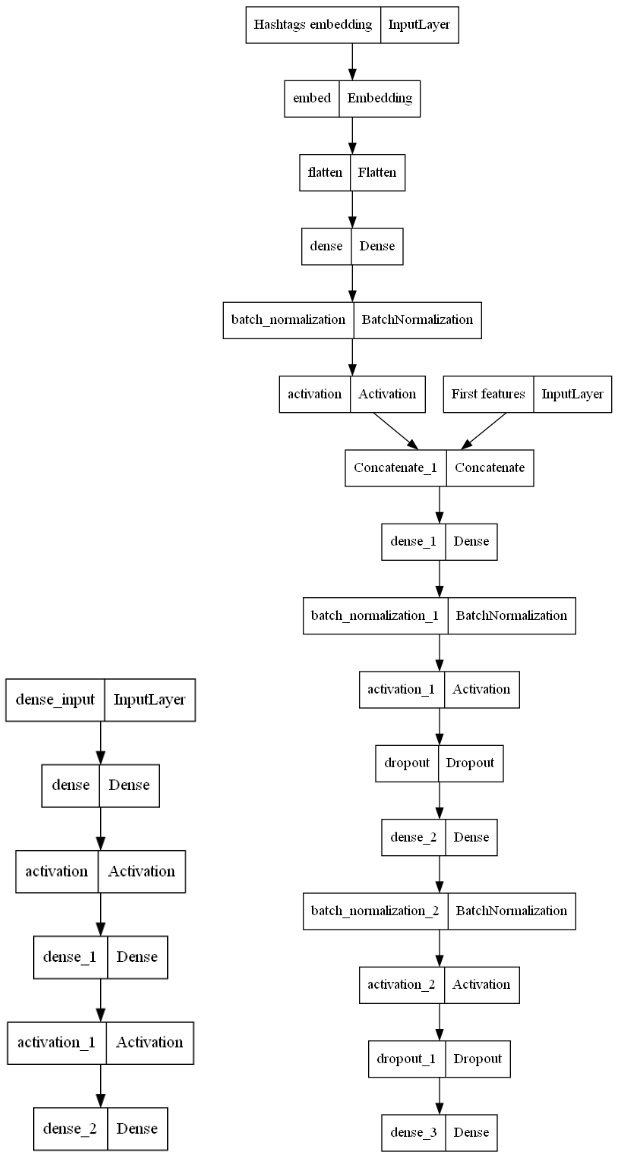
dense_input | InputLayer
dense | Dense
activation | Activation
dense_1 | Dense
activation_1 | Activation
dense_2 | Dense
Hashtags embedding | InputLayer
embed | Embedding
flatten | Flatten
dense | Dense
batch_normalization | BatchNormalization
activation | Activation
First features | InputLayer
Concatenate_1 | Concatenate
dense_1 | Dense
batch_normalization_1 | BatchNormalization
activation_1 | Activation
dropout | Dropout
dense_2 | Dense
batch_normalization_2 | BatchNormalization
activation_2 | Activation
dropout_1 | Dropout
dense_3 | Dense
(a) First MLP (b) With Embedding layer

Figure 1: Two Neural Network Architectures

[2]Let x_k be the feature X of the k-th tweet and $\bar{X}$ and σ_X respectively be the mean and the standard deviation of $(x_k)_k$, then scaling the feature X corresponds to do the transformation $x_k \leftarrow \frac{x_k - \bar{X}}{\sigma_X}$ for each tweet.

that is to say, we encode a unique integer for each word.

4. For each tweet, the new feature is merely the list of length n of indexes encoding for each hashtag in the tweet. If the tweet has less than n hashtags, complete the new feature vector with zeros.

4.4. New architecture

We will keep the same hidden dense layers as the first MLP. The newness of the architecture stands on the embedding layer. We have to distinguish our 23 first features and the new n hashtag features. We choose 10 as the output size of our embedding layer and $n = 4$. Right after the embedding layer, we add a small dense layer of size 20 before concatenating it with the rest of the features. The final architecture is shown in figure 1b.

In order to speed up the training phase (that is to decrease the number of epochs to converge earlier), we can add Batch Normalization right after each fully connected layer. In addition, we add a dropout layer with a rate of 0.8 right after each activation layer. The rate is the probability for a neuron to be ignored, more precisely, its output will give 0 during a temporary period. The purpose of a dropout layer is to reduce overfitting.

4.5. Results

We train our model for 100 epochs with batch size of 32. The MAE score we obtained after a K-Fold cross-validation was about 7.1 which is better than our previous Neural Network but still not enough to reach our Random Forest model. Thus, we can infer that our neural network is not complex enough. An interesting idea would be to include Convolutional and recurrent Neural Network and do the same embedding work for text as hashtags.

5. Conclusion

During this challenge, we had the opportunity to tackle several machine learning and artificial intelligence issues.

At first, we realized the importance of features engineering as it directly impacts the performance of our model. Methods such as *TF-IDF Vectorizer* weren't effective enough for instance and some features were irrelevant. Then, by implementing several methods, we had the opportunity to compare the effectiveness of different machine learning regression approaches. If *Random Forest* was our final choice, it has advantages and disadvantages as well as the other methods that we tried like *Gradient Boosting*.

Comparing this regressor with a *Neural Network* was an interesting different approach. Yet, we didn't had the opportunity to push it to its limits and best performance.