École Polytechnique

INF554

Retweet predictions

Gierski Mathieu
Bohm Maximilien
Mergui Hanna

Professor : Vazirgiannis Michalis

07/12/2022

Abstract

The aim of the project is to predict the number of retweets. The dataset provided contains different types of information. We are provided with unnormalised numbers and texts, either word by word in the hashtags or within a sentence in a text (the tweet itself). Given the high variety of data, we need to apply several treatments.

After normalising numerical data, we vectorised hashtags and texts and applied a simple PCA on them. Then, as words follow each other in the text, we need to be able to extract its meaning. Thus we want to use an RNN network. But first we built an encoder to preprocess sentences and output a matrix for each sentence.

The final network is a duo made of an RNN and a simple NN. We feed the vectorised sentence into the RNN which will output a vector. This vector is then fed into the NN, along with the other data such as, the normalised "mainstream" numerical data and the PCAs of the hashtags and texts. This NN outputs a number, the predicted amout of retweets.

1. Data observation and preprocessing

0.1 Data observation

First of all,when we loaded the dataset, we paid attention to the empty values, and all the possible blank spaces. Then, we looked at the correlation between the features, to have our first impression. We got the following matrix:

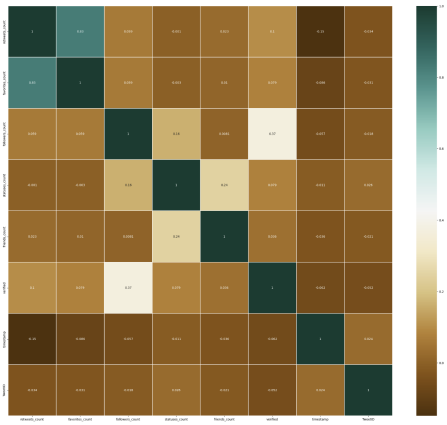


Figure 1: Features correlation

Thus, we noticed that the most important closely linked to the retweets_count is the feature favorites_count. Other features seem less significant.

We also wanted to take into account time. The number provided seemed meaningless so we used the library datetime to extract the month, day and what we call moment, which is the hour and minute. Two other features seemed insignificant for the processing: the URL and the TweetID.

0.2 Data normalisation

After keeping 80% of our dataset for the training and 20% for the validation, we needed to normalize our features. For some of them like "verified", it wasn't necessary because they were already between 0 and 1. We applied our normalisation for the following features: "favorites count","followers count","statuses count", "friends count" and "retweets count" for the training set.

Our data was not evenly spread out. For retweets count, 75% of our tweets had 3 or less retweets, whilst the maximum was at 63674 retweets. This configuration led us to use the log scale, to reduce such a gap.

We then decided to use a min-max normalisation. The idea was to bring all the values into the interval [0,1], while keeping the ratio of the distances between the values.

$$x_{norm} = \frac{(x - x_{min})}{(x_{max} - x_{min})}$$

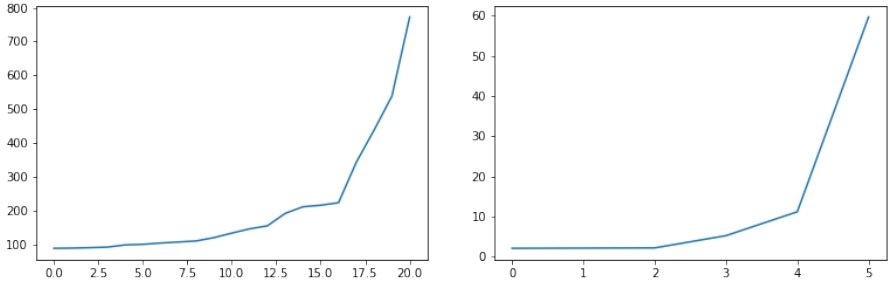
This data treatment is rather convenient for numerical data. But we cannot apply the same for textual features. The following parts are dedicated to explain how to treat texts, word by word or as a sentence.

2. Simple dimensionality reduction methods for texts and hashtags

A simple way to have a look at the outline of the text data is to apply a simple dimensionality reduction method.

The first thing to do is to vectorise words. Taking the whole dataset, we identified the most occurent words. For the hashtags, we selected them if they appeared more than 10 times whilst words with more than 5000 occurences overall were kept. Then, for each tweet, we allocated a vector for its text and hashtag. These vectors had a column for each word/hashtag selected above our threshold on the whole dataset. If that word/hashtag was present in our tweet, we put 1 in the corresponding column, 0 otherwise.

We can now justify the choice of PCA: it allows to detect the main components of variance, hence the main vectors of meaning. Similar methods such as MDS yield similar results. However MDS is much more costly to compute



(a) Eigenvalues for the PCA on the text (b) Eigenvalues for the PCA on hashtags

Figure 2: Implementation of PCA on the text and hashtags (first components are at the end of the plot)

as the Gram matrix would have the same dimensions as the number of tweets. After running some tests, we get the following spreading:

Beware the scales are not the same. One observes that hashtags have more or less 3 principal components, but no more as further components quickly become insignificant. This means that hashtags can be well summed up into a small number of features. The first component can easily be explained by the null hashtag (=no hashtag) which is dominant in the dataset. Hence one should not stop at the first component as it would essentially mean that you sum up to whether you have a hashtag or not.

On the other hand, words have more high principal components and a lot of smaller principal components. This means that one will lose some word dimentionality (essentially meaning) if the number of principal components chosen is too low. However increasing the number of components selected will not bring much meaning: words are hard to reduce into a small number of dimensions without compromising the meaning preservation.

This means that our PCA on words can significantly be impropved. Moreover, PCA does not take into account links between words, it just analyses the presence or not of some words, not the order. We thus need to find a way to analyse words in the text in a smarter way. This will be done through RNN, fed after putting words through a word embedding network.

3. Word embedding

Before feeding the RNN with words, we need to vectorise words. We could simply vectorise words using the letters present in the word: putting columns for the whole alphabet and each letter of the word is a row. We could put a 1 in the column of the corresponding letter. However, this way to proceed does not

take into account the neighbouring words. A smarter word vectorisation should be chosen.

We found the existence of FastText. This word encoder outputs a vector for words. However, its output should be highly dependent on the type of word you feed the network with. Having political words, in french, it was impossibile to find a model that was already trained for such a context. We thus decided to buil an econder ourselved, more relevant to the type of words we were given.

First we need to define what we mean by "word". We decided to cut sentences into patches of 3 letters (with a space being a blank letter). For the sentence "rt présidentielle" we would have: "-rt", "rt-", "t-p", "-pr"... until "le-".

Then, for each patch of letters, we make it into a vector of 30*3, 3 for each of our 3 letters and 30 for the length of our alphabet, 26 + the space + some characters we wanted to add as separate letters (é and è) + any other character (arabic, korean alphabets...).

We then fed this into an encoder/decoder network. The structure varied over our attempts, but ouver all we have the following:

Encoder:
layer1 :3*30 to 100
layer2: 100 to 50

Decoder:
layer1 : 50 to 115
layer2 : 115 to 6*30

The decoder has to be able to find the adjacent patches of 3, that is the one before and the one after. For the first patch of a text such as "-rt" it should thus output "- - -" for before and "-pr" for after to correspond to our tweet.

Within the layers we used ReLU as activation function and for the loss, we cut the output of the decoder into 6 (6 alphabets) and used the cross entropy for each (as we want to find one letter in particular for each one of the 6 alphabet-length output). We trained by batches of 250. Some results are provided below:

```
In [32]: find_output(Emb, "sid")
Out[32]: ('pré', 'ent')
```

Figure 3: Output of the decoder wich the vectorised "sid" is fed into the encoder/decoder network

These results show the maximum number in each one of the 6 alphabets of the decoder (3 alphabets for before, and 3 for after), corresponding to the letter that is most likely to appear according to the network. Other letters may have a probability to appear that is non zero, but in the example, to show the performance of the model, we only take the most likely letter according to the network.

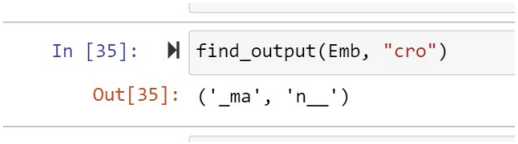


Figure 4: Same as above with "cro"

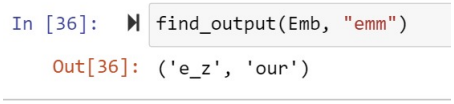


Figure 5: Same as above with "emm"

One can see that the model is very good, espacially for common words. For Zemmour, we see that the model wants to add the letter "e" before the space. Thus it may have been trained to understand that zemmour often comes after a word ending with "e".

Finally, once the model was trained, we took only the vector of length 50 that the encoder outputs and kept it for later for the RNN.

4. The RNN-NN network to predict retweets

Finally, we put all of our treated data into a final network. This one is made of an RNN and an NN.

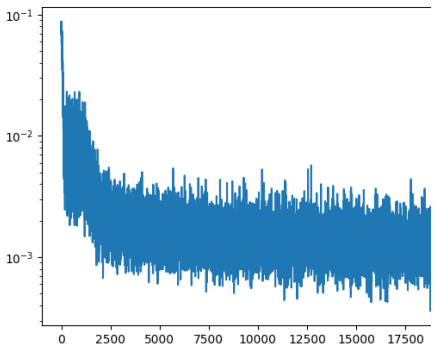
First, the RNN is fed with a concatenated sentence, each line of this matrix is the output of the encoder, of length 50. Then the RNN is made the following way:

RNN:
layer1 (rnn layer): 50 to 100
layer 2: 100 to 100
take h after the activation function at layer2. h is then fed into layer 1 again
layer 3: 100 to 10

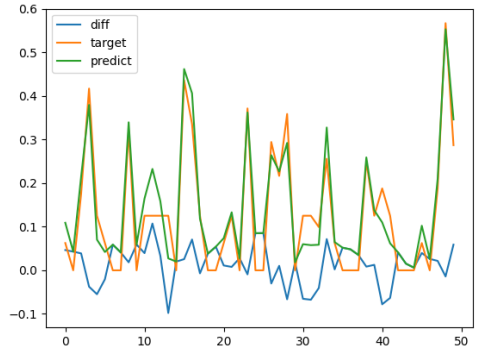
10 is called the "number of feelings", that is a vector of what the RNN can extract from the given sentence.

Finally, we put the mainstream normalised numerical features, the results of the PCAs of the text and the hashtags and the number of feelings into an NN:

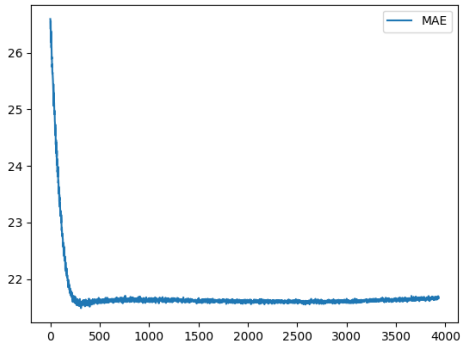
NN:
layer1 : number of dimensions in the input (changes according the our hyper-parameter changes) to 100
layer2 : 100 to 20
layer3 : 20 to 1



(a) Loss plot



(b) Predictions vs target on the training set



(c) MAE on the validation set

Figure 6: Plots of performances of the final network

The output of the NN is the final output: the number of retweets. The activation functions in the hidden layers are LeakyReLU and we used a sigmoid at the end of the RNN and NN. We also used a sigmoid for h. The loss is MSE. We also implemented a dropout of 0.3 to prevent the model from learning by heart.

Through backpropagation, which trains the NN, we also get to train the RNN. Hence we can predict "feelings" through the RNN even though we do not have any output to learn from. These feelings are adapted to the retweet predictions.

Even though our model seemed very solid, we did not get the expected results and our predictions have proven to be inaccurate. We thought about several possibilities:

-We do not believe that we are over fitting. Indeed, seeing the MAE plot, we do not have any significant increase in our loss

-An error in our code. This is possible, even though we hope it is not the case given the time we spent on the project. Word embedding works well, normalisation and denormalisation too, PCAs too... (We had a bug on denormalisation and fixed it but still had low results).

-Maybe our model is not so bad, it just struggles to predict values close to 0 and is accurate for higher predictions. However, the amount of retweets is much more towards the 0 and less towards higher values. Hence we may be good for high values, but our struggle to be accurate for low values shows a bad MAE in the end.

These bad results pushed us to go check other methods to improve our kaggle score.

5. The serious improvements of Boosting

After several week of research, it seemed that our neural network wouldn't have great scores like the other in the kaggle competition. We were confident about our structure, but we decided to search how improve our score. We read several research paper and one got our attention : XGBoost: A Scalable Tree Boosting System written in 2016 by Tianqi Chen and Carlos Guestrin from University of Washington. This paper describe a scalable endto-end tree boosting system called XGBoost, which is used widely by data scientists to achieve state-of-the-art results on many machine learning challenges. Among the 29 challenge winning solutions 3 published at Kaggle's blog during 2015, 17 solutions used XGBoost. This algorithm is very powerful because of its generalization capacity, and its extreme rapidity in the training in comparaison with other machine learning algorithms.

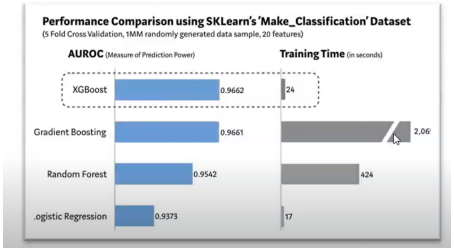


Figure 7: Xgboost brenchmark

The most important factor behind the success of XGBoost is its scalability in all scenarios.The scalability of XGBoost is due to several important systems and algorithmic optimizations.

Xgboost algorithm is a serie of binairy tree. Alone, each tree is not very relevant, but as an association, this serie produces very strong results.

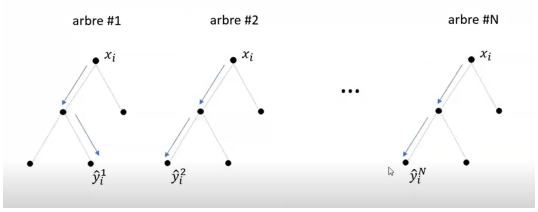


Figure 8: Xgboost binary trees serie

As we can see, each tree received xi as input and gave yi_t as output. At the end, the real prediction for xi is the sum of all the yi_t prediction of each tree.

$$yi = \sum_{t=1}^{n} yi_t$$

So the prediction is improving itself, after each passage in a tree. That's why this algorithm is called "boosting". We can compare this algorithm to the gradient descent, which try to reach the lowest value with a loss function.

Salient features of XGBoost which make it different from other gradient boosting algorithms include: Clever penalization of trees, a proportional shrinking of leaf nodes,newton Boosting,extra randomization parameter,implementation on single, distributed systems and out-of-core computation, automatic feature selection.