INF554 - Team TVRPZ

Romain Fouilland (romain.fouilland@polytechnique.edu)
Philemon Gamet (philemon.gamet@polytechnique.edu)
Jacques Song (jacques.song@polytechnique.edu)

January 8, 2019

In this report, we present our work on the Kaggle competition. The goal is to recover the edges of a citation network where the original edges have been partially erased. To achieve this, the data available is a set of 615 512 citations between 27 770 documents. That amount of data is enough to consider a machine learning based model in order to perform our task. For each document, we have at our disposition a lot of information which will help us with our task: the date and the title of the article, the name(s) of the author(s) and an abstract of a few lines.

The report is split into three parts corresponding to the three main steps we followed. First we show the importance of feature selection to avoid overfitting. We then gain some insight on the performances with some baseline algorithms and draw conclusions which help us tune the final model. We eventually describe the final model and explain how were the parameters chosen.

1 Feature engineering

1.1 Raw features

The dates

For each couple, we can draw interesting information from the publication dates of each article. If each date separately is not valuable, the difference between the two is relevant. It is this time difference that will be used as a feature. When we plot the distribution of this feature, we observe that there are fewer large date differences when the documents cite each other.

The authors

Each document is signed by one or many authors. It is interesting to check the number of author in common between two articles. However, in a huge majority of cases, there is no common author, regardless of the label, that is why we will later use an other feature based on graphs to use the information from the authors.

The titles

The feature we build from the title is quite the same as for the authors. We count the number of words in common between two titles without taking into account irrelevant words (and, the...). This time, the distribution is significantly different depending on the label. We also check if the title of the target documents appears in the abstract of the source document and convert the boolean result into a binary feature.

1.2 Abstract embedding

In this part, we focus on the last part of the data: the abstracts. To use it correctly, we have to be careful not to check directly the overlapping of words between the texts. One way to compare two texts is to represent them in a vector space which preserves some of the characteristics of the language and semantic. We use the TF-IDF representation of a text to perform that embedding.

Once each document is represented as a vector, we need to compare them. In order to do that, we will use the cosine similarity which gives a good measure of how close two vectors are in a high dimension vector space. The distribution of normalized cosine similarity over the training dataset is given by Figure 1 (we can easily see its relevance).

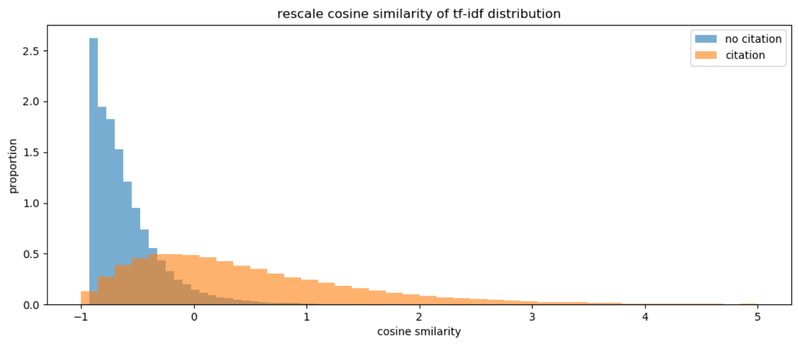


Figure 1: TF-IDF vector cosine similarity (normalized) distribution on the training set.

1.3 Graph based features

This section is dedicated to the extraction of features based on graph analysis. First we have a look at the graph of citations between the documents (in which we want to understanding the edges logic), then we apply the same methodology to a graph whose vertices would be the authors in order to have better features based on the authors.

1.3.1 Based on documents

Let's call $D \in \{0,1\}^{N \times N}$, where N is the number of documents, the matrix of adjacency between the documents. A first feature that we build based on this matrix is the degree in and out of each node, that is, the number of time a document is quoted (degree in) and the number of references a document makes (degree out). We are only interested in the degree in of the target document and the degree out of the source document.

The previous features give us a good idea of the position of each node within the graph but it can be interesting to find out whether two nodes are close to each other or not. Given a pair of two documents (i, j) for which we want to know if i quotes j, we compute different quantities. First we compute the coefficient $(D^2)_{ij}$. This coefficient is the number of article quoted by the document i, which quotes the document j. That means that indirectly the document i already makes a reference to the document j. We then compute $\left(DD^T\right)$ and $\left(D^T D\right)$. The first feature indicates there is a third article quoted by both i and j while the second feature indicates there is an article which quotes both article i and j. The Figure 2 illustrates these situations.

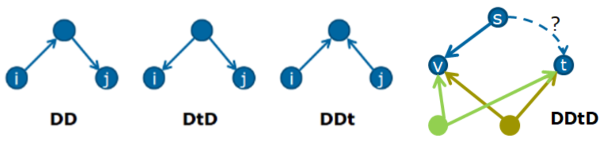


Figure 2: Illustration of the features computed based on the citation graph.

We use the same methodology to compute DDD, $D^T DD$, $DD^T D$ and DDD^T. One of them is more interesting to explain: $DD^T D$. Indeed, this feature points out the fact that t is often quoted by articles citing the articles cited by s. Thus there is more chance for t to be equally cited by s.

1.3.2 Based on authors

We build an oriented graph whose vertices are the authors and whose edges represent citations between authors. These edges are weighted by the number of citations from one author to another. To compare two texts A and B, we will compute how close they are according to this matrix by computing exactly the same features performed previously on matrix D. The only difference with the previous case is that many authors can appear on the same paper. We handle this problem by summing over the relations between each source paper author with each target paper author.

2 Baseline approach

2.1 Support Vector Machine

Our first baseline to test the quality of our features is a SVM, which is a supervised classifier allowing non-linear separation. A good score on a validation set with this classifier indicates that our features are meaningful for our classification problem and succeed in separating the data according to their label. We reached a score up to **96,7%** on the testing set on Kaggle.

2.2 Random Forest

2.2.1 Model tuning

To have another baseline, we decided to implement a random forest. The interest we saw in this method was to be able to have a "human-readable" prediction algorithm and use it to better design our features and our other algorithms.

We tried to find the best random forest by simply performing a random search over many sets of parameters (number of estimators, number of features, the depth, the minimal number of samples to split a node, the minimal number of samples in a leaf, whether we boostrap or not our samples).

The best random forests we found had a small number of Decision Trees (between 50 and 100), used only a few features and were eager to split (minimal numbers around 30 and 10) and didn't bootstrap the samples. It seemed to us such random forests were eager to overfit because they were too precise. However, even if our very best one did (scoring 99.14% on the training set, 97,01% on the testing set and 97,11% on Kaggle), the one we selected required 16 samples to split a node and overfitted way less. This one scored 97.39% on the training set and 96.93% on the testing set.

2.2.2 Features insights

Using these forests, we could gain insight on the features importance.

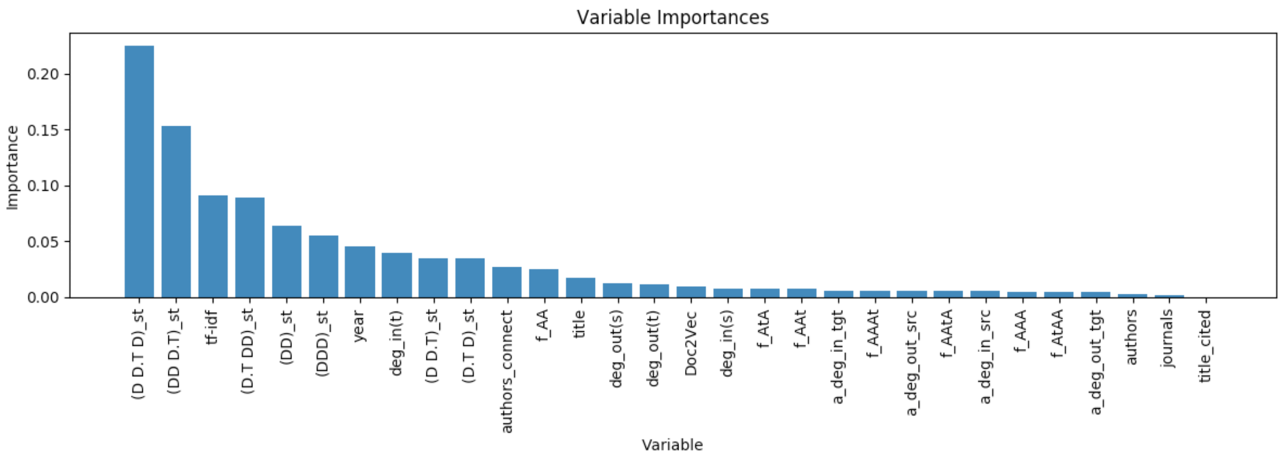


Figure 3: Features importance for our best random forest on the testing set (97.02% and 99.14% on the training set) built with 50 decision trees of 3 features each with a max-depth of 20, at least 20 samples to split and no bootstrap of the data. We can note the impressive importance of the graph of citations (D is its adjacency matrix) which led us to build the same graph for the authors.

To know how these features were actually used, we displayed some interesting decision trees.

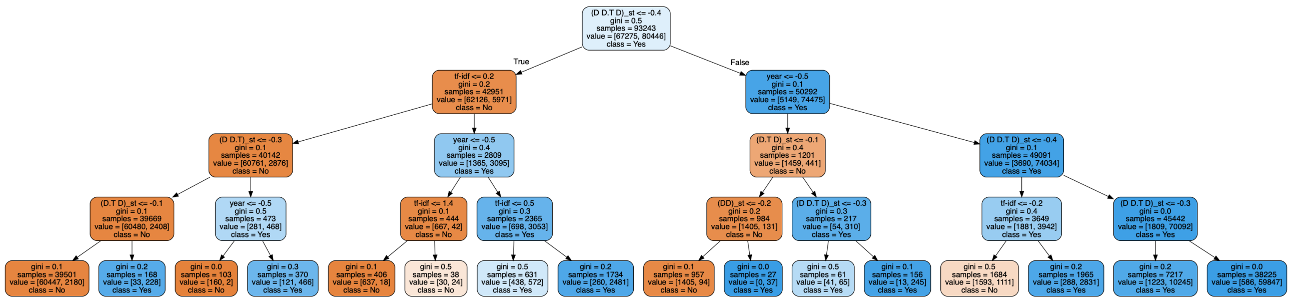

(D D.T D)_st <= -0.4 gini = 0.5 samples = 93243 value = [67275, 80446] class = Yes
True
False
tf-idf <= 0.2 gini = 0.2 samples = 42951 value = [62126, 5971] class = No
year <= -0.5 gini = 0.1 samples = 50292 value = [5149, 74475] class = Yes
(D D.T)_st <= -0.3 gini = 0.1 samples = 40142 value = [60761, 2876] class = No
year <= -0.5 gini = 0.4 samples = 2809 value = [1365, 3095] class = Yes
(D.T D)_st <= -0.1 gini = 0.4 samples = 1201 value = [1459, 441] class = No
(D D.T D)_st <= -0.4 gini = 0.1 samples = 49091 value = [3690, 74034] class = Yes
(D.T D)_st <= -0.1 gini = 0.1 samples = 39669 value = [60480, 2408] class = No
year <= -0.5 gini = 0.5 samples = 473 value = [281, 468] class = Yes
tf-idf <= 1.4 gini = 0.1 samples = 444 value = [667, 42] class = No
tf-idf <= 0.5 gini = 0.3 samples = 2365 value = [698, 3053] class = Yes
(DD)_st <= -0.2 gini = 0.2 samples = 984 value = [1405, 131] class = No
(D D.T D)_st <= -0.3 gini = 0.3 samples = 217 value = [54, 310] class = Yes
tf-idf <= -0.2 gini = 0.4 samples = 3649 value = [1881, 3942] class = Yes
(D D.T D)_st <= -0.3 gini = 0.0 samples = 45442 value = [1809, 70092] class = Yes
gini = 0.1 samples = 39501 value = [60447, 2180] class = No
gini = 0.2 samples = 168 value = [33, 228] class = Yes
gini = 0.0 samples = 103 value = [160, 2] class = No
gini = 0.3 samples = 370 value = [121, 466] class = Yes
gini = 0.1 samples = 406 value = [637, 18] class = No
gini = 0.5 samples = 38 value = [30, 24] class = No
gini = 0.5 samples = 631 value = [438, 572] class = Yes
gini = 0.2 samples = 1734 value = [260, 2481] class = Yes
gini = 0.1 samples = 957 value = [1405, 94] class = No
gini = 0.0 samples = 27 value = [0, 37] class = Yes
gini = 0.5 samples = 61 value = [41, 65] class = Yes
gini = 0.1 samples = 156 value = [13, 245] class = Yes
gini = 0.5 samples = 1684 value = [1593, 1111] class = No
gini = 0.2 samples = 1965 value = [288, 2831] class = Yes
gini = 0.2 samples = 7217 value = [1223, 10245] class = Yes
gini = 0.0 samples = 38225 value = [586, 59847] class = Yes

Figure 4: Display of a decision tree built on all the features with a max-depth of 4, at least 16 samples in the leaves and no bootstrap of the data. It scored 95.62% on the testing set (95.70% for training). It has learned itself that the source is rarely published long before the target article (*year* variable) and uses the abstract (*tf-idf*) and the citation graph (adjacency matrix D) to filter more precisely.

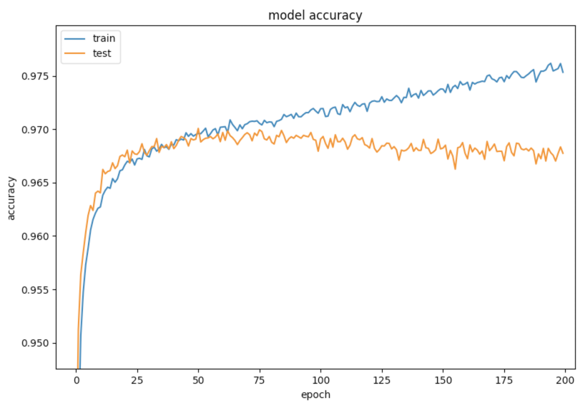
3 Model tuning

3.1 Multilayer Perceptron

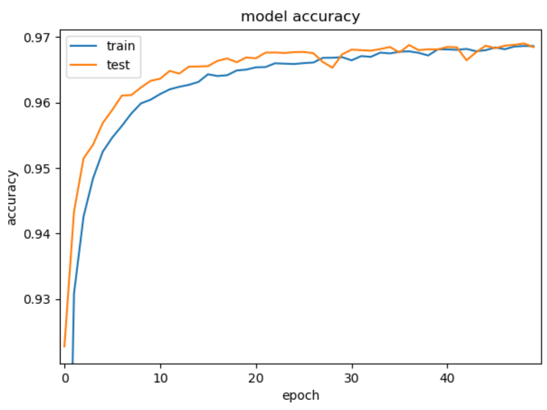
The model we eventually decided to use for this problem is the multilayer perceptron which is the basic feed-forward neural network. The advantage of the multilayer perceptron is its ability to classify non-linearly separable data and more generally, to find complex regressions between the input data and the labels.

To build and tune the model, we used the keras library which gives a simple way to create neural networks by adding sequentially layers into the model. Therefore, we built a simple multilayer perceptron by adding dense layers of various sizes with the final layer being of output dimension 1 to compute the binary value we wanted as predicted label. The hidden layers were given popular activation functions, first hyperbolic tangents and then rectified linear units which are even more widely spread. The output layer was given a sigmoid activation function.

The structure of the neural was tuned empirically by analyzing the historic of the training and plotting the model accuracy and loss graph. We immediately noted the model was overfitting the data since on one hand it performed increasingly good results on the training set while on the other hand, the results on the validation set were decreasing and that happened no matter the network's structure.



(a) overfit



(b) no overfit

Figure 5: Comparison of two learning curves of the MLP for different parameters.

3.2 Avoiding Overfitting

There are two main reasons why the model was overfitting, one concerns the computation of the features, the other, the multilayer perceptron parameters.

3.2.1 Features computing

When we first trained the model, the features we used had been computed on all the training set. We then split the features into a training set and a validation set to feed the neural network. However, the features we computed over the graphs were dependent. For example, when giving the degree in of a certain node, it counted the edges of the validation set. That means that our training set already contained the information about the labels of the validation set. Therefore, the model would perform well on the validation set while we were tuning it by overfitting on the data but would then perform poorly on the Kaggle testing set. This was a major problem since the performances on the validation set we used to tune our model were, in fact, wrong which means the model was wrong.

To remedy this, we simply split the dataset into a set used to compute the graphs and a training set. Only after that, we computed the features for the training set using the data contained in the graph. This way, the results on the validation set gave a good approximation of the results on the testing set. Of course, to compute the predictions of the testing set, we used features that were computed on the whole dataset.

3.2.2 Model parameters

Once we figured out the problem with the features, we were able to really tune the model to reduce overfitting. The first thing we did was introduce l2-regularization into the layers. We then put dropout layers between the hidden layers. Finally, we looked at the model accuracy and model loss plots to find the epoch where the performances were the best.

Using these parameters, we managed to build a model that hardly overfitted. The next challenge was to find balance between avoiding overfitting and not giving the model too much constraints so that it would still learn and improve its performances.