Kaggle Competition - Mail Classification Challenge

Kaggle Team : SPAI
Alexandre Duval, Ibrahim Chiheb, Philibert de Broglie, Sebastien Saubert

26 November 2019

1 Feature Engineering

The **features engineering** part is definitively the task to which we dedicated the most time in this competition but also the most valuable as it provided us tremendous gains on the Kaggle Leader Board during the competition up to the top.
We can split the features engineering part into two tasks:

- **Data validation and cleaning**, which ensures the data provided is correct (not null values nor duplicates) and can be fed to any machine learning algorithm, e.g. numeric values, not text.
- **Features creation**, which extracts and generates features that will benefit our models. This is certainly the most complex part but also the most rewarding one.

1.1 Data validation and cleaning

We identified not-numerical data that can't be processed as such: **date, org, tld and mail type**. Regarding the **date**, as several formats are provided, we decided to parse it with a regular expression to generate datetimes. The issue is that many dates contained a timezone. How can we deal with this information ? Assuming the timezone is related with the sender, it indicates the time gap between the sender and the receiver (no timezone, no gap). From the initial date information we finally have two informations : the **time gap** that we used as it is and the email **received datetime** that we didn't use directly, as it is too much specific by email, but as an input of features creation describe later.

Regarding **mail type**, we apply some transformations to uniform the values : considering only values in lower case and removing any beginning and trailing spaces. As there was only 7 values (including null that we considered as unknown) we apply a one hot encoding of the values transforming this data into 7 binary/dummy columns.

Regarding the **org and tld**, the number of org and tld is above 500 each, so we do not apply the same technique as above. We therefore do not use this information directly. However, we use org and tld to obtain the sender of the e-mail, which will be studied deeply later on.

On this note, we notice that all missing values of the dataset concern the sender. While investigated those, we discovered that 1170 mails with **missing values** (out of 1277) were classified into 'update'. Since there are 583 mails where the sender is not known in the test set, we create an attribute **Null_sender**, which should strongly contribute to helping classify them into 'update'.

1.2 Features creation

Once the validation and cleaning parts completed, we have a dataset that can be ingested into a model. Training can start right away but to obtain better performance, additional features must be created. This what this part is about.

The **process** is iterative: we brainstorm a set of new features, implement it, check if the performance improves, keep only the relevant attributes using Recursive Feature Selection and Feature Importance results, and then repeat the process with new ideas.

Note that we mainly used XGBoost and LightGBM for this task since they are the most powerful algorithms and thus susceptible to yield the best results. This is crucial as they are both boosting methods based on Decision Trees (in this case). This means that we look for variables that help us reduce entropy. And this not only on the whole dataset but also on subsets of the data. Therefore, even some variables whose significance is difficult to capture theoretically could turn out to be useful in some ways. They could contribute to a great information gain in some low nodes of the tree, where only a small subset is regarded. That's why we need to consider and test many possibilities.

The creation of each new feature was first thought from a logical point of view before being applied and assessed in terms of increased performance on the test set. Cross validation (with respect to f1 metric) was first used for that purpose but as model complexity increased, it became very computationally intensive. We therefore took advantage of the reasonable number of submissions allowed to test the relevance of new features directly using the Kaggle score.

Our first dimension was the **date** as any type of information regarding time could help us classify a mail correctly. Maybe more promotional emails were sent in December before Christmas, or maybe forum emails were sent at a specific time on a specific day. We simply need to be creative and check the significance of each new set of time features, to then choose which ones should be kept. Indeed, adding too many irrelevant variables will not benefit our

model, it will increase the computational time of each split and make its interpretation more difficult. Once the date is converted to the right format (datetime), we extract information about year, month, etc; and proceed to multiple tests regarding the significance of the following categorial variables: year, semester, trimester, month, weekday, hours, buckets of hours, end/beginning of a month, month per year, hour per day... After several tests, we concluded that only the year and weekday dummy variables were sufficiently important to our classification task and deserved to be kept. The rest is discarded as it increases significantly the complexity of the model while not improving consequently the performance. Nevertheless, later on, we will couple some of them together with the sender and some other indicators to capture new segmentation possibilities. This will be describe in next sections.

At first, we decided to study the **org and tld** independently but we then noticed that adopting a more aggregate view via the **sender** (coupling of tld and organization) was of greater efficiency. Indeed, looking at the distribution of senders across labels, we noticed two essential facts. Senders whose email is classified into 'social' never send emails classified into 'forum' and conversely. Furthermore, many senders are exclusive to the category 'update' and 'promo'. We therefore included seven corresponding dummy variables, which turned out to be extremely significant! These features are among the most important of our model and contributed to a big jump in performance. Although we first thought these new variables would lead to overfitting, no such thing happened. This is probably because e-mails in test/train belong to the same mailbox and therefore have the same distribution. Classification rules might have been done on the whole set before splitting, which leads these variables to be highly accurate. Going even further, we combined three of these binary categorical variables with existing features (ccs, number of characters...) to be able to differentiate even more senders who send messages classified into several labels.

On the other hand, except for the 'social' and 'forum' classes with the specifics describe previously, this is not the case for the four classes considered together. A same sender can send for exemple update emails and social emails. This is were we decided to provide some comportemental information from each **sender** based on a **time frequency**: What are the emails frequency from a sender on several periods [the total period, on a specific year, month, week number, day, hour, minute]. Indeed, 'update' emails may be send at regular times, like each Monday at 8am. To capture this frequency/behavior for each period, we decided to calculate the mean and the variance over the period for each sender. As the model results were significantly boosted we transposed then these calculations with the **sender** on the following **data** : **chars in body, chars in subject, images, urls, carbon copy**. Thus, the number of features starting to grow over 100 but the model was performing better without too much overfitting. The remaining question was how can we do better? We captured the senders' behaviors but we haven't yet positioned a given email into his sender behavior. Do to so, for each **sender x time** frequency and **sender x data** (chars in body, chars in subject, images, urls, ccs) where the mean and variance were calculated, we substrate the calculated mean with the email value producing around 12 additional features, let's call them **feature gap**. Then for each **feature gap**, we divided them by the sender variance adding again 12 additional features for a total of 24. As indicated, we were using XGBoost and LightGBM for features importance that reveals that mainly month, week number, carbon copy, chars in subject and chars in body mean/variance were important.

Since the way of proceeding to study the sender was highly relevant, we looked for similar correlations among all other original variables. Only **ccs**, **chars in subject** and **chars in body** yielded promising theoretical results. For instance, we noticed that only 'forum' and 'update' mails contain strictly more than 2 email adresses in cc. All 'social' mails don't include cc and all 'promotional' mails present 0 or 1 cc. We further notice that adding a new multivariate dummy variable combining cc and bcc could help us classify a few observations into label 2. Concerning the **number of characters**, we notice the mean differs a lot across labels and we decide to include the difference between a mail's number of characters and the mean number of a label, for each label. This was added late in the model and yielded a small jump in performance (0.96086 to 0.96149).

Finally, we have tried to derive basic features closely related to the original set of attributes of the dataset. For example, we created many **multiplicative dummies** such as Salutation x Designation, image x url, image x chars in body, timezone squared... All could potentially capture some interesting segmentation that would benefit our model. Unfortunately, only a few like timezone squared, Salutation x Designation appeared to do so significantly and yielded a tiny increase in performance (0.96149 to 0.96211).

At the end stage of the competition, we decided to fork the features engineering in dedicated version for XGBoost, LGBM, RandomForest and Knn as some positive features from one model can have negative impact on others. This way, we try too achieve the best scores for each models independently.

2 Models tuning and comparison

To perform well in this competition, choosing the appropriate algorithm is an essential aspect: it has a great impact on the submission score. Overall, we have implemented a wide range of classifiers because it is a great exercise to help us grasp a better understanding of their inner workings. We started from the most basic ones, such as Logistic Regression or k-NN and went on to the most powerful, like RF, NN or XGBoost.

Before delving into the implementation, parameter tuning and performance of each classifier, we would like to emphasise that the feature set utilised differ across models. Indeed, classifiers operate differently and cannot all be treated the same way. We therefore customised the feature set for each model, as mentioned in the previous part. Also, for each learning algorithm, parameter tuning was done using GridSearchCV repeatedly. This tool was imported from the "sklearn" library ("GridSearchCV") and allowed to test many manually inputed combinations of parameters. It then outputted the parameters' value yielding the best cv score with respect to the f1 metric. Note that it's not always the parameters giving the best results that should be chosen, one might prefer a combination of parameters which is in a "zone" where neighbours configurations have good and very close results, in order to avoid overfitting. The latter was mostly regarded as the difference between cross validation score and the Kaggle score (on unseen data). We thus tried to improve the generalisation of our model and consequently to minimise the distance between both scores. Just to mention a way of doing so, we tried to use PCA technique to reduce both noise and computation time but it significantly lowered scores and was aborted.

At first, a simple **Logistic Regression** was implemented. Due to its simplicity, it started with a relatively low score: around 0,74 on both cross val and Kaggle. Tuning it via grid search (L2 reg. weight) and using the final set of features yielded a final score of 0,9211, which is great for such basic algorithm.

We then moved to a **K-NN** algorithm, which gave us a Kaggle score of 0,9257. The hyper parameters that we have optimised are the number of neighbours (K), the neighbours' weight function and the algorithm technique used to compute the nearest neighbours. After computing several tests, K was set to 8. Remember that the higher K is, the more likely you are to avoid overfitting. Furthermore, it was decided to give a greater influence to close neighbours by weighting points by the inverse of their distance.

Thirdly, we tackled the **LDA** algorithm, which seemed too simple to actually maximises separability between classes and thus couldn't improve its score higher than 0,898 on the training set. Thus, we quickly stopped working on it.

A **Neural Network** was also created as it was believed to be very efficient and promising. The neural network was design as a succession of hidden layers of various size (512, 1024, 256, 128, 128, 128, 64). A softmax was set at the end of this network to assign each instance to exactly one class. We are not interested in the probabilities of belonging to each class. For this classifier, the grid search tested the following parameters: learning rate, number of PCA components to keep (in cases where PCA was used), activation function after each hidden layer (elu, relu, leakyRelu), the batch sized and the adam parameters (amsgrad, beta1, beta2). With this parameter tuning method and the final set of features, our neural network managed to improve its kaggle score rapidly from 0,8559 to 0,9376. However, in addition to the big difference between the CV and Kaggle scores, which suggests overfitting, XGBoost and LGBM gave higher results so we stopped working on the NN.

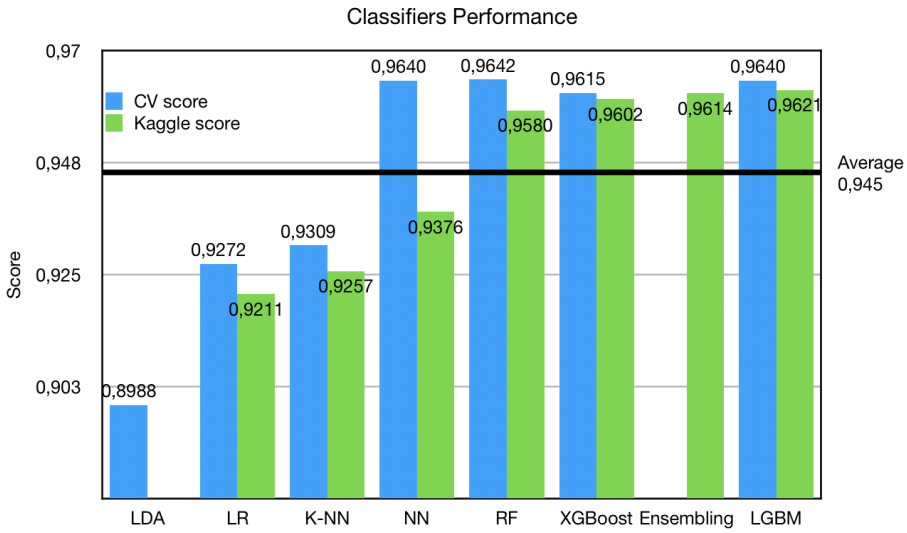
The **Random Forest** gave us a primary cv score on the training set of 0,93. Grid search was then used to optimise the number of trees in the forest (n_estimators) and the maximum depth of the tree. Other hyper-parameters such as the criterion (the function to measure the quality split) were set up manually. Similarly, it was decided to have no weight on classes as there is no reason to favours one, and to use the whole dataset for each tree. Subsampling the data contributes to decreasing the variance even more and hence tackles overfitting. However, overfitting was not so much of a problem here and using the whole data turned out to improve our Kaggle score. Moreover, we started doing some interpretability of the model by for instance, plotting feature importance graphs and doing feature selection. This helps us refine our model and create new relevant features. In the end, all this process allowed our RF to reach a cv score of 0,9642 on the training set and a Kaggle score of 0,9577, having 900 trees and 300 as maximum depth.

XGBoost, which is one the most powerful boosting method, provided great results for this project! It included many variables and therefore needed to have a not too big number of estimators (decision tree) in order to avoid overfitting. The optimal number of trees and their depth was found using GridSearchCV many times and yielded the following result: max depth = 12, n estimators = 300. If you decide to add more trees to improve accuracy on the train set, you start to overfit seriously and the Kaggle score drops significantly. Regarding the other parameters you can tune, such as the learning rate for instance (also quite important), the default value was derived as optimal most of the time. Note that some further explanations about the functioning of XGBoost were given in the feature engineering part.

On a similar note, we implemented **Light GBM**, which is another gradient boosting method extremely similar to XGBoost, also state of the art. It provided our highest Kaggle score 0,9621, with a cv score equal to 0.964. Even though this learning algorithm runs quickly, it has a lot of parameters making the optimisation of the model complex. Once again, the grid search designed to find out the best param-

eter combination between the following ones: learning rate, max depth of the trees, minimum of child per leaf, number of epochs, number of leaves, L1 regularisation weight and the L2 regularisation weight, managed to improve the score to its highest. A particular feature from the LGBM function allowed us to plot the importance of the features for the model which permit to hone our features selection.

At the end of the competition, we tried to **ensemble some of the best classifiers** as a last strategy to improve our score. Different combinations of our best classifiers were performed but unfortunately, none could beat the LGBM alone. As The LGBM performed best on all features set each time, it was choose as our best classifier.



3 Conclusion

We could say that some algorithms seem to be more sensible to overfitting than others, probably because they are too complex (NN, K-NN, RF or XGBoost with 600 trees). Others, too simple, couldn't deal with all the features and therefore didn't manage to perform well (LDA). Overall, we noted that for this project, decisions tree based algorithms performed best (they are the only one above average score) and allowed us to classify e-mails extremely accurately. Finally, bear in mind that the feature engineering part was key to improve our scores all along the competition.