Global Terrorist Attacks Challenge Report

Development of a Prediction Model

Fellowship.AI

Beatrice Vincenzi

April 8, 2018

1 Introduction

The Global Terrorism Database (GTD) is an open-source database including information on terrorism events around the world from 1970 until 2014. Even though many data have been gathered, some portion of the attacks have not been attributed to a particular terrorist group yet. In this report we present a way, based on machine learning, to use some features present in the data set to build a model that can predict what group may have been responsible for a terrorist attack.

The downloaded dataset contains three different excel files: "gtd_70to94_0617dist.xlsx", "gtd_95to12_0617dist.xlsx", "globalterrorismdb_0617dist.xlsx". They report attacks from 1970 to 1994, from 1995 to 2017, and all attacks from 1970 to 2017 respectively. For this project I have chosen randomly to work on the first dataset, so we consider attacks from 1970 to 1994.

This original dataset has 58099 examples and 134 attributes as reported into the file description "Codebook.pdf". However, for the project purpose, we decide to consider all examples and 14 attributes. We take a subset of the all features to have more control on the whole process and because we focus on the learning model technique instead of how feature technique encoding. In particular the chosen features are: `iyear`, `imonth`, `iday`, `extended`, `crit1`, `crit2`, `crit3`, `doubtterr`, `country`, `region`, `attacktype1`, `success`, `weaptype1`, `targtype1`. They are considered as categorical variables. The class attribute is the column called `gname`, representing the group most probably responsible for a terrorist attack.

We model the problem as a multi class classification problem. The original classes were 1978 in total, without considering the `Uknown` label. We discard all the examples with this label because they represent the portion of attack that have not been attributed to a particular terrorist group yet. Thus, at the last stage we could use them to make a prediction. Finally, the starting dataset has 39945 examples and 14 features. We choose to learn the model using the Support Vector Classification algorithm. In particular we adopt the "ovr" (one versus rest) approach to model the multi classification case and we focus on the RBF kernel.

2 Environment description

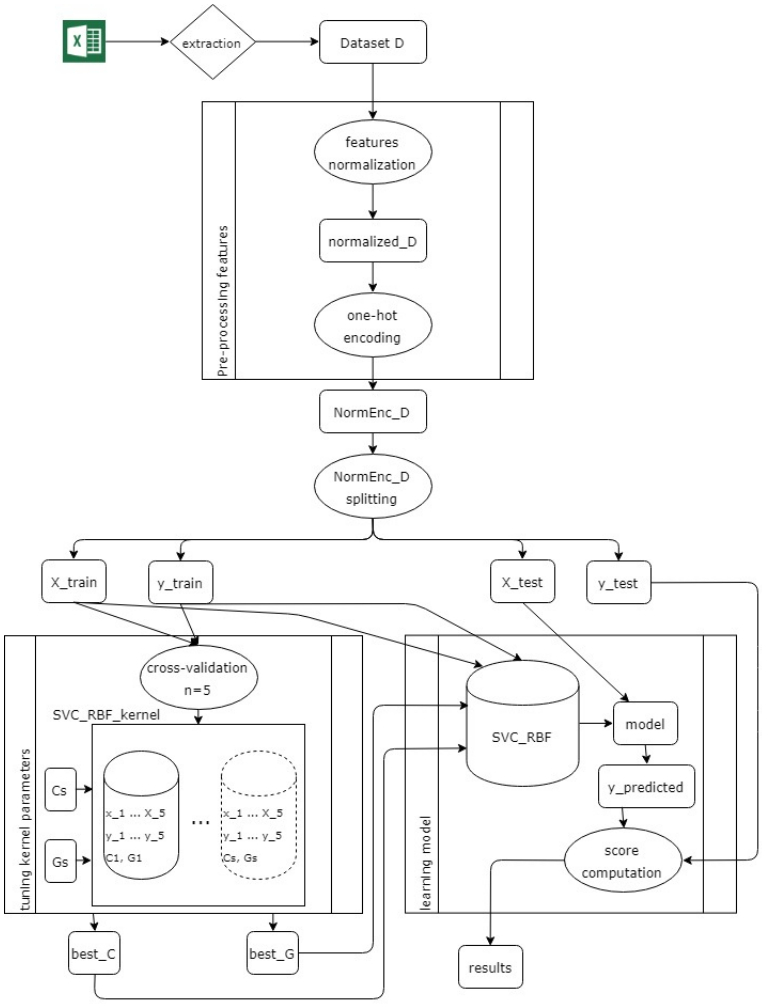
The project is available on `GitHub` at the following link:
`https://github.com/BeatriceVince/GTD_prediction_model`.

We develop the project in `Python 2.7` using `jupyter notebook`.
There are mainly two file concerned the learning model:

1. `main_GTDmodel_prediction.ipynb`;

2. `utils_GTDmodel.py`

The former is a jupyter notebook and contains the main program. The figure below shows in more details its operations. The second file instead contains some useful functions to support the whole process.



The project requires the following packages:

- pandas 0.19.2
- numpy 1.12.1
- scipy 1.0.1
- matplotlib 2.0.0
- scikit-learn 0.19.1
- joblib 0.11
- pathlib 1.0
- imbalanced-learn 0.3.1

3 Pre-processing Features

In the following we refer to the code contained in the files main_GTDmodel_prediction.ipynb and utils_GTDmodel.py. The pre-processing feature operation consists in two steps: *features normalization* and *one-hot encoding*.
In particular, this two operations are implemented in the **normalize_features(dataset)** function and **one_hot_encoding(filename, dataset)** function. Both of them are in the utils_GTDmodel.py file.

Feature Normalization

Using `preprocessing.LabelEncoder` class of *scikit-learn* package[7], we implement the feature normalization. We consider each dataset attribute as a categorical variable. However, at the beginning all attributes are integer numbers, but the values are not consecutive because we have chosen attacks from 1970 to 1994 and they contain a subset of possible values. For instance, as reported below, the first feature `iyear` has 24 possible values, but the actual values associated to each example are between 1970 and 1994, the 1993 year is missing. Thus, thank to that feature normalization we can cast the domain values and associate new consecutive values for each attribute and the one-hot encoding result will be more efficient.

feature: `iyear`
max value: 1994
min value: 1970
num values: 24
all possible values:
[1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1994]
new array values: [0 0 0 ..., 23 23 23]

One-Hot Encoding

We apply the one-hot encoding to each dataset feature. We use `preprocessing.OneHotEncoder` class of *scikit-learn* package[8]. Each categorical (discrete) feature with n possible values is encoded into n binary feature. Thus, the output will be a sparse matrix where each column corresponds to one possible value of one feature. The output of the first training example is reported below. The final dataset is 37119 examples and 268 binary features.

4 Model Selection

As described in the introduction section, we implement the SVC algorithm to learn a prediction model. In this section we refer to the code contained in both of our files.
Precisely, we created **svc_param_selection(filename_tuning, Cs, gammas, class_weight, n_folds, X_train, y_train)** function in the utils_GTDmodel.py file and implemented the learning model in the main notebook.

We search the best parameters (C and $gamma$) for the RBF kernel through cross–validation. We apply 5 folds for the cross validation and we use $\frac{4}{5}$ of the original dataset for that. The remaining $\frac{1}{5}$ portion will be used in the final testing phase. Also the best parameter searching is implemented in the function cited above.
It uses the `model_selection.GridSearchCV` class and we test two different values for the $gamma$ parameter (0.001 and 0.0001) and four values for the C parameter (1, 10, 100, 1000). Thus, in total we test eight different pairs and the metric used to select the best model is F1_micro in multi class case.

After that we take the best parameters and we use them to re–build the model. In the training phase we consider $\frac{4}{5}$ of the data, the same used before in the cross–validation. The output will be the final model.

Finally we re–test it with the remaining $\frac{1}{5}$ data and we compute different validation parameters, such as precision, recall and F1 micro both for each class and their weighted average to get a final single index to evaluate the model.

Different files are saved throughout the running. We list them below with a short description.

- *data_70to94_v.npz*: the compressed and filtered dataset at version v (the first right version after few trials);
- *enc_70to94_v.npz*: the dataset after one–hot encoding at version v (the first right version after few trials);
- *tuning_cv_weight_70to94_v.npz*: results after the parameter tuning, *weight* indicates the type of weight related to each label. It can be of four different types (*noWeight*, *balanced*, *computedWeight* and *resampling*). The saved results are the best C, the best gamma and an array of F1_score for each cross–validation. We can access to them through the keys *C*, *gamma* and *mean_test_score* respectively;
- *modelSVC_rbf_Cbestgbest_weight_70to94_v.pkl*: final model after learning. It is built using *Cbest* and *gbest* parameters, the type *weight* is relative to each class label at the version v considered;
- *scores_modelSVC_rbf_Cbestgbest_weight_70to94_v.npz*: the final file reports different values and arrays as the test results. Being more precise, we can find three values: precision, recall and F1 micro accessible through the following keys: *precision_micro*, *recall_micro* and *f1_micro* respectively. We have also the same validation parameters computed for each label. We can access to their values through the keys *precision_labels*, *recall_labels* and *f1_labels*. In addition, we have other two arrays, which represent the list of labels (*labels_unique*) and the label occurrence in the test set (*labels_support*).

5 Class–Weight Implementation

After the first model implementation, setting the same weight for each class of the target set (class_weight= "*noWeight*"), we analyse better the original dataset trying to improve the current F1 score which is around 0.77. For this reason, we plotted the label distribution and we see that the dataset is very imbalanced. It means that few classes are represented by a lot of examples and a lot of classes have just few examples as we can see in the Figure 1. Thus, the classes are not represented equally.

The presence of many different rare events (classes containing few examples) is a common machine learning issue, because even though they are important to be spotted by the final prediction model, they are hard to be learnt in the training process. The importance of this is, for instance, in applications to fraudulent transactions in banking processes, in the identification of rare diseases and also in the identification of rare terrorist attacks.

For this reason we decided to test three different approaches based on solving such imbalanced class problem, trying to improve the accuracy score. These three techniques are described in the following three subsections.

Class–weight = "*balanced*"

In the first approach we set the *class_weight* parameter of the *SVC* function equal to *"balanced"*. As reported in the documentation [9], this means we give a different weight to each class inversely proportional to class frequencies:

$$w_y = n\_samples/(n\_classes \cdot np.bincount(y))$$

where w_y is the weight associated to the target class y, $n\_samples$ are the total number of example and $n\_classes$ is the number of classes. The function $np.bincount(y))$ is a *numpy* function and it counts the number of y occurrences.

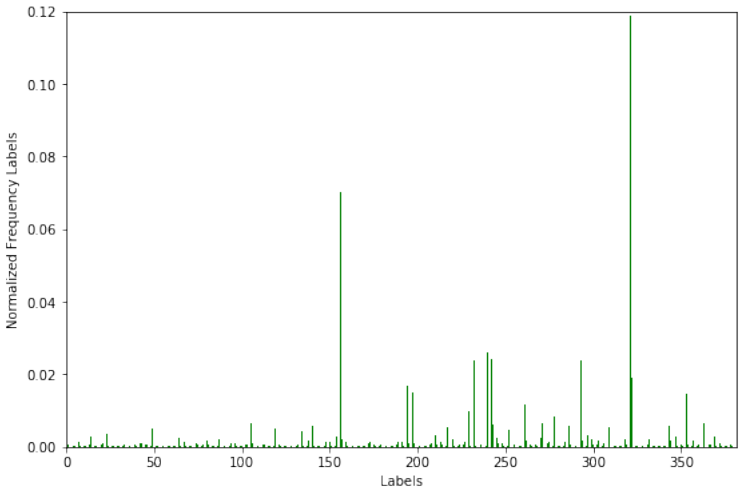


Figure 1: Target Labels Distribution

Class–weight = "*computedWeight*"

Secondly, we try to define an original weight to associate to each example in the same class $\hat{l}$. Indeed examples can be seen as points in the n_f-dimensional feature space (in our case $n_f = 268$ after encoding), and in order to find the best parameters describing the hyperplane in the transformed feature space we can think to give different weights to different points basing on the class they belong.

Example i in class $\hat{l}$ is weighted by

$$c_i^{\hat{l}} = \frac{\#p^{\hat{l}}}{\#_{tot}} \,, \tag{1}$$

where

$$\#p^{\hat{l}} = \sum_{l=1}^{l_{max}} f^{\hat{l}}(p^l) \,, \tag{2}$$

$\#_{tot}$ is the total number of classes and

$$f^{\hat{l}}(p^l) = \begin{cases} 1 & \text{if } p^{\hat{l}} - \delta \leq p^l < p^{\hat{l}} + \delta_p \\ 0 & otherwise \end{cases} \tag{3}$$

is a function which returns 1 (and thus considered in the sum) if the number of examples associate to a class l, p^l, is inside a certain range defined by the parameter δ.

As it is evident the weight defined in that way could be useful for our dataset because we have many (more or less) equally under represented classes. The weight we associate to each example in a class is function of how many classes are represented equivalently (in a range defined by δ).

Nevertheless, the measure defined in that way is not a priori applicable to any kind of data set where rare classes are present, but just to those data sets where there are many rare classes more or less equally under represented. Indeed, for instance, let us think to the case where in the data set there is just one rare class, and a bigger number of typical classes, in this case the rare class would not be better considered by the weight we define above because it is the only one, ending up with values of c_i^l very low.

In conclusion, we expect this measure behaving well in case the data set is rich of rare classes.

Class–weight = "*resampling*"

Finally, we consider a new approach based on re–sampling techniques. In particular we try creating a new balanced class distribution applying ensemble learning strategies designed especially for this purpose. Some of these concern to under–sampling dataset, others to over–sampling and more to a combination of them.

The former aims to balance class distribution by deleting examples of the most represented classes (majority classes). The second strategy, instead, creates or duplicates examples from the minority classes, those having very few examples. The latter applies a combination of the two strategies.

For our purposes, we use the `imbalanced-learn` [6], [3] that implements all strategies. In particular, we focus on *SMOTEENN* class, which performs over-sampling using SMOTE first and under-sampling through *ENN*. More theoretical details are available in [1] and [5].

However, we meet an issue in running it and we could not carry out this approach. The issue is a memory error, happening during the execution. The error stack is reported in *memoryError_resamplingDataset.txt*. I have attempted to fix the problem without success. The cause seems the little available RAM, so we run the program on a system with more RAM, but the error persists. Some on-line reports [2], [4] say that the RAM error is due to the fact that the method requires to save the data set (if big) as a sparse matrix.

6 Discussion of the Results

The code for graphs creation is available in plotting_tunining_results.ipynb and plotting_scores.ipynb notebooks.

The Figure 2 represents the results for all possible combination of C and *gamma* values by weight type. As we can noticed the best parameters pair is $C = 1000$ and $gamma = 0.001$ for all different weight. More precisely, The F1_micro score is 0.76, 0.69, 0.60 for "*noWeight*", "*balanced*" and "*computedWeight*" weight type respectively. We obtain the best score setting weight classes equal to "*noWeight*"0.

Unexpectedly, the further approaches proposed does not lead to any improvements in the model accuracy.

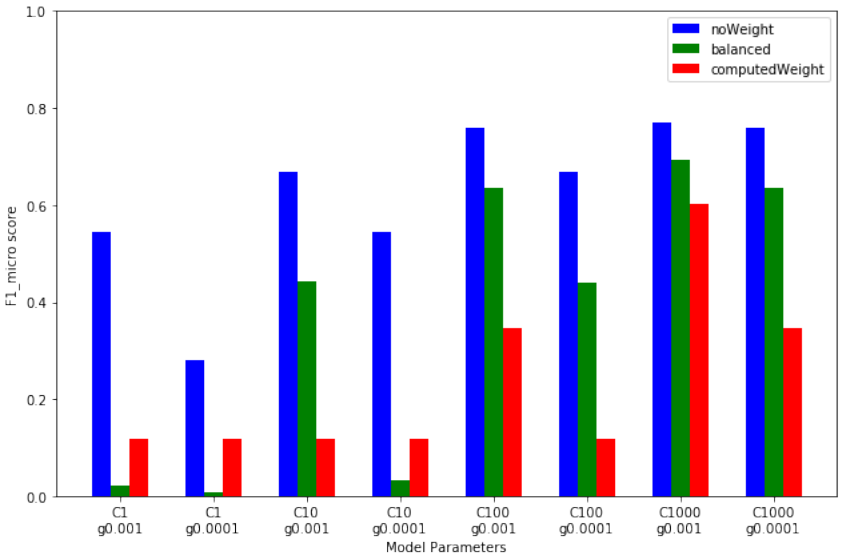


Figure 2: Tuning Parameters Results

The Figure 3 shows the F1_micro score after the learning model with the best C and *gamma* parameters.

The histogram describes the results for each weight type and how we can imagine they reflect those obtained during the parameters tuning. In particular, we have 0.78 for "*noWeight*", 0.70 for "*balance*" and finally 0.61 for "*computedWeight*" type.

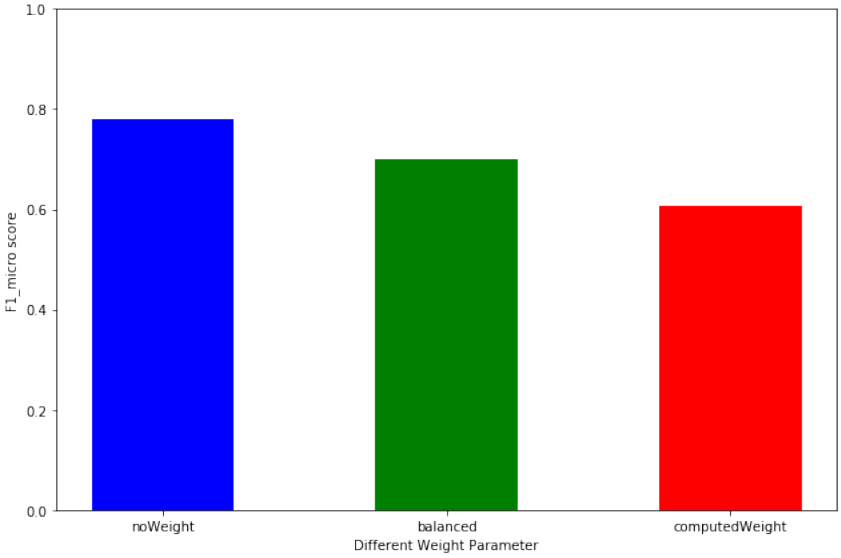


Figure 3: Testing results with C and g fixed

7 Further Improvements

Different ways could be considered to improve the prediction model. First of all we could carry out a deep study to understand why the approaches we studied, which consider a different weight for each example, do not yield better results.

We could also try to tune the δ parameter in the "*computedWeight*" approach, in order to find the best δ value that maximize the accuracy of the prediction model.

After that it could be interesting to do some other researches to fix the memory error in the re-sampling approach and re-test it.

Finally, it would be also interesting to investigate new approaches to include other features in the learning model. We could both including those features discarded previously and creating new attribute instances. For instance we could take `weaptype1`, `subweaptype1`, `weaptype2`, `subweaptype2`, `weaptype3`, `subweaptype3`, `weaptype4`, `subweaptype4` attributes and synthesize a new feature that considers the weapon information comes from all of them.

Finally, always in order to optimize the accuracy of the prediction model, we could also define a new weight to associate to each example in the class $\hat{l}$. Easier than the approaches defined above, example i weight in class $\hat{l}$ could be inversely proportional to the number of examples in the considered class $\#_{\hat{l}}$

$$w_i^{\hat{l}} = \frac{1}{\#_{\hat{l}}} \, . \tag{4}$$

References

[1] G. E. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter*, 6(1):20–29, 2004.

[2] G. L. et al. scikit-learn-contrib/imbalanced-learn error report. `https://github.com/scikit-learn-contrib/imbalanced-learn/issues/300`.

[3] G. L. et al. scikit-learn-contrib/imbalanced-learn repository. `https://github.com/scikit-learn-contrib/imbalanced-learn`.

[4] D. O. G. Lemaitre, F. Nogueira and C. Aridas. Imbalanced learn package. `http://contrib.scikit-learn.org/imbalanced-learn/stable/introduction.html`.

[5] T. R. Hoens and N. V. Chawla. Imbalanced datasets: from sampling to classifiers. *Imbalanced Learning: Foundations, Algorithms, and Applications*, pages 43–59, 2013.

[6] G. Lemaître, F. Nogueira, and C. K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[7] Scikit-learn. Label encoder documentation. `http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html`.

[8] Scikit-learn. One hot encoder documentation. `http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html`.

[9] Scikit-learn. Svc documentation. `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`.