Kaggle Contest Entry

Team **Flechazo**

Project Report

Xavier Ignacio González (**xig2000**)
Diego Miguel Llarrull (**dml2189**)
Gary Sztajnman (**ggs2121**)

COMS 4721
Machine Learning for Data Science
Columbia University
MS In Data Science
May 4th, 2016

Contents

Chapter 1

Introduction

In the following report, we will describe the methodology we used to develop the solution presented by team **Flechazo** (arrow shot) during the in-class Kaggle competition for *COMS 4721 Spring 2016*. We developed an algorithm based on ensemble methods that enabled us to reach a prediction rate of 95.156% on the public leaderboard after 13 attempts. On the private leaderboard, the prediction rate of our model was reduced to 95.023%, although our position in the leaderboard went up by 2. Compared to the rest of the class, we reached the 15^{th} position, with a 0.4% prediction rate difference compared to the winning team.

The tasks for the project were assigned in the following way: since Xavier has significant experience in Machine Learning projects, he initially ran the data preprocessing tasks that historically yielded good results to him. After agreeing upon the preprocessing, each member ran different models in their machines using the processed dataset, in order to improve processing times. Only those models which actually improved on our existing cv-rate were submitted to Kaggle, so as to keep the number of submissions as low as possible. This report is co-authored by each of the three members, by firstly agreeing on a core structure, then splitting chapter writing and finally peer-reviewing each of the assigned sections.

Chapter 2

Data Preprocessing

We started by analyzing the feature set of the dataset. It is a collection of numerical, categorical and binary variables. Given that the algorithms that we wanted to use only considered numerical or binary variables, we decided to transform categorical data to dummy variables in both the data and quiz file.

We noticed that some categories had a very low frequency. Hence, we grouped these categories in a new category called *other*. We chose so in order to simplify our feature map and to avoid outliers. (first hyperparameter min number of obs in a particular category – What?)

To further simplify and optimize our dataset we transformed variables that took only two values to binary. One such example is the variable called *Variable*.

We anticipated that the decision boundary may be non-linear in x but could be linear in $\phi(x)$. To manage this effect, we tried two different feature maps.

2.1 Feature Maps

We tried three feature maps, of which we only kept 2 in the final implementation

2.1.1 f2 complete map

A complete *order-2* feature map, considering all degree-2 and degree-1 terms. Let $\phi : \mathbb{R}^d \to \mathbb{R}^D$ be the expanded feature mapping given by

$$\phi(x) := (x_1, x_2, ..., x_d, x_{2,1}, x_{2,2}, ..., x_{2,d}, x_1x_2, x_1x_3, ..., x_1x_d, ..., x_{d-1}x_d)$$

We initially tried this feature map on our algorithms but, unfortunately, it did not yield any improvements which, added to its significant computational cost and the complexity it added to the model, was reason enough for us to discard it.

2.1.2 f2 strict map

This feature map converts a feature set into a *strictly-order-2* map. That is, only interaction terms between different features are considered. Let $\phi : \mathbb{R}^d \to \mathbb{R}^D$ be the expanded feature mapping given by

$$\phi(x) := (x_{1,2}, x_{1,3}, ..., x_{1,d}; x_{2,1}, x_{2,3}, ..., x_{2,d}, ..., x_{d-1,d})$$

2.1.3 fix2 map

This map replaces low-frequency occurrences of the value $'2'$ in categorical features taking values $'0', '1'$ and $'2'$, and adds an extra column stating whether a $'2'$ value was replaced or not. This is done to improve the efficiency of the models by replacing one ternary variable with two binary variables.

2.2 Implementation of the Feature Maps

- Initially, we fitted a *TreeBagger* ensemble model once on the dataset in order to detect the predictors with the highest importance. Then, we extended the dataset is extended by applying the *f2 strict* map to these predictors. However, since this approach yielded no performance improvements, we discarded this option.
- After discarding the previous step, we extended the dataset again by applying the *f2 strict* map applied on all numerical variables.
- Finally, we applied the *fix2* map on the categorical variables where it applied.

Chapter 3

Models considered

To address this general classification problems, the recipe is to try a diverse variety of models in order to get suitable candidates. Several classification algorithms from the Machine Learning Toolbox were tested. Here are some brief comments about the result.

Linear Models: reusing some functions from previous homeworks we evaluated a Logistic Regression Model. We used the function `gmlfit` with the fitting function `logit` to shape a binomial distribution. The best results obtained for this model were around 7-8% error.

Discriminant Models: Reusing also some functions from previous homeworks we evaluated a Linear Discriminant Model and a Quadratic Discriminant Model. We implemented the function `fitcdiscr` with discriminant type as `linear` and `pseudoQuadratic` respectively. Again the resulting error ranged from 7-8%.

SVM Models: We tried a Support Vector Machine Model with polynomial kernel order 2. This is under the assumption that NLP classification problems are best fitted with the combination of occurrences of two words in a particular text. We implemented the function `fitcsvm` with kernel type as `polynomial` and `PolynomialOrder` equal to 2. The resulting error ranged from 6.5-7%.

General Boosting Models: The Matlab Machine Learning Toolbox offers a wide number of boosting algorithms that we evaluated: `AdaBoostM1`, `LogitBoost`, `GentleBoost`, `RobustBoost`. For our binary classification purposes, the best results were obtained with `AdaBoostM1`, with errors arround 6-7%, followed closely by `LogitBoost`. The other algorithms did not perform well with errors around 9-10%. In general, this group of algorithms returned acceptable results although the time to train them, when the number of features is in the order of thousands, made them impossible to use in practice.

The best results were obtained with a `TreeBagger`, the MATLAB version of Random Forests. Its implementation is explained in next chapter.

Chapter 4

Overview of the TreeBagger (Random Forest)

Random forests is an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. In this application the algorithm was used as a classifier. Random decision forests correct for decision trees' habit of overfitting to their training set. The algorithm for inducing Breiman's random forest was developed by Leo Breiman [Breiman, 2001].

TreeBagger is the matlab implementation of the Random Forest algorithm. Bagging stands for bootstrap aggregation. Every tree in the ensemble is grown on an independently drawn bootstrap replica of input data. Observations not included in this replica are "out of bag" for this tree. To compute prediction of an ensemble of trees for unseen data, TreeBagger takes an average of predictions from individual trees. To estimate the prediction error of the bagged ensemble, you can compute predictions for each tree on its out-of-bag observations, average these predictions over the entire ensemble for each observation and then compare the predicted out-of-bag response with the true value at this observation [MathWorks, 2016].

TreeBagger relies on the Classification Tree functionality for growing individual trees. In particular, Classification Tree accepts the number of features selected at random for each decision split as an optional input argument.

The most significant parameters the algorithm takes and their optimal values are listed as following.

MinLeafSize: Minimum number of observations per tree leaf. We tested these values: 1, 2, 5, 10. The optimal value obtained was 5.

NumTrees: Scalar value equal to the number of decision. This parameters counts as the number of iterations that the algorithm performs. In theory, and verified with the results, both training and testing error does not increase with the number of iterations as the overfitting is avoided. The values tested were 5-50-100-150-200-250-300-350-500. The best results were obtained with the highest numbers of trees.

NumPredictorsToSample: Number of predictor or feature variables to select at random for each decision split. This is a very relevant parameters to tune. The value tested range from 10 to 200. The optimal value obtained was at 100.

Chapter 5

Model Selection

5.1 Model selection

We tried different families of classifier models. As an initial experiment, we simply divided the dataset into training and testing in a partition of 40%-60% and we evaluated all models commented in the previous chapter. Then, we selected some candidate models and we performed cross-validation reusing the code that we did for HW2 to tune the hyperparameters. The models with best results were the `AdaBoostM1` with 6%, the `SVM` 6.5%, and `TreeBagger` with around 5%, considering a trade-off between performance and running time. Then, with the tuning, the `TreeBagger` classifier error was significantly lower that the other algorithms.

5.2 Model hyperparameter selection

The most significant hyperparameters relative to the classification error of the `TreeBagger` were:

iter which in this case stands for the number of trees used for the random forests.

npred which is the number of predictor or feature variables to select at random for each decision split, as described in the previous section.

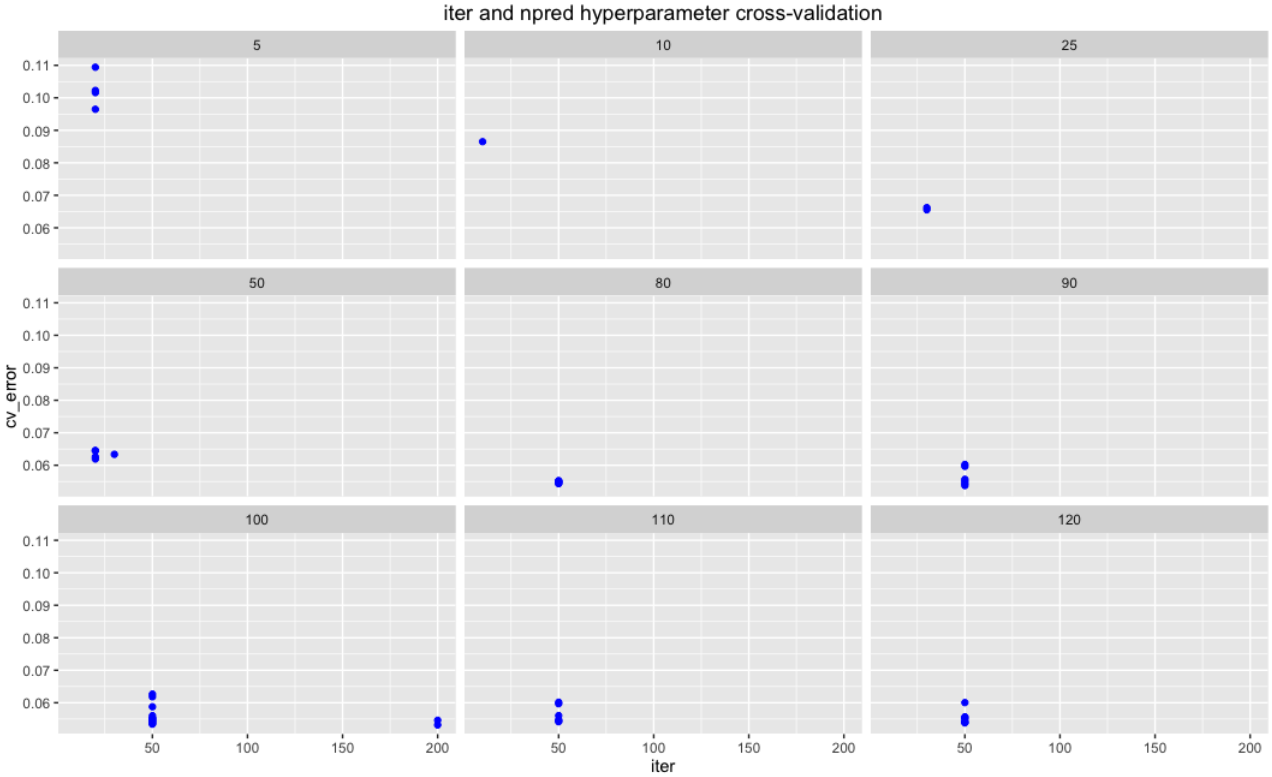
Also as hyperparameters, the following parameters related to the pre-treatment were considered:

minobs which was the minimum number of observations to create a dummy.

fm2 which was a boolean variable representing whether to consider or not the order 2 feature map.

For the set of values described in the previous paragraph, a 2, 5 and 10-fold cross validation on the dataset was carried out. The best cross validation error was obtained with $iter = 500$, $npred = 100$, $minobs = 70$, for any value of fm. For both of these, the error was in the order of $5,296\%$. We decided not to consider a feature map because of simplicity of the model.

Computation times for cross-validation were high, so we decided to output partial results on an incremental *.csv* file. A plot displaying the relationship between the *iter* and *npred* parameters with respect to the cv error is shown below:


iter and npred hyperparameter cross-validation
5
10
25
50
80
90
100
110
120
cv_error
0.11
0.10
0.09
0.08
0.07
0.06
50
100
150
200
iter

Chapter 6

Conclusion

Since the beginning of the first practice contest (the one about particles with high-energy), our team was really enthusiastic with regard to this competition. In that practice contest and in the following instance (the one suspended by the controversy) we got outstanding scores. In the final competition about NLP (the third one) we also started topping the official ranking, as we began working since the first days after the competition was open. By the second week, we were leading with a performance far above from the full-grade baseline. Hence, we decided that our exploration was enough. We had carried out the application of the tools we have learned from homework's and classes along the course. First, we strictly followed the methodology of avoiding abuse of the hold-out set, to separate into train, test, validation, cross-validation, etc. Second, we defined a 2-order feature map, representing the bi-grams in a NLP learning problem. Third, we converted the categorical variables into numeric, to be able to feed a wider scope of algorithms. Fourth, we explored a large number of diverse classifiers and obtained the expected results: some ensemble methods stand out from the rest (i.e. Boosting and Bagging). Fifth, we tuned the best performing classifiers' hyper-parameters by successive executions. Finally, we untied classifiers with similar performances by the parsimony principle in order to make our submission more robust. We were optimistic about our final model's outcome, but in the last days, and even hours before the closing, some groups significantly improved their performance, eventually moving our team away from the top.

Looking back in time and, after discussing with the highest-ranked teams when the contest had finished, we learned that there was a little more to be done to get the extra points in the ranking. We realized now that coding by ourselves an ensemble of trained classifiers would have given us the extra classification power we needed to win the competition. We will try this approach in the next opportunity.

Bibliography

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

MathWorks (2016). TreeBagger class bootstrap aggregation for ensemble of decision trees.

Appendix A

Considerations on the Source Code

1. The code relies on the 'ml.mat' file, which is provided amongst the source code. This file was generated from the provided dataset by using the 'Import' option from the MATLAB menu, due to a bug with the *loadcsv* built-in function. No changes have been made to the original data.

2. The code relies on the *strseq* function that creates a sequence of indexed strings. This is part of the *System Identification Toolbox* which he have just found might not be available immediately on all Matlab distributions.