Prudential Life Insurance Risk Assessment

Source: (https://www.kaggle.com/c/cs412-insurance)

Tools and Technologies used: Python 3.6.3, Jupyter Notebook

Introduction:

"When you know more about the inner workings of life insurance, you can make more educated choices about the type of life insurance to buy and feel more comfortable with getting it."
Retrieved from https://www.prudential.com/

We were provided with a multiclassification problem of Prudential Life Insurance Assessment on Kaggle.com. The goal of the project is to develop a machine learning model that correctly classifies and predicts the response variable (risk factor) for a client on the basis of the given data provided for every client. The response variable is one of the 8 risk factors which is the level of risk in providing insurance to the client. We have implemented the '**AdaBoost**' algorithm (also called as Adaptive boosting algorithm) as our solution which is highly used to boost the performance of the decision trees on classification problems. Adaboost is a gradient boosting technique in which a predictive model is used in the form of ensemble, particularly decision trees. It combines weak classifiers to form a strong classifier.

Dataset: The dataset contains two files: training and testing which are comma separated files. training.csv file has information of 20,000 clients on 128 different parameters and the response associated with each client. testing.csv file has data of 10,000 clients on same parameters to check the accuracy of the trained model using the first file.

The problem was challenging as the dataset was quite huge and a large number of values were missing from various columns. We applied extensive data cleaning. We applied feature engineering to identify the important variables related with the response variable and then focused to clean those variables and rest of the dataset and generated new features using most important attributes. We mainly focused on 'BMI', 'Ins_Age' attributes of the dataset.

To achieve the goal stated, we developed a predictive model which classifies the risk factor associated with every client using some libraries in Python programming language.
The results of the predictive model were extracted and submitted on kaggle.com to check the final output of the model which shows the accuracy of the predictive model made.

Project Description:

We started with cleaning the data and testing it on different measures. Before cleaning the data, we tried to understand the reasons why data is missing. We found the following cases:

1. **Missing at Random (MAR):** It occurs when the inclination of data points to be missing is not related to the missing data but to some of the observed data.

2. **Missing completely at Random (MCAR):** It means that the missing value has no relation with the hypothetical value and with the values of other variables.

Studying this, we realized that it is safe to replace the missing values for categorical variables with linear regression. Also, for continuous variables, the missing values can be replaced with mean, median or mode or multiple imputation.
For nominal attributes, if the variables contain either 2 or 3 unique values, we replaced it the missing values with the **mode** of the attribute. For attributes with object data type, we factorized the data column. In the dataset, variables with continuous values in range of 0 to 1 are replaced with mean else they are replaced with mode. After cleaning the data, we filled the remaining columns with the value '-1'.
Also, we checked the prudential life insurance client application form for the reference.

For the classifier implementation, **CART** (Classification and Regression Tree) algorithm was used. The tree construction involved the division of each node into two branches (recursively) depending upon the attribute which divided the data with the maximum reduction in gini index. To further improve the accuracy, an ensemble technique called "Adaboost" was employed where multiple classifiers were trained sequentially and depending upon the prediction of the active classifier, the weights for dataset tuples were adjusted for the next classifiers, with more focus on misclassified data.

We started with reading the training.csv file into **pandas** dataframe. Later, we created a class 'attribute' to divide the dataset at each node of the tree. It also includes the column number and the value for the column number. Within the class, there are two functions 'compare_attribute' and 'divide_data' which compares two attributes to decide which side of the tree it should be placed and to divide the data based on a given attribute x, respectively. Further, to decide on the division attribute for the root node of the tree, we created a function 'division_attribute' which is based on 'gini index'. This function gets the best attribute through the output of **Gini index** of the dataset, which gives the best attribute among all the attributes. Based on a comparison with the best attribute extracted by the division_attribute function, the dataset of the **node** is divided into two branches. Each of the branches points to a node, which is constructed recursively in the same order. Two ensure high quality for rules generated by the decision tree, the recursive partitioning of the tree was only performed until 50 tuples remained in the dataset. Any further partitioning would have overfitted the training dataset.
After calculation of best attribute, the **gini reduction** value of that particular attribute is calculated which becomes the base criteria for the further division. The attributes less than the Gini reduction values go in the left branch and those with high values goes in the right branch.
After that, construction of decision tree is implemented, and it makes dataset based on the value of their weights. After constructing the trees, we merged the results of multiple decision trees and assigned votes to the classifier results. After that, we calculated the gini_index of the prediction results we found in the previous step to find the best class among all.

Now, we implemented the Adaboost algorithm on the output. The equation of the Adaboost can be represented as (Rokach, L., 2010):

$$F(x) = sign(\sum_{m=1}^{M} \theta_m f_m(x)),$$

where f_m stands for the m_th weak classifier and theta_m is the corresponding weight.

Initially, each instance of the training dataset is weighted. The weight is set to:

$$Weight(x_i) = 1/n$$

where, x_i is the i'th instance and n is the number of the training instance.

A **decision stump** called a weak classifier is prepared on the training data using the weighted samples. Then, each decision stump makes one decision on one input variable.
We also calculated the misclassification rate for the trained model at every instance which is termed as error. Misclassified item is assigned a higher weight so that it appears in the training subset of next classifier with higher probability. A second classifier is built using the new weights, which are no longer equal. Error is calculated using the formula:

error = sum(w(i)*error(i)) / sum(w)

where, w is the weight of the training instance i.

A stage value is calculated for the trained model which provides a weighting for any predictions that the model makes.
Later the algorithm runs the classifier as specified by the user. We tested it for values like, 10, 20, 50, 100, 130.
AdaBoost Ensemble: All the weak models are added sequentially and trained using the weighted data. The process runs for the number of times specified by the user.
After calculating the error, a vote is specified to each classifier, and the final classifier is defined as the linear combination of the classifiers from each stage. It is calculated using the formula:

curr_vote = 0.5*math.log((1-error)/error)

Once an ensemble of decision trees is trained, it can predict the result for any given dataset employing the predict() function. The predictions were made depending upon the results produced by the multiple decision trees. If there was low entropy in the results returned (gini < 0.4) the most frequent class in the results was returned. But, if the entropy exceeded a certain threshold (gini >= 0.4), the returned class was the average of all the classes returned by all the decision trees. This ensured a reduction in error for weighted kappa evaluation.

Finally, the testing data is read and cleaned with the same function, 'clean_data'. A dictionary of final values is created by learning from the Adaboost algorithm model created before. The output is stored in the new file with the id and response attributes. The final output file was submitted on Kaggle.com to check the final result of the classified predicted values.

Relationship to prior work:

For the result prediction, we started to implement baseline algorithms such as linear regression. We started with cleaning data by imputing the missing values with mean or median or by frequently occurring values in the dataset. In Linear regression, we faced the issue because the relationship between the dependent and the independent variable was not linear, so the accuracy was not impressive.
Later, when the results were not impressive we tried Logistic regression and Random forest regression methods. In that, we tried more data cleaning, but the results oscillated around 0.45-0.48. We found out that, logit models are vulnerable to overconfidence and overfitting. It sometimes overstates the value of the prediction.

For the mid-term checkpoint, we used the Gradient boosting classifier of XGBoost library. we achieved our best output with the help of XGBoost classifier i.e. 0.55101

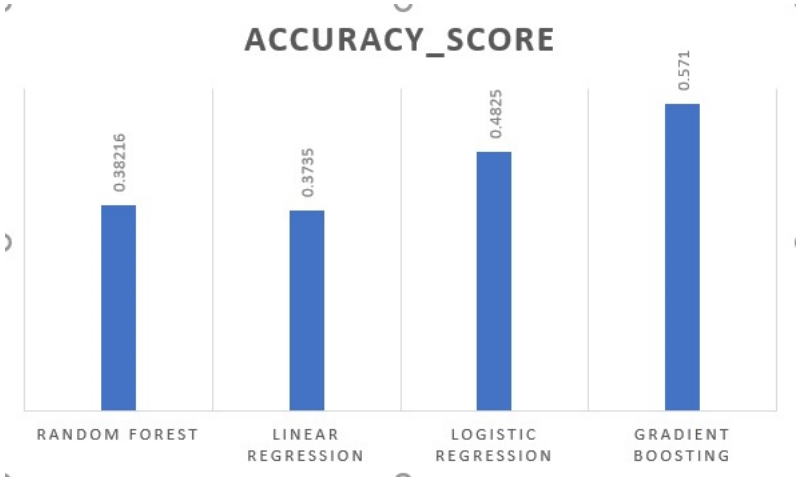

Figure: Accuracy score achieved in different algorithms

Finally, we used the decision trees algorithm and boosted it with the Adaboost learning algorithm to achieve the final results. With 10 decision trees, we achieved the accuracy score of 0.55714.

Result Interpretation:

Till midterm checkpoint, we tested models such as Linear Regression, Logistic regression and random forest algorithm to predict the value of the response variable. But the efficiency achieved was low from all of them (<0.5). After that we implemented XGBoost algorithm and achieved an efficiency of 0.55101.
For the final checkpoint, we have implemented Adaboost decision tree learning algorithm. The model learns from 10 classifiers iterations with the run time of 5 minutes of building each decision tree. We have created around 27 functions which supports the efficiency of the output. We achieved an accuracy of 0.55582 with 10 decision trees and separate trained model and 0.57967 with 100 decision trees.

Conclusion:

The results achieved now are satisfactory but there is a scope of improvement. For the improvement, multiprocessing techniques can be implemented to increase the run time of the code. It will decrease the time to create one decision tree classifier and therefore we can use high number of decision trees. The highest accuracy we have achieved is by training the model with 100 decision trees, but the run time was quite high.

References

Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016.

Hastie, T., Rosset, S., Zhu, J., & Zou, H. (2009). Multi-class adaboost. Statistics and its Interface, 2(3), 349-360.

Rokach, L. (2010). Ensemble-based classifiers. Artificial Intelligence Review, 33(1-2), 1-39.