Life Satisfaction Report

841 Winter

Donya Hamzeian

Maziar Dadbin

Summary

For predicting the life satisfaction, we tried a wide range of models namely KNN, naïve Bayes, logistic regression, random forest, boosting, SVM, and stacking. The KNN was the worst model for this data. We even tried two different preprocessing methods for preparing the data for the KNN, but the performance was not much better than random guessing. Surprisingly, random forest's, SVM's, and boosting's performance were close to each other and they all resulted in approximately the same scores in Kaggle. Naïve Bayes' and logistic regression's performance were a little worse than them. However, using random forest, boosting, SVM, and naïve Bayes in stacking did improve the score considerably and resulted in our best score which we ended the competition with. We also tried other combinations of methods in stacking but this gave the best result. In what follows, first we explain how we did the preprocessing and then we explain each of the methods mentioned above in detail.

Preprocessing

In order to prepare the data, we needed to remove some variables, to deal with missing values and outliers, to combine some categories, and to add some new variables that we thought may be of benefit. For several variables, what we needed to do was exactly the same, so in what follows, we are giving a brief explanation about what changes we made and why we did so on each group of variables.

- v3, v57, v59, v61, v63, v64, v98, v100, v124, v125, v126, v128, v130, v132, v224, v228: These variables were numeric or ordinal in essence, but their types were factor because of the existence of missing values (both "." And ".*"). We replaced all the missing values by the median to be able to deal with these variables as numeric not categorical.

- v4, v5, v17, v25, v78, v102, v123, v127, v131, v151, v153, v161, v174, v195 to v203, v258, v262, v267 to v270: We removed all these variables. The reason of removal was different case by case. In some variables, there were too many categories that were so difficult to handle. In some other cases, there were too many missing values so we thought there was no information there. There were also some variables that were redundant because

we found very similar variables with some minor differences e.g. they might have one or two more levels.

- v20: This was a categorical variable with so many categories for which we used the binning technique. The first two characters correspond to regions and the rest are related to sub regions, so we only used the first two characters.

- v69, v168, v252, v263 to v266: These were integer variables with so many missing values. We divided the range of the variables into some intervals and dealt with each interval as a category. This way we managed to keep the information in the missing values by having a category for each of the ".a", ".b", ".c", and ".d".

- v150: This was a categorical variable with so many categories. We used the function isco08toisei08 from ISCO08ConveRsions library to take the ISCO-08 codes, which are related to job titles, and assign their corresponding ISEI-08 scores. ISEI-08 score is a measure to quantify the prestige of a job, and the higher the score, the more prestigious is the job. This way we could deal with v150 as a numeric variable.

- v154, v155, v167: These are factor variables with too many categories. In order to reduce the number of categories, we just included those variables in which there was a huge difference between the proportion of satisfied and unsatisfied and excluded those in which the proportion of satisfied did not have a significant difference from the proportion of unsatisfied.

- v250 and v251: In these two variables, we detected two problems. The first one was that some answers were unreasonably high. We replaced all the observations which were more than 100 with median. The second problem was that we expected that the answer to the v250 to be smaller than v251 because v251 corresponds to hours of working "including" overtime. However, some observations in v251 were smaller than the corresponding observation in v250. We assumed this happened because some people thought v251 is about only the overtime hours. With this in mind, we replaced those observations in v251 with the sum of the corresponding observations in v250 and v251.

- We also added 4 new variables a, b, c, and d which count the number of ".a", ".b", ".c", and ".d" in the answers of each respondent respectively. We did so because we thought it may be informative if someone answered for example ".a" to many questions. However, they did not show up in the important variables derived from random forest. Even so we did not eliminate them.

Train-Validation split

It is obvious that cross validation is a common technique for finding the best tuning parameters and also for having a rough estimate of the test error. However, cross-validation is computationally intensive for this data. If we wanted to use cross validation to tune the parameters, we had to limit our search grid to be very small, but even so the tuning would have taken so long on our personal computers. Therefore, we preferred to randomly split our training data into train (75%) and validation (25%) parts. By this technique, we were able to search for the best tuning parameters on bigger grids. By doing this split, we were also able to compare different methods without being forced to having too many submits on Kaggle. It is worth mentioning that in what comes next by "train data", we mean the train data after splitting and by "whole train data", we mean the train data before splitting. Also note that we used "train data" just for the purpose of tuning the parameters and comparing different methods. For the purpose of prediction and submitting on Kaggle, we used the "whole train data".

Classification methods

After preprocessing and splitting the train data into train and validation parts, we were then ready to apply the following classification methods. As the score in Kaggle was computed based on the AUC of the predicted probabilities, we tried to maximize this value. Therefore, in the following parts, by "minimizing error", we mean "maximizing the AUC".

Random Forest

There was no need to do the feature selection before applying Random Forest because feature selection is already a part of this algorithm so, we used all the variables. The parameters that needed to be tuned in Random Forest were "ntree" and "mtry". By looking at the plot of the OOB error vs "ntree", we figured out that the OOB error did not decrease significantly for ntree>300. Therefore, we decided "ntree" to be 300. We then used tuneRF function in RandomForest library to tune "mtry" with respect to OOB error. "mtry=50" was the best. We also tried {ntree=300 , mtry=40}, {ntree=300 , mtry=50}, and {ntree=300 , mtry=60} on the train set and calculated the AUC of the predicted probabilities for validation data and the result was consistent with the previous result from the tuneRF function. This was our first submission on Kaggle and our score was around 0.86.

Boosting

Similar to Random Forest, there was no need to do feature selection before applying Boosting on the data. We set the distribution argument to "Bernoulli" and set the bag.fraction argument to its default value i.e 0.5, but the other parameters, namely shrinkage, n.trees, and interaction.depth needed to be tuned. We searched on a grid to find the best tuning parameters i.e. we trained the gradient boosting model with each set of parameters on the train data and compared the validation errors. Finally, {n.trees = 1000, interaction.depth = 4, shrinkage = 0.01} gave us the smallest validation errors.

For other models, we needed to first select some of the features. In the next section, we are going to explain our approach to feature selection.

Feature Selection

KNN, SVM, and Naïve Bayes methods do not have a built-in feature selection, so we needed to select some of the features. As stated in the lecture note "Feature selection and Stacking", there are two methods for doing feature selection. For the first method, we selected the features based on p-values derived from univariate logistic regression for numeric variables and chi-squared test for categorical variables. For the second method, we used the union of top 20 features based on the "importance" from random forest and top 20 features based on the "influence" from boosting. This gave us 27 variables. We trained SVM once with the features selected based on the first method and again with the selected variables based on the second method; the result was better when using the second method. Therefore, we simply decided to use the second method also for Naïve Bayes, KNN, and logistic regression.

KNN

We did some research and figured out that KNN does not work properly in high dimensions. Thus, we decided to select the most important features as explained above and not to apply the KNN on all the variables. Then, we needed to convert all the categorical variables to numeric because KNN works with Euclidean distance and that requires numeric variables. We had two types of categorical variables namely ordinal and nominal. As the first attempt, we simply used n-1 dummy variables instead of a categorical variable with n categories, no matter ordinal or nominal. As a result, we ended up with a large number of variables and the obtained test error was disappointing. Then, we decided to treat the ordinal categories as numeric (note that we did not do this in the main data because we wanted to preserve the information in the missing values.) This way, the number of variables decreased, but still it did not work much better than random guessing. It is worth mentioning that in both attempts we tuned k by cross validation using caret library.

SVM

We did the feature selection as explained above. After that, we needed to tune {cost, degree, gamma} and to decide about kernel and scale. Our intuition was not to scale the variables because there were many {0, 1} variables, and scaling those did not make sense. To test this intuition, we left all other parameters unchanged (not necessarily at the best values) and trained the SVM on the train data once with scale=T and once with scale=F. As we expected, the validation error was smaller for scale=F, so we decided to set scale=F for the rest of the tuning process. For the kernel, we decided to tune all the parameters for each of the "linear", "radial", and "polynomial" and then compare the three kernels (at their best settings of the other tuning parameters) and choose the best one as our ultimate choice. First, we tried polynomial kernel, but we waited for a long time and it did not work, so we completely gave up and decided just to focus on linear and radial kernels. After setting scale=F, we then did a similar search grid as we did in boosting for both linear and radial kernels i.e. we searched on a grid and trained the SVM on the train data at each point of the grid and calculated the error on the validation data. Finally, the linear kernel's test error was smaller and we ended up with cost=0.25 and kernel=" linear".

Naïve Bayes

We used the selected features as explained above i.e. the same variables as in SVM and KNN. The only parameter that needed to be tuned was "laplace". We fitted the naïve Bayes model to the train data with different values of laplace parameter, 0.01,0.1, 1, 10, and etc. Surprisingly, the validation error was minimized for laplace = 100000.

Logistic Regression

First, we tried running logistic regression on all the variables in the train data, but the model did not converge. Therefore, we decided to select only the most important features similar to what we did for the previous methods. This time the model converged, but the validation error was worse than the validation error given by random forest, boosting, and SVM.

Stacking

For doing stacking, we first trained all the above methods on the train data. Then we used the trained models to predict the validation data. After that, we fitted a logistic regression which has the column satisfied of the validation data as the dependent variable and the predictions of validation data, from different models, as independent variables. The result was a single weight for each of the models. Then we trained all the models again this time on the whole train data and predicted the test data. The final prediction was obtained using a linear combination of the predictions from each of the models using the weights from logistic regression. In terms of what methods to include in the stacking, we tried different combinations e.g. {random forest, boosting,

SVM}, {random forest, boosting, SVM, Naïve Bayes}, {random forest, boosting, SVM, logistic regression}. Because we had used the validation set to train the logistic regression part of the stacking to obtain the weight, we were no longer able to use the validation set to compare these different combinations. Therefore, we had no choice but to submit the results on the Kaggle. The best combination was {random forest, boosting, SVM, Naïve Bayes}.

We also tried to include all the selected features, obtained as explained in the "Feature Selection" section, in the logistic regression part of the stacking. The problem was that the logistic regression changed all the categorical variables with n levels to n-1 dummy variables and this led to n-1 different weights for a single categorical variable. To avoid more complexity, we decided not to include nominal variables in the logistic regression. We also converted the ordinal variables to numerical ones (actually we did the same thing as in KNN). As our last attempt, we added these remaining variables to what we had before i.e. to the predictions from random forest, boosting, SVM, Naïve Bayes. Unfortunately, this led to a worse Kaggle score and our ultimate method ended up to be stacking with {random forest, boosting, SVM, Naïve Bayes}.