CS412 Group Project

Demir Demirel, Edin Guso, Gökberk Yar, Güneş Özgün

1 Problem Definition

In this project, the goal is to build a machine learning model to predict if the applicant will pay the loan back or not given the previous decisions. Our model uses selected features that is best related to the decision, handles missing data and imbalanced data. The approach is explained in the sections below.

2 Feature Handling

We began by splitting our data into two groups; numeric and categorical. We dropped two features ***addr_state*** and ***emp_title*** because both had more than a thousand types, they were irrelevant to our task. Then we reduced the number of answers of purpose, ***home_ownership*** and ***emp_length*** features by concatenating some answers of into the same answers. We also discretized the feature ***zip_code*** by assigning its first digit to it and if it is not in the format it should be assigning 9 (the type with smallest size) handle it easier. The feature ***earliest_cr_line*** has been divided into 5 interval groups with the same motivation.

We removed the rows (applicant entry) that have more than 10 features missing, this resulted in removing more than fifty thousand applicant data.

There is a possibility of having a categorical feature with only two answers, the intuitive approach is transforming into binary answers. Thus we modified two features term and ***application_type*** with this nature. For features with multiple answers, we applied one hot encoding method.

Then we concatenated all of the feature groups. At this point we have 81 features.

We applied *Principal Component Analysis* (PCA) method as dimensionality-reduction method. With PCA, we converted the possibly correlated features into linearly uncorrelated features.

3 Missing Data

For missing data in categorical features, we used the most popular answer to impute. We filled the missing data in numerical features with each feature's median value.

Then we normalized the numerical features by using the zero-mean technique

4 Model Selection

We started off by trying a couple of different models just to get an idea about which models are best suited for this task. We tried out models such as logistic regression, decision tree and random forest. We also tried adaboosting these models to see if there were any improvements. We stayed away from all the Neural Network models due to their long training times and our TA's suggestions.

After the initial testing, we saw that adaboosted random forests gave us the best validation set scores. Even if this approach takes relatively long training times, we decided that we have the best chance of achieving high scores using this model.

5 Hyper-parameter Optimization

The next step was hyper-parameter optimization. However, trying to optimize adaboosted random forests in a brute force manner would have taken weeks, maybe even months. Therefore we decided to split our optimization process into steps and greedily pick the best parameters at that given point. Even though this would potentially result in sub optimal hyper-parameters, the huge gain in time made the trade-off reasonable.

We first wanted to find the optimal random forest and from there optimize the adaboost hyper-parameters. Initially, we realized that `class_weight='balanced_subsample'` increases our scores on average by 15% compared to `class_weight='balanced'`. The main reason behind this dramatic increase could be that giving balanced samples to each decision tree rather than giving balanced samples to the whole random forest (ignoring the samples decision trees get) makes every decision tree give more accurate predictions, resulting in much higher accuracy at the end. Therefore, we fixed that hyper-parameter from this point on.

At this point we fixed several more hyper-parameters. Those were `criterion='gini'` and `max_features='sqrt'`. After fixing three hyper-parameters in total, we found the most relevant ones out of the remaining hyper-parameters and started optimizing those. Initially, we did a brute force optimization for `max_depth` and `min_samples_leaf`. We found out that the optimal value for `max_depth` is around 40 and we found two different optimal values for `min_samples_leaf`. When we set it to 3, we would get the most successful estimations. However, train set scores were much higher compared validation set scores, indicating an over-fit. Therefore, we also saved value 75, too, which gives us the best possible scores without over-fitting.

Final step in random forest optimization was optimizing `n_estimators` hyper-parameter. The results showed that increasing `n_estimators` Improves the strong, over-fitting model (`min_samples_leaf=3`), but does not affect the weak, generalizing model (`min_samples_leaf=75`).

After picking our best two random forests, it was time to optimize the hyper-parameters using both. In the beginning, we again fixed a hyper-parameter which gave us better results most of the time. This hyper-parameter was `algorithm='SAMME.R'`.

Then, we moved onto the brute force step for optimizing `n_estimators` and `learning_rate`. We realised that for both of our random forest models, increasing the `n_estimators` hyper-parameter increased the scores slightly but unfortunately lengthened the training times greatly. `learning_rate` hyper-parameter, on the other hand was not as straightforward. It depended on `n_estimators`. If we increase the `n_estimators` to a value larger than 10, `learning_rate` lower than 1.0 gave us better scores. Contrary, if we decrease `n_estimators` to a value smaller than 10, `learning_rate` higher than 1.0 gave us better scores.

Observing the results we obtained throughout the steps of our hyper-parameter optimization, we decided our final model to be as follows:

Listing 1: Optimized Parameters

```
sub_model = RandomForestClassifier(n_estimators=75, criterion='gini',
                                    max_depth=40, min_samples_leaf=3,
                                    max_features='sqrt',
                                    class_weight='balanced_subsample')
model = AdaBoostClassifier(base_estimator=sub_model, n_estimators=40,
                           learning_rate=0.5, algorithm='SAMME.R',
                           random_state=None)
```