1. Initial data analysis and preprocessing

- **Constant columns**: There are a few columns are constants, theoretically those columns does not give us any information on different classes. But the data we are using is just a sample, maybe there will be other values if we sample more data, or there will be other values in the future data, and those might be critical for minor classes. So I kept them.
- **Outliers**: in a few columns there are some entries appear to be outliers, for instance in column 36, most entries are below 10, very few are around 1000s, but there are also a few entries in 1,000,000s. Again, without further information, it's hard to say those are really outliers, on the contrary, they might even be critical data to identify minor classes.
- **Collinearity**: If we plot the correlation matrix (Figure 1), there are a few things we notice:
    - The white stripes are the constant columns mentioned above (pandas return NA when calculating correlation between constant column with another column)
    - Most none-Na area are close to 0, but there are 4 light red line stands out, after investigating, it turns out that column 66 – 75 and 169 – 178 are most likely a sum of the columns they have high correlation with (with less than 1% exception), in other words, those columns can be calculated through linear operation from some other columns. So in order to keep the feature set simple, we can remove those columns (still better get more information about those columns before do that if possible)
- **Unbalanced label:** labels are unbalanced, with around 70.9% being class C, 14% class D, and 5% with E and A combined. So metrics other than accuracy like AUC score would be a better measurement of performance in this case. Or if more information is given about the goal (which class do we want identify more, do we care about false positives or negatives on each class), we can assign different weight to different classes.
- **Sparse features:** most part of the data sets are 0-1 columns with less than 1% entry being 1 (median of column mean is 0.84%, around 500 in 66k rows), that means it will be really hard to identify minor classes such as A and E. So if the sample size is small, there will be a lot of features with only 0s for minor classes.
- **Scaling:** data are in different magnitude, so scaling is necessary (using Standardization with 0 mean)

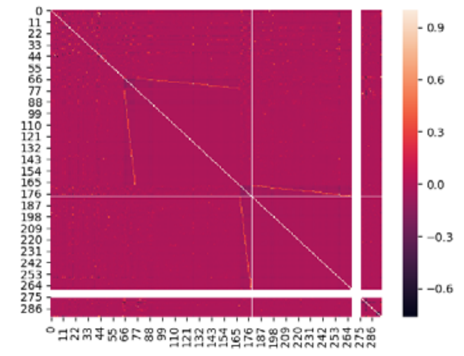


Figure 1 Correlation matrix for all features

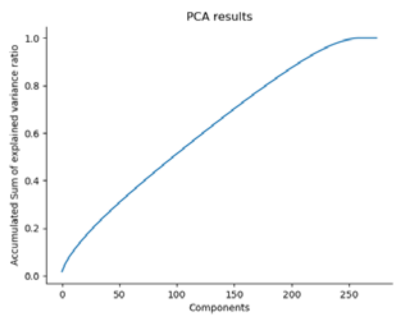


Figure 2 Accumulated Sum of explained variance ratio

2. Initial modeling

Perform some untuned classification models (Random forest, neural network with 1 hidden layer, SVM, LR, ETC …) separately with default parameters on the dataset without feature engineering (only with removing Collinearity and scaling at this stage). Accuracies are 70% on average, but considering class C is 70% of the data set, so this result is simply a result of the model guessing every data as the major class C.

This might mean the features used are not optimal and some kind of feature engineering is needed, since we don't have any background knowledge about the data, this could be hard to do. But we can still try to let the computer help us with some feature engineering, such as PCA.

PCA

Figure 2 shows the result of PCA on the original data set, to our disappointment, the information distribute on most components evenly, there is no clear "elbow point" and no real 'Principal' components, and even the little 'plateau' at the end might be just caused by the constant columns mentioned in 1st bullet point in Part 1. So sadly there is nothing we can do to improve using PCA either.

3. Ensemble model (stacking)

When the features are not optimal, we can still try to attack the dataset with complementary models and use the wisdom of the crowd to bring some decent results. An overall structural of a 2 - layer stacking model are shown in Figure 3.

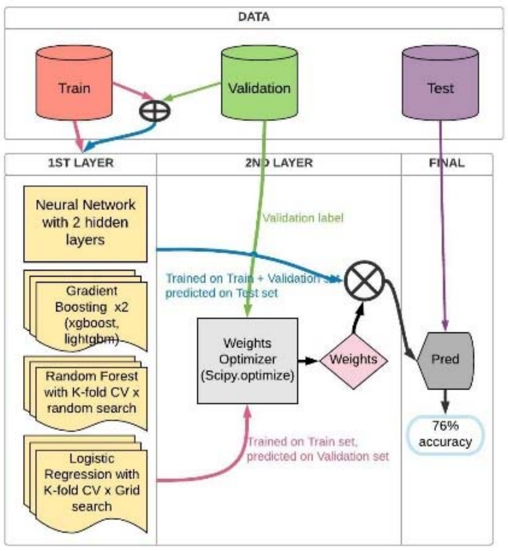
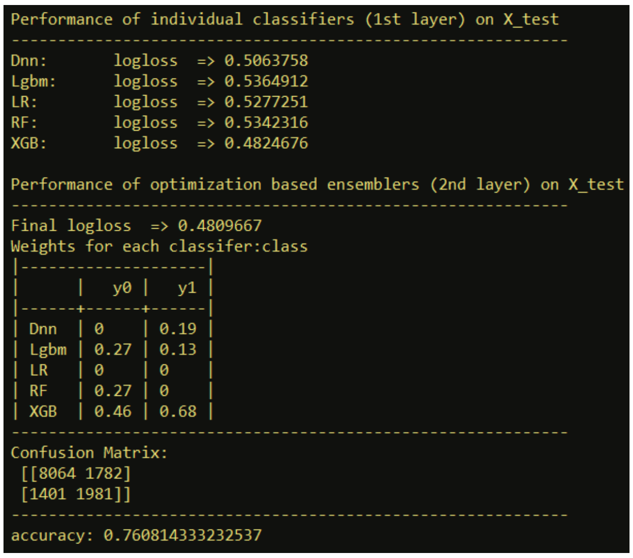

DATA
Train
Validation
Test
1ST LAYER
2ND LAYER
FINAL
Neural Network with 2 hidden layers
Gradient Boosting x2 (xgboost, lightgbm)
Random Forest with K-fold CV x random search
Logistic Regression with K-fold CV x Grid search
Validation label
Trained on Train + Validation set predicted on Test set
Weights Optimizer (Scipy.optimize)
Weights
Pred
76% accuracy
Trained on Train set, predicted on Validation set

Figure 3 Two-layer architecture for stacking ensemble classifiers



```
Performance of individual classifiers (1st layer) on X_test
-------------------------------------------------------------
Dnn:       logloss  => 0.5063758
Lgbm:      logloss  => 0.5364912
LR:        logloss  => 0.5277251
RF:        logloss  => 0.5342316
XGB:       logloss  => 0.4824676

Performance of optimization based ensemblers (2nd layer) on X_test
-------------------------------------------------------------
Final logloss  => 0.4809667
Weights for each classifer:class
|-------------------|
|      |  y0 |  y1 |
|------+------+------|
| Dnn  | 0    | 0.19 |
| Lgbm | 0.27 | 0.13 |
| LR   | 0    | 0    |
| RF   | 0.27 | 0    |
| XGB  | 0.46 | 0.68 |
-------------------------------------------------------------
Confusion Matrix:
 [[8064 1782]
 [1401 1981]]
-------------------------------------------------------------
accuracy: 0.760814333232537
```

Figure 4 results of ensemble model for 2-classes

1st layer

1.Choice of Models: For ensemblers to work, it's critical to select models that are complementary to each other, so different bias of the models can be neutralized. Neural network and tree based model are usually good complementary models. As for trees, I've chosen both bagging based (random forest) and boosting based (gradient boosting using xgboost and lightgbm), and also added logistical regression just to help out a bit (originally I have also added SVM and ETC, those are not used in the final model due to low efficiency and high loss). Final choice and structure are as shown in Figure 3.

1. **Neural Network**: first ignore the performance on validation set and only focus on the performance on train set (by increasing the complexity of the network) till it overfits (95% accuracy, certainly overfitting), then slowly simplify the network and add dropout regularization to close the gap between train set and validation set performance.
2. **Random forest**: parameters are tuned with K-fold cross validation and random search, I did some pre – search first to find out the approximate optimal region, and then only did 25 searches x 3-fold CV. Because the search process takes a long time, the best parameters found in the first training(trained on train set) will be used on the second training(using train + validation-set for final results), reason why I trained the models twice see next section **Train model**.
3. **Logistical Regression:** there are not as manly hyperparameters as in Random forest, so here I used K-fold cross validation + Grid search, also based on a pre-search to limit the search area.
4. **Gradient boosting:** due to computing power limitation, only tuned the parameter manually with sample of the data set, used slightly different parameters for xgboost and lightgbm to introduce some more variety.

2.Train model: the models will be trained twice:

1. Model set 1: only trained on train set, those models will be used with validation set to optimize weights for each model in the second layer.
2. Model set 2: trained on train+valid set, use those models combined with the weights we get from layer 2 to predict the final result on test set, the reason is so we can make use of more data for the final model

2nd layer

This part I used a simple optimization model from Scipy, the objective function being crossentropy of validation label and predictions made on validation features by Model set 1(trained with trainset).

Weights are **NxM arrays**, with rows being different model, columns with different classes.

Using those weights, we can then ensemble the results from Model set 2(trained on train + valid set) and get the final prediction for test set.

2-class Classification

Let's fist try this algorithm for a 2 class (class C vs. All) classification to see if this algorithm can do any better than just guessing every data to be class C. The results are shown in Figure 4. It shows:

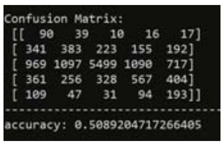
- Accuracy is 76% with a default threshold 0.5, it's more >5% better than just guess everything to be class C.
- The ensemble model has a better performance than any of one single model.
- The gradient boost model contribute the most to the final results.
- Although neuro network performs well on its own, it did not contribute much to the final results.
- Logistic regression does not contributed anything at all.
- How much each model contributed on identify each class can also be analyzed, e.g. Lightgbm contributed more on identify the minor classes than xgboost and neuro network combined.

5 class Classification (with class weights)

The algorithms can be changed to a 5 class classifier by change the n_classes to 5 in the main file. It only get an accuracy of around 70.76% if equal weights are assigned to each class, that's still not better than guess the most frequent class C, but if we make better use of the prediction probability, we can still get a lot of insights.

Assigning Class weights to minor classes: by assigning different classes weights, we can choose to put more focus on a certain class (similar as threshold in binary classification, here I simply multiply each testing data's prediction probability by class weight, then select the class with the highest weighted probability as the predicted class). A few examples with different weights are show in figure 5 – 7, (rows in the confusion matrix are real classes from A-E, and columns are predicted classes from A-E)

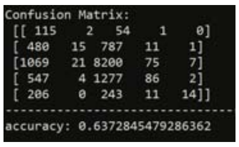
A ->



```
Confusion Matrix:
 [[  90   39   10   16   17]
 [ 341  383  223  155  192]
 [ 969 1097 5499 1090  717]
 [ 361  256  328  567  404]
 [ 109   47   31   94  193]]
-------------------------------
accuracy: 0.5089204717266405
```

Figure 5 with weights inverse proportional to each class's appearance frequency in train set

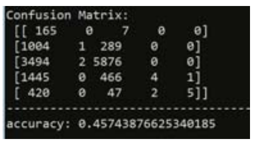
A ->



```
Confusion Matrix:
 [[ 115    2   54    1    0]
 [ 480   15  787   11    1]
 [1069   21 8200   75    7]
 [ 547    4 1277   86    2]
 [ 206    0  243   11   14]]
-------------------------------
accuracy: 0.6372845479286362
```

Figure 6 Focusing on getting a higher recall on Class A, assign class A a 20 times higher weigh than others

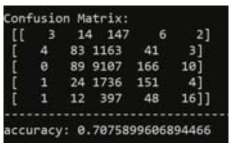
A ->



```
Confusion Matrix:
 [[ 165    0    7    0    0]
 [1004    1  289    0    0]
 [3494    2 5876    0    0]
 [1445    0  466    4    1]
 [ 420    0   47    2    5]]
-------------------------------
accuracy: 0.45743876625340185
```

Figure 7 Focusing on getting a extremely high recall on Class A, assign class A a 100 times higher weight than others

A ->



```
Confusion Matrix:
 [[    3   14  147    6    2]
 [    4   83 1163   41    3]
 [    0   89 9107  166   10]
 [    1   24 1736  151    4]
 [    1   12  397   48   16]]
-------------------------------
accuracy: 0.7075899606894466
```

Figure 8 without weight, or equal weight for every class

- The accuracy has dropped to even lower (63%) than assuming all data are class C (70%) after adding weights. But that's ok because we are not focusing on getting a high overall accuracy.
- In Figure 5, I assigned each class with a weight inverse proportional to its appearance frequency in train set (e.g. if class 1 appears 100 times, class 2 appears 50 times, class 3 appears 25 times, the weight for 1,2,3 will be [1, 2, 4]). The misclassified data are more evenly distributed among the whole confusion matrix, and if we look carefully row by row, the diagonal elements (true positives) are always the biggest in each row, this means **this assignment of weights is trying to give each class a balanced recall**.
- But try to force balance on a highly unbalanced dataset can't bring us too much insight, since we do not have any background knowledge about the data, I'm going to assume two scenarios:
  - We are **somewhat** interested in finding out A (Figure 6), e.g. it's a group very interesting to us
  - We are **extremely** interested in finding out A (Figure 7), e.g. missing an A will cause a huge lost to the company.

In Figure 6, A get a 67% recall, significantly higher than with no weight (1.7% in Figure 8). And in figure 7, the recall of A increased to a impressing 96%, but the cost is also high, we have to deal with a large amount of false positives, but as long as the cost is lower than missing an A, we are good.

Next steps

1. The weight I assigned to each class in the last part is very subjective and arbitrary, given enough time, write a optimization function to optimize the weight in a way that the we can statistically get a target average recall on a certain class, for example, we can **add another optimization layer next to our second layer, one of the constraint would be: the recall of class A should be equal or higher than 75%, objective function still being cross entropy, and class weight will be the variables to be optimized.**
2. Get more background knowledge about the data, so we can deal better with outliers and do some domain-based feature engineering. Decide on which class are we most interested in based on the nature of the problem, set the right goal, so we can assign the right weight for each class to get the results we want.
3. Try to get more data on the minor classes, also modify the code to be able to process much bigger dataset
4. More tuning on the models with more varieties, try some more advanced structures of neuro network (but it's better to get more knowledge on the data to judge if this is actually needed), do grid/random search with more granularity.