Machine Learning Project Report

Syndicate1: Alson Yang, Carrie Luo, Crystal Liu, Peggy Budidharma, Simon Cai

1 Introduction

Given the training dataset with 8000 observations, containing a reasonably balance target class ratio of 4017:3983, our objective was to build the best classifier with most appropriate features and hyperparameters to deliver accurate and robust classification for the test set of 4000 unlabeled observations. The performance was evaluated by area under the curve (AUC). This report is organized as the following: Section **2** introduces and compares the models that we considered to use as well as the evaluation metrics that we used to test and tune the models. Section **3** describes the process and reasoning of feature and model selections, and the use of tuning and stacking for performance boosting. Section **4** presents the final models selected and discusses the selecting methodology. Section **5** reflects on the challenges encountered and our approaches to resolving the challenges. Finally, Section **6** concludes our journey.

Below is a summary of **abbreviations and notations:**

LR	Logistic Regression	**KNN**	K- Nearest Neighbor	$\boldsymbol{X}$	Data with all 27 features
NN	Neural Network	**RFE**	Recursive Feature Elimination	$\boldsymbol{X_{subset}}$	Data with a subset of features selected by SFS
RF	Random Forest	**SFS**	Sequential Feature Selector		
NB	Naïve Bayes	**SCV10**	10 folds stratified crossvalidation	$\boldsymbol{X_{scaled}}$	Scaled data with all 27 features
SVM	Support Vector Machine	**AUC**	Area under the curve		

2 Models and Approaches

2.1 Model Consideration and Comparison

As the domain of this dataset and the actual meaning of each feature are not disclosed, the interpretability of the result is less of a concern; thus, theoretically, all binary classifiers can be our model candidates. Our candidate models are summarized in *Table 1* below, with their corresponding descriptions, practical advantages and disadvantages compared:

Models	Description	Advantages	Disadvantages	Normalization
Naïve Bayes	Conditional probability model	Simple, fast, do not need a large amount of data, works with a large number of features.	High bias low variance. Cannot handle complex dataset. Prediction could take more time than training.	Needed for class probabilities
LR	A linear regression classifier	Fast, explainable	Linear, cannot handle complex data.	Optional
RF	Decision tree with bagging	Accurate, embedded feature selection, less variance	Can be slow to train if the number of estimators is high.	Not needed
SVM	A linear binary classifier	Fast if a linear kernel is used with relatively low C value	Trade-off between Linear and training time	Needed
KNN	A predictive model based on its K closest points	More robust than parametric algorithms such as neural network.	All the data needs to be stored to make prediction.	Needed
NN	A non-linear classifier combining multilayers of perceptrons	Works well on unstructured data.	Need large sample size. Large amount of hyperparameters. Need to preselect features to avoid noise. Prone to overfit if epoch and batch size are not properly chosen. Slow to train.	Optional
Adaboost	A boosting algorithm focus on	Have good results with less parameter tuning	Sensitive to outliers and prone to overfit	Not needed

	training hard data more			
XGBoost	A popular tree-based gradient boosting algorithm	Speed and high accuracy, imbedded feature selection.	Large amount of hyperparameters	Not needed

Table 1Model comparison

We also researched on other popular algorithms, and LightGBM and CatBoost were chosen for their good performance. The advantages of these 2 models compared with XGBoost are summarized below:

- **LightGBM**
  LightGBM is an enhanced implementation of Gradient Boosting Decision Tree (GBDT) introduced by teams in Microsoft and Peking University (Ke et al., 2017). It improved efficiency and scalability compared with GBDT when dealing with high-dimension big data without jeopardizing the accuracy. This was made possible by Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). GOSS only utilizes data instances with large gradients, instead of using overall datasets in its Information Gain (IG) estimation; meanwhile, EFB reduces the number of features by bundling the mutually exclusive features. The optimal bundling is identified with a greedy algorithm. Therefore, it generates better performance than XGboost model.
- **CatBoost**
  CatBoost is a tree-based gradient boosting algorithm that was introduced a few years after XGBoost and LightGBM (Dorogush, Ershov & Gulin, 2017). This algorithm is designed with a focus on categorical features and tackling prediction shift. While other gradient boosting algorithms usually suffer from overfitting problems, CatBoost uses an unbiased estimate of the gradient. This is done by building the next tree pretending the observation is unseen with a trick that uses the sequence of the training set data, which is called ordered boosting. However, this trick leads to a disadvantage that all new trees share the same structure, so the algorithm generates multiple copies of the training set with random sequences. It is also a type of online training where each observation is used one by one to update models. Therefore, CatBoost is much slower than XGBoost and LightGBM, but it generates better performance.

2.2 Evaluation methodology and metrics

As this Kaggle competition evaluates performance based on area under the curve (AUC), the same evaluation metrics were used for model selection and tuning. Given a training set of 8000 observations and the evaluation metrics, we used an appropriate model validation technique to test and tune our model. One commonly used technique is cross-validation (CV). In this way, classification performance is more representative, and the tuned model would have less propensity to overfit. However, the default Scikit-learn CV function does not shuffle the data before splitting, and the ratio of the class is not consistent over all folds. Therefore, we used **stratified K-fold validation with shuffle** instead, to get a more reliable evaluation score. The training set was shuffled and split into K fold with each fold having the same class ratio. Since we have two balance classes (4017:3983), stratified CV ensures each fold to have balanced class and thus the test score over each fold would be relatively more stable and representative with less noise. This is confirmed later by the fact that we achieved a much smaller CV standard deviation (0.009 to 0.0058) after switching from default CV to SCV10 as an evaluation technique for tuning given the same classifier.

3 Experimental Process

3.1 Stage1 Exploration

At the first stage, we explored different models with their mechanisms, advantages and disadvantages in mind. We also explored different ways of feature engineering to see if any transformation or feature selection method could help.

3.1.1 Feature Engineering

Since we had no background knowledge of the dataset, it is possible that there were redundant or duplicate features. We considered feature transformation and feature selection to improve the accuracy and the generalization power of the model by removing irrelevant features. All approaches were experimented with various model explorations described in the following sections.

Feature Transformation: We drew a pair plot and we saw most of the features had skewed distribution, especially for f27 which had a high skewness. Therefore, we considered 2 log transformation, 1 on all the features as a whole set, and another 1 with only on f27, and fed them into models to compare the performance. In addition, we noticed that most distributions have long tails, which indicates the presence of outliers. We considered using the RobustScaler in sklearn to transform the features, which scales the data according to the median and quantile ranges instead of using the mean.

Feature Selection: we considered both filter methods and wrapper methods. For filtering, we first checked the feature correlations to observe if any highly correlated features can be removed to give better-generalized models. After plotting the correlation between features, we saw very high correlations between f6, f7, f8, f9 and f10. Hence, we considered removing one or more of them to improve predictive accuracy as well as reduce model complexity and training time. Moreover, we considered removing features with low variance with a threshold of 90%, as they potentially gave less information compared with other features. However, both methods did not result in any performance improvement.

We then examined the features with wrapper methods. We first used the **recursive feature elimination (RFE)** which looks at feature coefficient (LR, NN) and feature importance (RF, XGB) for different models to recursively remove features. For example, for RF we removed features with the least importance in order, until there is no AUC improvement. Another more computationally expensive method we adopt is **sequential feature selector** (SFS) where features are sequentially removed based on AUC improvement. The SFS is an iterative selection process that calculates cross-validation AUC scores for removing each feature and drops one feature at each iteration until the specified minimum number of features is reached. We found the sequential method gives us the best feature selection result and the resulting feature subset is used in the fine-tuning section.

To summarize, we consider applying log transformation, feature selection by correlation (filter), feature selection by RFE and SFS (wrapper) in this project.

3.1.2 Model Exploration

We started building models by exploring various algorithms. We also ran these models both with and without feature transformations/selections to determine if any feature preprocessing was needed. To tune the model at this stage, we ran SCV10 with grid search on possible parameters starting with default values. Take SVM as an example, we started with rbf kernel with default parameters c = 1.0 and gamma = 1/n_features. Then we built grid search with combinations of c = [1, 10, 100] and gamma = [0.05, 0.01, 0.001]. This approach was done for all candidate algorithms and the results are summarized below. Running time was calculated on a virtual machine on Google Cloud infrastructure with 24 CPUs and 16 GB RAM. Details of tuning process of XGBoost, LightGBM and CatBoost will be discussed in the Stage 2.

Models	AUC (Untuned SCV10)	AUC (Tuned SCV10)	Training Time	Predicting Time
KNN	0.6817778232822467	0.7665835782832305	16ms	2.01s
NB	0.7902211195539367	0.7902241194201526	8ms	7ms
LR	0.9043152013024913	0.9050765241151819	173ms	6ms
NN	0.873930439263894	0.9060146071609038	3.67s	9ms
SVM	0.59072910201825	0.914898057936924	6.66s	1.96s
AdaBoost	0.9011731514202254	0.91902864301263	1.22s	39ms
RF	0.8926273961842474	0.9285902334352752	329ms	15ms
LightGBM	0.9265555622761071	0.9314095910920452	767ms	43ms
XGBoost	0.9269239824820378	0.9310008299209329	1.11s	19ms
CatBoost	0.9300848759906613	0.9316689155232291	12.2s	23ms

Table 2 Model comparison

Table 2 shows all 5 tree-based type classifiers (from AdaBoost to CatBoost) outperform other classifiers, which is consistent with common empirical observations. NN did not perform well because it requires a large amount of data to train because of its complex nature.

3.2 Stage 2 Fine Tuning

At this stage, we focused on the 5 well-performing models and tuned their relevant hyperparameters. Hyperparameter-tuning is critical in the context of machine learning as it controls the trade-off between variance and bias. Having the right combination of hyperparameters can potentially deliver the lowest bias and variance, hence better performance.

Since XGBoost, Catboost and LightGBM have a large amount of hyperparameters, using grid search for all hyperparameters simultaneously was not computationally feasible. XGBoost, for example, has 13 hyperparameters to tune, and 9 of them are considered important empirically. Nonetheless, if we conducted a grid search SCV10 for 9 variables sets at the same time for a set size of only 3, we would end up with 19683 (3^9) sets of SCV10s. As it roughly took 1 hour to run approximately 2000 combinations (with n_estimators = 120) on the Google virtual machine, 19683 combinations were just not feasible to tune. Therefore, for classifiers with a large number of hyperparameters, we used a rather heuristic approach for tuning.

The following example uses XGBoost as an example but can be applied to any classifiers with a large number of hyperparameters. Instead of trying to find the global optimal value for each hyperparameter, we grouped them into several classes with a size of 2-3. For hyperparameter grouping, we considered: whether hyperparameters are of the similar type, how likely they are to be affected by each other, and the importance of parameters to the final performance of the classifier. We were then able to tune 10 sets of values for each hyperparameter within a group with a grid size of only 1000 (10^3) or 100 (10^2) at a time by using the default value for hyperparameters in other groups. After we found the optimal combinations for a group, we fixed the hyperparameter values of that group and start tuning for other groups until all groups are tuned. The order of the group tuning is also important as some groups have a greater impact on the performance than others. The group in the *Table 3* below is ordered based on the significance of impact.

Hyperparameter Groups	1. Complexity of tree structure	2. Subsample of instances and features	3. Regularization parameter	4. Speed and rounds of adjustment
XGBoost	• Min_child_weight • Max_leaf_nodes	• Subsample • Colsample_bytree	• Gamma • Lambda • Alpha	• Learning rate eta • N_estimators

Table 3 XGBoost hyper-parameter tuning groups

Group 1 is the most important as it controls the tree complexity. Finding the right values for min_child_weight and Max_leaf_nodes that fit our data is crucial, because it can lead to underfitting/overfitting if the value is too high/low. Group 2 controls the utilization rate of sample and features for building each tree. Both with a value less than one can increase the generalization of the model to reduce the risk of overfitting. However, with values that are close to zero will increase the bias for the decision tree in each iteration. Thus, finding an appropriate value is also crucial. Group 3 controls the regularization parameters. Gamma is the minimum loss reduction required to make a split, while Lambda and Alpha are the L1 and L2 normalization parameters. Since each of them imposes regularization in a different way, finding the right parameters are also useful. The learning rate eta and N_estimators were tuned last as they compensate each other. Also, larger N_estimators parameters reduce the training speed substantially. Therefore, we decreased the learning rate while increasing the N_estimators, as smaller learning rate makes convergence to the optimal solution slower, which require more iterations. As the learning rate decreases, the adjustment of XGBoost instance weight parameters is slower and hence reduce overfitting, and the finer adjustment should result in a better optimal solution than a coarser adjustment, which both can result in a lower bias and variance classifier. The same grouping and tuning methods were also applied to Catboost and LightGBM, because they all have very similar parameters.

Models	AUC (Untuned SCV10)	AUC (Tuned SCV10)	AUC (Fine-Tuned SCV10)
LightGBM	0.9265555622761071	0.9314095910920452	0.9328157716092099
XGBoost	0.9269239824820378	0.9310008299209329	0.9316818926739064
CatBoost	0.9300848759906613	0.9316689155232291	0.9331132528973873

Table 4 Improved performance after fine tuning

As observed in *Table 4*, finer tuning did not improve the model's performance significantly from a rough tuning, so stacking was performed in the next stage for further exploration.

3.3 Stage 3 Stacking

Stacking aggregates the base classifiers which look at potentially different aspects or interaction of features and therefore increases the overall performance (Džeroski & Ženko, 2004). At this stage, we combined the fine-tuned models (from Stage 2) as base estimators to build a stacking model. We used LR to be the meta-classifier to avoid adding too much complexity to the model. For base classifiers, we consider different combinations of RF, SVM, NN, XGBoost, LightGBM and CatBoost. Also, since stacking normally works better with different types of models, we decided to include SVM and NN for testing as well

despite their relatively lower AUC. We also experimented with two ways of using meta-features. One is to use the predicted probability from each base classifier and treat them as meta-features, while the other is to average the predicted probabilities from all base models first to form only one meta-feature. We have evaluated a couple of combinations of models to be the base models and results of the 4 best performing models are presented below in *Table 5*. The results show that the stacking model performs better without NN on SCV10. This may be because the AUC performance of NN is far too low compared to the scores of other models. But we kept SVM as one of our base classifiers, which in theory should provide some diversification benefits since it is not a tree-based model. Besides, simple interaction between features did not improve performance for this specific dataset, thus the kernel function in SVM could potentially look at dimensionality that we haven't captured. Therefore, we decided to choose the combination of XGBoost + LightGBM + CatBoost + SVM.

	SVM	NN	RF	XGBoost	LightGBM	CatBoost	AUC (stacked)	AUC (average)
Stack 1	Yes	Yes	Yes	Yes	Yes	Yes	0.92292819 (+/- 0.00446476)	0.92461060 (+/- 0.00456433)
Stack 2	Yes	No	Yes	Yes	Yes	Yes	0.92399925 (+/- 0.00388388)	0.93302265 (+/- 0.00424396)
Stack 3	No	No	No	Yes	Yes	Yes	0.93318250 (+/- 0.00415181)	0.93312754 (+/- 0.00427702)
Stack 4	No	No	No	No	Yes	Yes	0.93269370 (+/-0.00430525)	0.93262088 (+/- 0.00459138)

Table 5 Stacking models

4 Results and Final Model

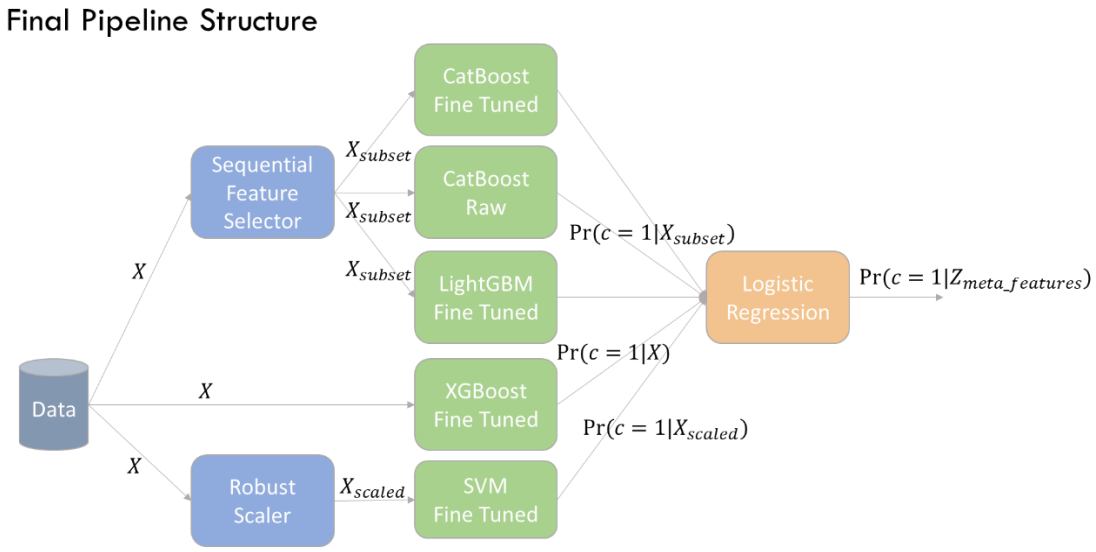

Figure 1 Final Pipeline

We revisited the feature engineering stage to seek potential improvement. We compared the feature subsets given by doing RFE and SFS and found that the subset with 22 features using SFS gives much better AUC when doing stacking. However, while we further tuned each model, we noticed that this subset did not give the best AUC for XGBoost. Hence, we developed a pipeline structure so that XGBoost used all 27 features, but other models used the subset selected by SFS.

In addition, we found that CatBoost performed well even without any parameter tuning. Hence, we include the untuned version of CatBoost as a base classifier, which should help in the case we over-tuned the hyperparameters. It turned out that the overall performance increased.

We also explored and tuned the meta-classifier to further improve the model. We started with LR by tuning c value and maximum iteration. A newer solver called ‘saga’ was used and delivered better results than the default solver. On the other

hand, we tried other algorithms as meta-classifier such as SVM and XGBoost, but we chose the tuned LR as it gave the best SCV10 AUC with stacked (without average) probabilities from each base classifier as meta-features.

Two models are chosen for submission. The final pipeline of the first model is presented in *Figure1*. This model performed the best in terms of local SCV10 with AUC 0.9337203912. The second model contains a stacking model with CatBoost and LightGBM only because it performs the best on the Kaggle public leaderboard.

5 Reflection on Adopted Approaches and Breakthroughs

At different stages of the project, we adopted various approaches to improving either the speed of training and testing or the performance of our models. This section provides the summary of the challenges encountered and the approaches we adopted to get around it. The first challenge faced during the tuning process as grid search took a huge amount of time. Running parallel grid search on Google cloud virtual machine with a higher number of CPUs was used to mitigate this. We also carefully chose the number of jobs to run parallel to avoid cases where the whole search is only waiting for 1 or 2 CPUs to finish. As discussed earlier, grouping hyperparameters and tuning one group at a time instead of tuning all hyperparameters at once also helps, but grouping has to be done with care. When we noticed there was a consistent big gap between local SCV and Kaggle public score, and models that delivered better result in terms of local SCV sometimes perform worse on the Kaggle. Thus, we revisited our SCV methods and changed to use SCV10 to provide a more stable and representative performance evaluation, and hence decrease the noise for the tuning process.

After some fine-tuning, we reached a stage where we cannot further improve both the local SCV10 AUC and online Kaggle public AUC score through hyperparameter tuning. So, we revisited our feature selection methods and found that sequential feature selections help to improve the performance for most of the modes. Stacking was also utilized to enhance the classification performance as different models might be able to capture a different aspect of the data. Thus, the stacking classifier was believed to be more sophisticated and should be able to capture more aspects than any single base classifier. Tuning the meta-classifier was another breakthrough we did to further improve our models. LR as meta-classifier with default parameters did not give an optimal prediction and with tuning the performance is greatly improved. At last, setting up pipelines for different models to use different preprocessing methods helps us to reach the optimal model we developed.

Two potential improvements could have been done to avoid overfitting and improve the evaluation score: using nested CV for tuning hyperparameter (but it is much more computationally expensive) and starting to tune the stacked classifier as a whole in early stage (instead of fine-tuning individual classifiers). We realized only closed to the submission date that hyperparameters that perform best individually do not necessarily perform well in combination with other classifiers.

6 Conclusion

Based on the training set with 8000 instances and 27 features, we build a supervised binary classification model. The final model is then used to make a classification for the testing set with 4000 unlabeled instances. The evaluation metrics that we use to guide our process is AUC on both Kaggle public score and local SCV10 score. The process involves looping through feature selections and transformations, tuning and stacking in an iterative way. Different approaches are adopted during the process to tackle various problems such as computational speed limitation. The final two models that we have are both stacked classifiers with LR being the meta-classifier, and combinations of SVM, CatBoost, XGBoost and LightGBM as the base classifiers. XGBoost uses all features and the others use selected features. One model has local SCV AUC 0.93372 and Kaggle public AUC 0.92861, and the other model has local SCV AUC 0.93356 and Kaggle public AUC 0.92959.

7 References

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T. Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In Advances in Neural Information Processing Systems (pp. 3146-3154).

Dorogush, A. V., Ershov, V., & Gulin, A. (2017). CatBoost: gradient boosting with categorical features support.

Džeroski, S., & Ženko, B. (2004). Is combining classifiers with stacking better than selecting the best one?. Machine learning, 54(3), 255-273.