Airbnb Availability Kaggle InClass Competition Report

CompSci 671
Qin He
(Kaggle username: Henry)
12/10/2021

1 Exploratory Analysis

After looking at the dataset, my first impression is that it contains multiple categorical features, and I think it is important to properly encode them so that these features can be used properly in the model. Before encoding, I checked the correlations among the predictors, because many of the predictors intuitively seem to be correlated. Fig 1.1.1 shows the correlations among all predictors. The heatmap indicates that bedroom, bed and accommodation are highly correlated, which intuitively make sense. I thought features Neighborhood and Essentials could have high correlations with some of the other features, but surprisingly they do not seem to be correlated with others.

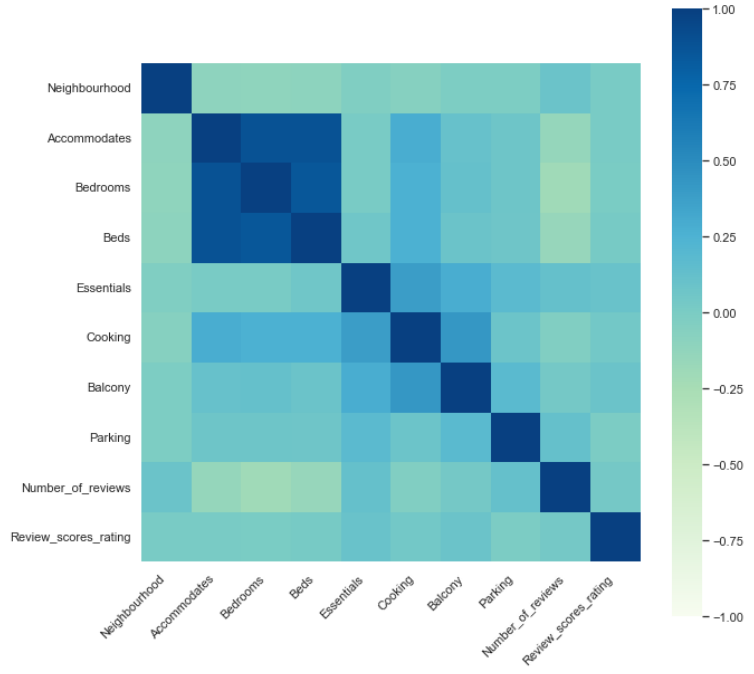
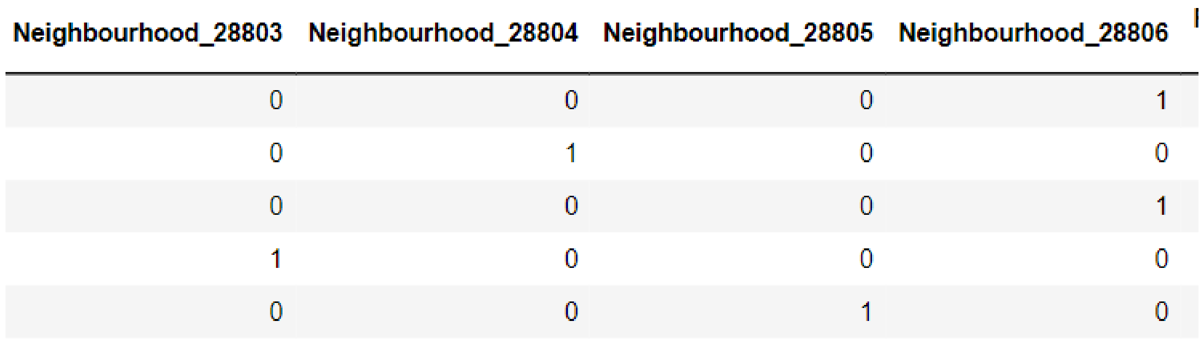


Fig 1.1.1

Then I consider the approaches to encode these features. Some of the categorical features' levels can be ordered, so it make sense to encode using integers. For example, Host response within an hour is close to host response with a few hours but are different from response within a day. Hence encoding in numbers can preserve such relationship. For other features such as neighborhood, which is labeled by zip code, it is difficult to ensure that region zip code close to each other are more similar to each other. I use one-hot encoding to encode these features. They are encoded as a series of dummy variables, and hence the original feature vector is replaced by a sparse matrix, as demonstrated by Fig 1.12 below.



Neighbourhood_28803	Neighbourhood_28804	Neighbourhood_28805	Neighbourhood_28806
0	0	0	1
0	1	0	0
0	0	0	1
1	0	0	0
0	0	1	0

Fig 1.1.2

There are two features containing text data, from which information needs to be extracted before feeding into models. Bathroom texts include information on number of bathroom and if it is private or shared. I created an additional Boolean column indicating if this unit has a private bath or not, and then I replace Bathroom texts with numeric value that simply indicating number of bathrooms. Property type includes many levels. It should be one-hot encoded, but that would created too many columns that contains little information, and I eventually decide to drop this column to avoid expanding dimension significantly.

Finally, missing values in this dataset need to be imputed. There are missing values in both training and testing data, so I cannot simply drop missing values. For categorical values with missing value, NA will also be encoded as one of the levels. It makes intuitive sense that 'unknown' may also contain information on the property. For example, property who has been listed for long time or being popular are less likely to have missing values, and new or less popular properties are more likely to miss some information. For numeric values such as review scores, missing values are imputed by the mean of all training data.

2 Methods

2.1 Models

In this project, I have attempted several algorithms to predict the availability of Airbnb based on training data. After examining data, I believe that some of the famous algorithms for classification tasks including Logistic Regression, Linear and Quadratic Discriminant Analysis, and Naïve Bayes classifier probably are not good fit for this task. The common characteristic of these methods is that they have strong assumptions on the data generation process. For example, logistic regression uses assumption based on binomial distribution, and Linear and Quadratic Discriminant Analysis model data by Gaussian distribution. Naïve bayes relies on assumptions that predictors are independent, but this assumption is not likely hold for this dataset. In addition, it is difficult to conclude if the data generating mechanism follows a specific distribution, and so these methods are not ideal candidate for this task. I then consider algorithms that do not build on strong assumptions on data models.

Support Vector machine is one of the classical algorithms for classification task. As our dataset is clearly not linear separable, a soft-margin support vector is used here. The objective of support vector machine is to construct hyperplanes to classifies data points, while the margin is maximized when constructing the hyperplanes. That is, we want not only a decision boundary that separate the classes, but also want that the points are as far away from the decision boundary as possible. Support vector machine supports various types of decision boundary. While simple linear SVM produces linear decision boundary, use of different choice of kernels, including RBF and polynomial, produce different decision boundaries. As mentioned above, in this task it is challenging to assess the underlying data model for our dataset. However, the construction of SVM does not involve assumptions on distribution of data, and hence it could be a good candidate for this task. Sci-kit learn provides a good implementation of support vector machine for Python. This package includes many other tools that are useful in preprocessing of data, scaling of data, conducting cross-validation and evaluate accuracy. So, it is easier to build and test performance of SVM. The computational complexity of SVM may be high for optimization step of SVM when compared with algorithms with analytically closed-form solution or packages that are optimized for computations such as XGBoost. However, SVM does not have many hyperparameters in its formulation, and it means that the tunning process is not complex for SVM.

In addition to Support Vector Machine, I also consider tree-based methods. There are many categorical features in our dataset, and it makes intuitive sense to adapt decision trees to make predictions. While single decision tree may seem too simple for this task, boosting decision trees produce a good candidate. Specifically, the method used here is gradient boosting of decision trees. It is an ensemble method which gives predictions based on combination of weak learner, decision trees in this case. In addition to characteristics of ordinary boosting methods, it also supports optimizing over a choice of loss function. Since this is a binary classification task, the loss function selected here is logistic loss. XGBoost is one of the commonly seen implementations of gradient boosting, and it is famous for speed. On my device, most of the XGBoost classifier takes around 0.1 second or less to train. Gradient boosting generally has many hyperparameters to include in its formulation, and it is crucial that training time of one model is not long, so that I can tune these hyperparameters over large range of values. In addition, XGBoost packages has Scikit learn style function that will be supported by cross-validation and accuracy evaluating functions offered in Scikit learn package. Especially, since gradient boosting has many hyperparameters, there are many parameters to tune on, and the grid search functions built by Scikit learn makes it easy to code this tuning process without use of multiple layers of loops.

There is another method that can improve accuracy of tree-based methods, which is bagging. Instead of training on all of the dataset, bagging build bootstrap samples. That is, it randomly samples data points from the entire dataset, and then build decision trees on these bootstrap data. Random Forest is a commonly seen bagging algorithm. In addition to building decision trees on bootstrap samples, it also select random subset of features when splitting to avoid overfitting of data. Scikit learn offers an implementation of random forest that is easy to use, train and evaluate. However, I noted that the time it takes to train random forest model is much longer than that for training gradient boosting using XGBoost. That is probably caused by differences in implementation. But differences on training time is offset by range of hyperparameters. The formulation of random forest generally has less hyperparameters than that for gradient boosting, but still running time force to narrow the range of each parameter when doing grid search.

2.2 Training

In this project I mainly use cross-validated grid search to tune my model and search for the optimal set of parameters. Initially my training data are split into train and test as I cannot test on the actual test data. 20% of the training data are assigned to testing data. Then, I use 5-fold cross-validation to evaluate the performance my models with variety choice of parameters. 5-fold cross validation would slice my training set into 5 folds and use four of them to train the model while test on the remaining 1. It will iterate make sure all of the 5 folds serve as testing data once. In the process of grid search, every time one of the hyperparameters is updated, a cross validation is used to evaluate its performance, and the mean of 5 losses is stored. The choice of parameters with lowest average mean of cross-validated loss will be chosen as the best choice of parameters. This approach is more robust than evaluating on the reserved 20% test directly, as the result on test data may depend on the train test split process, which is random. Cross-validation evaluate results over a greater number of random splits, which reduces the variance, and hence provide more robust evaluation. One of drawback of training cross-validated model is that it is computationally inefficient to train multiple models, especially when there are larger number of hyperparameters. The table below illustrates the number of model fitting required to train each of the model. As we can see that XGBoost, which the greatest number of hyperparameters, requires fitting of more than nine thousand models, which is a hundred times more than that required for training SVM model. The consequence of fitting such large number of models is also reflected by this table. While XGBoost has the best performance on fitting single model, it actually take the longest time to train, as it has to fit nine thousands model. SVM generally fits slower as it need a solver for the optimization step, but it in fact train faster than XGBoost and Scikit-learn's Random Forest Implementation when adding hyperparameter tuning.

Model	Hyperparameter tuned	Total Number of Model fitted(Using 5 fold CV)	CPU Time (in seconds)
SVM	2	5 *6*3 = 90	80.3
Gradient Boosting	5	5*(7*15 + 19*9*10) = 9075	2190.6
Random Forest	2	5*6*5*3 = 450	298.32

2.3 Hyper-parameter Selection

There are three models used in this project: support vector machine, gradient boosting and random forest. They each has very different set of parameters to select from, and different range of values to tune on. In general, support vector machine is the easiest to tune, as the formulation of it does not take many parameters. From my perspective, there are two hyperparameters that are of interest: kernel and regularizing term C. Kernel trick is important to support vector machine in the sense that different feature map can lead to different shape decision boundary. For example, linear SVM produces only linear decision boundary, and even though we consider soft-margin SVM for this task, linear decision boundary is still not a good fit of our dataset. To fix this, use of kernels can produce different decision boundary. For example, polynomial kernels can lead to decision boundary that is hyperbole or ellipse. Hence, kernel is an important hyperparameter that need to be tuned. In addition to decision boundary, another important aspect is regularization. I do not want my SVM algorithm over-exploit the training dataset and consequently generalize poorly on training dataset. Hence it is important to consider a penalty term for this model. For the package I used for training SVM, such penalty term is controlled by a hyperparameter C, which is inversely related to the strength of regularizations, and the implementation here adapts a l2 penalty. Fig 2.3.1 demonstrates how cross-validated loss changed with respect to different choice of C, and the fact that this cross-validated on choice of kernel as well.

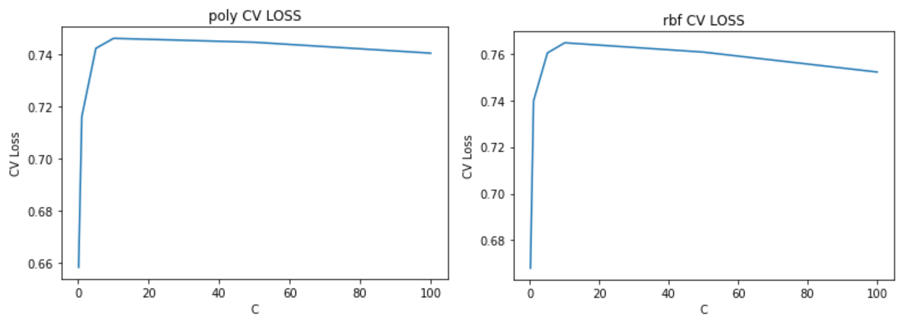


Fig 2.3.1

Unlike Support Vector Machine which does not have many parameters, tree-based methods usually include many hyperparameters such as number of trees, maximum depth and minimum weight when splitting. When training the gradient boosting, I selected five hyperparameters to be tuned: number of estimators, maximum depth of trees, learning rate for gradient boosting, minimum weight on child when splitting and lambda, a parameter that control strength l2 regularization. While I wanted to grid search over the entire parameter space span by these five hyperparameters, the tuning time for this approach is too long, and have split the tuning process. I tune two hyperparameters first and treat the optimal value of them as fixed when tuning the remaining hyperparameters. I decide to tune number of estimators and maximum depth of tree at together because they are both positive correlated with the complexity of model. That is, if I iterate value of number of estimators and maximum depth from small values to large values, both of them would shift the model from simple to complex. Then, when the optimal choice of these two hyperparameters that determine the complexity of model, I switch parameters that control the strength regularizations. The remaining three parameters control how conservative the model is when fitting data. Learning rate shrink the weights on each step which makes the updates more robust. Minimum child weight controls the minimum sum of weights of all observations required in a child, which prevent tree from growing too complex structure. Lambda is scaler that control the strength of l2 penalty. Tuning on three of them together finds the optimal strength of regularization for this model. Fig 2.32 demonstrate how the cross-validated error changed with respect to max depth and number of estimators. Both of these hyperparameters increases the model complexity, and we can observe drops in CV loss as values get larger, which signals overfitting.

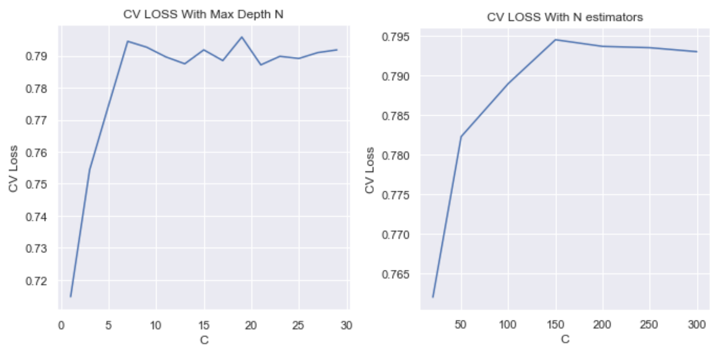


Fig 2.3.2

Random Forest's parameters are similar to gradient boosting's, but random forest has fewer important hyperparameter to tune. When tuning random forest, I decide to tune maximum depth of tree, number of estimators and maximum number of features. The first two hyperparameters play roles that are basically the same as in gradient boosting. Maximum depth controls the deepest level a tree can achieve, which in turn control the complexity of algorithm. Number of estimators determines number estimations to included, and it also determine the complexity of algorithm. Maximum features parameter is unique to random forest, as it specifies maximum number of features to split. Random Forest algorithm would split on a random subset of features at each split, and that parameter controls the maximum possible number of features can be used for one split. The package I mean using supports passing numeric value for this parameter as well as string indicating a function of sample size. I believe that passing arbitrary number independent of sample size may lead to model that is not robust, and I decide to tune on choices of built-in functions, which include square root and logarithm.

3 Results

3.1 Prediction

After tuning the model, I have evaluated all of the three models using cross validation as well as evaluating on the 20% data I reserved for testing. From the table below, it is clear that the performance of two tree-based method gradient boosting and random forest outperform SVM on both cross validation and test accuracy. The difference between gradient boosting and random forest is very small. They have cross validation and test accuracy very close to each other, and they both have slightly lower test accuracy than cross-validation error. This difference is too small too conclude that there is severe overfitting. It means that the regularization parameters do successfully prevent these two models from overfitting the data. SVM actually has test accuracy slightly higher than its best cross validation accuracy, but it is probably caused by randomness on data split.

Model	Best CV Accuracy	Test Accuracy
SVM(with rbf kernel)	0.764724674997551	0.7732441471571906
Gradient Boosting	0.7945083332167198	0.788628762541806
Random Forest	0.8030433383244008	0.7979933110367893

In addition to looking at accuracy of them directly, it is also helpful to consider the ROC and AUC of these models. Fig 3.1.1 shows that the ROCs are almost identical for all of the model. Random forest has the highest AUC and SVM has the lowest AUC, which aligns with the results on Testing data. Gradient Boosting has slightly lower AUC score than Random Forest, but the difference is not significant. Based on these metrics, I finally decide to submit predictions made by XGBoost and Random Forest separately to Kaggle website. While Random Forest achieves highest accuracy on my test data, it is Gradient Boost that has the highest accuracy on the actual test data. However, the difference on performances is insignificant. As described above, training time of Gradient boosting is much longer than that for training Random Forest, so

Random Forest can also be considered as a good candidate especially in terms of efficiency.

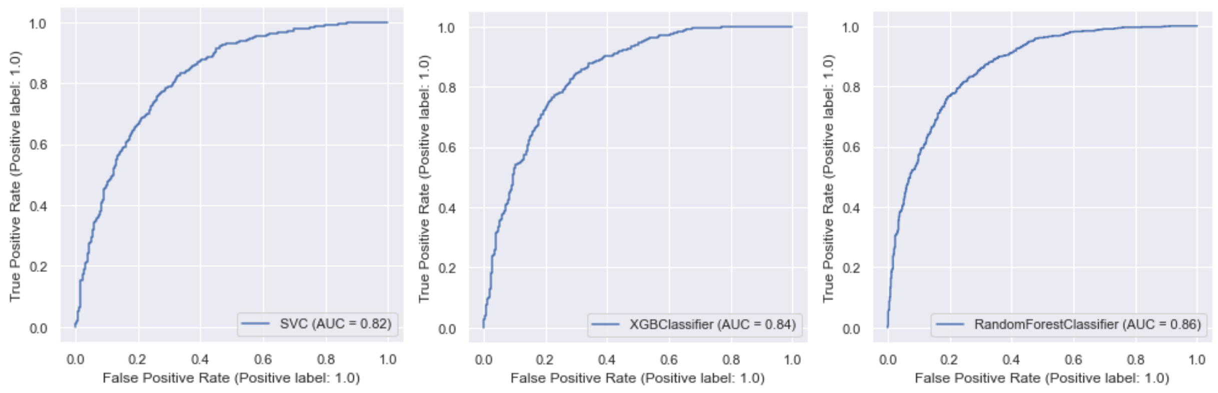


Fig 3.1.1

3.2 Fixing Mistakes

One of the most important mistakes I made is that during feature engineering I use one-hot encoding for a categorical feature with many levels, and eventually I have over 80 features included in my data set, and the training accuracy I get was stuck around 0.6. I spent hours trying to improve it, but the difference was minor. I finally realize that it is due to the dimensionality of my dataset. I included too many features and most of them carry little information that can be learned by algorithm. My reflection is that feature engineering is the most challenging part of this project. I went back and forth multiple times to change my encoding of features, and these changes do have significant impact on the performances of models. However, I feel that I have little experience on how to encode features and impute missing values.

Citations

There are several websites I used for reference

Visualization:

ROC curve: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_roc_curve.html

Heatmap: https://seaborn.pydata.org/generated/seaborn.heatmap.html

Modeling:

XGBoost: https://xgboost.readthedocs.io/en/stable/python/python_intro.html

GridSearch: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

SVM : https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Random Forest: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Code

Please refer to the file attached!