Introduction

Airbnb is a Kaggle project, whose purpose is to predict where a Airbnb user will make his first booking. My main goal in this project was to improve my programming skill and machine learning knowledge; also I'd like to get a good score. Here are the interesting things about this project:
- it has sequential data, which cannot be used in any classifier without transformation
- the datasets have a lot of missing data, which cannot be handled by most classifiers
- time element is important for prediction and model evaluation
- it evaluates predictions by a special metric NDCG (Normalized discounted cumulative gain)

Data

There are two datasets: users and sessions. Users dataset contains simple features for all the users, and sessions dataset contains sequential data (actions in the browser) for some users.

Methods

Feature Transformation

For users dataset, I used one-hot encoding for categorical features. I transformed the time feature into 1 trend and 3 seasonality components:
- trend is year + dayofyear/365
- annual seasonality is categorical month
- weekly seasonality is categorical dayofweek
- daily seasonality is binned categorical hour

For session dataset with sequential data, I tried several approaches to convert it into features. Some of the features I created are the counts of different actions for each user.

Then I combined session dataset with users dataset in two different ways:
- union with missing values added
- intersection

Dealing With Missing Values

I added indicator columns for features that are missing because the fact that they are missing is in itself useful for prediction (even if I left missing values unchanged). Most classifiers in Scikit-learn cannot handle missing values, so I imputed the missing values for some features with means.

Scaling

Typically continuous features should be scaled for better convergence and meaningful regularization. But in XGBoost classifier, scaling is unimportant because it doesn't use regularization penalty and it doesn't affect the convergence.

Weights

Most classifiers like XGBoost allow me to weigh examples. Since I was predicting for a later time period relative to the training data, I assigned bigger weights to the more recent training examples.

Validation

Because the time of the test data is after the training data, I selected validation data by random sampling from the more recent examples. In this way, the evaluation would be more precise. And I verified that the validation score is quite consistent with the test score on Kaggle.

Model Selection

With simple feature transformation, I tried several classifiers like XGBoost, Logistic Regression, Neural Network. Since XGBoost performed best and it could handle missing values, I then focused on improving XGBoost. For other classifiers, it's possible that they can perform well with different feature transformation, but I didn't explore this further.

Generally I would compare models based on their validation scores. But sometimes I would treat models as equally good if their validation scores are close enough (how close depends on standard deviation of cross validation scores and the number of models I ran). To select among those models, I would choose models whose features are more meaningful or simpler.

Coding

Replicability

Replicability is the ability to get exactly the same result in the future. It's required for analysis and for testing.

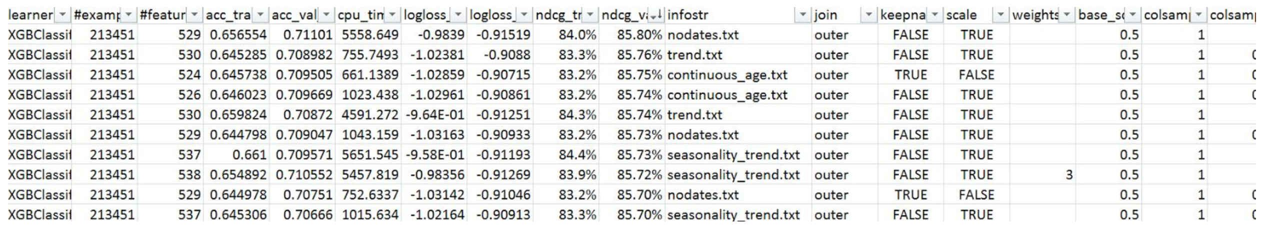
I achieved the replicability by using random seed and remembering all my inputs.

Instead of setting the random seed for the whole program, I put random seed in each function or class that has randomness. Sometimes it is unavoidable (pandas shuffle function doesn't use global random number generator). In other cases, I did it to make sure that each function or class output is not affected by the previous calls to the random generator.

The inputs are:
- command line arguments
- git revision number
- python and libraries versions
- random seed

While this achieves the replicability, for my own convenience, I added more information that is easier to look at than the git revision number:
- version number of my program; I only change it when my program output is going to be different
- classifier names and their hyperparameters
- information (in the form of a string) that determines the feature transformation

To make results more organized, I added the result of each execution into a summary table. The columns of this table contain all the inputs and the evaluation metrics. Here is a small part of the table:



learner	#examp	#featur	acc_tra	acc_val	cpu_tin	logloss_	logloss_	ndcg_tr	ndcg_va	infostr	join	keepna	scale	weights	base_sc	colsamp	colsamp
XGBClassif	213451	529	0.656554	0.71101	5558.649	-0.9839	-0.91519	84.0%	85.80%	nodates.txt	outer	FALSE	TRUE		0.5	1	
XGBClassif	213451	530	0.645285	0.708982	755.7493	-1.02381	-0.9088	83.3%	85.76%	trend.txt	outer	FALSE	TRUE		0.5	1	(
XGBClassif	213451	524	0.645738	0.709505	661.1389	-1.02859	-0.90715	83.2%	85.75%	continuous_age.txt	outer	TRUE	FALSE		0.5	1	(
XGBClassif	213451	526	0.646023	0.709669	1023.438	-1.02961	-0.90861	83.2%	85.74%	continuous_age.txt	outer	FALSE	TRUE		0.5	1	(
XGBClassif	213451	530	0.659824	0.70872	4591.272	-9.64E-01	-0.91251	84.3%	85.74%	trend.txt	outer	FALSE	TRUE		0.5	1	
XGBClassif	213451	529	0.644798	0.709047	1043.159	-1.03163	-0.90933	83.2%	85.73%	nodates.txt	outer	FALSE	TRUE		0.5	1	(
XGBClassif	213451	537	0.661	0.709571	5651.545	-9.58E-01	-0.91193	84.4%	85.73%	seasonality_trend.txt	outer	FALSE	TRUE		0.5	1	
XGBClassif	213451	538	0.654892	0.710552	5457.819	-0.98356	-0.91269	83.9%	85.72%	seasonality_trend.txt	outer	FALSE	TRUE	3	0.5	1	
XGBClassif	213451	529	0.644978	0.70751	752.6337	-1.03142	-0.91046	83.2%	85.70%	nodates.txt	outer	TRUE	FALSE		0.5	1	(
XGBClassif	213451	537	0.645306	0.70666	1015.634	-1.02164	-0.90913	83.3%	85.70%	seasonality_trend.txt	outer	FALSE	TRUE		0.5	1	(

Reliability

I achieved the reliability by running unit and integration tests with pytest and doctest.

When my output was too large to include inside the test, I used two different approaches：
- read the expected output file into a DataFrame and compare it with the actual output DataFrame; the benefits are:
    - I get more detailed information when the test fails
    - this handles approximate decimal comparison
    - I can choose which columns to compare
- compare the actual output file and expected output file. The only benefit is it’s more convenient for visual comparison.

One of my integration tests is testing my whole program. In order to achieve that, I modified the main function so that I can call it from the test. Since many files are created in each call, I created a new folder each time I ran the test so that I can compare the whole folder with the expected output folder.

Code Reuse

I created a file to include all of the common functions. These functions are not specific for this project; they can also be used in other situations.

For example, feature transformation is commonly used in data processing. So I created a function that can get a transformed dataset based on a description of the dataset and the data. Here is an example of the description for airbnb project:

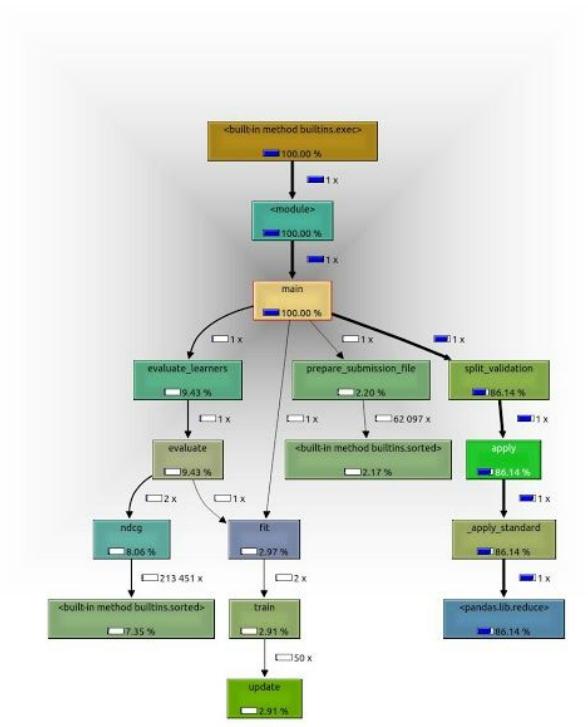
```
'''id: id
date_account_created: skip
timestamp_first_active: date, %Y%m%d%H%M%S, dayofyear
date_first_booking: skip
gender: cat
```

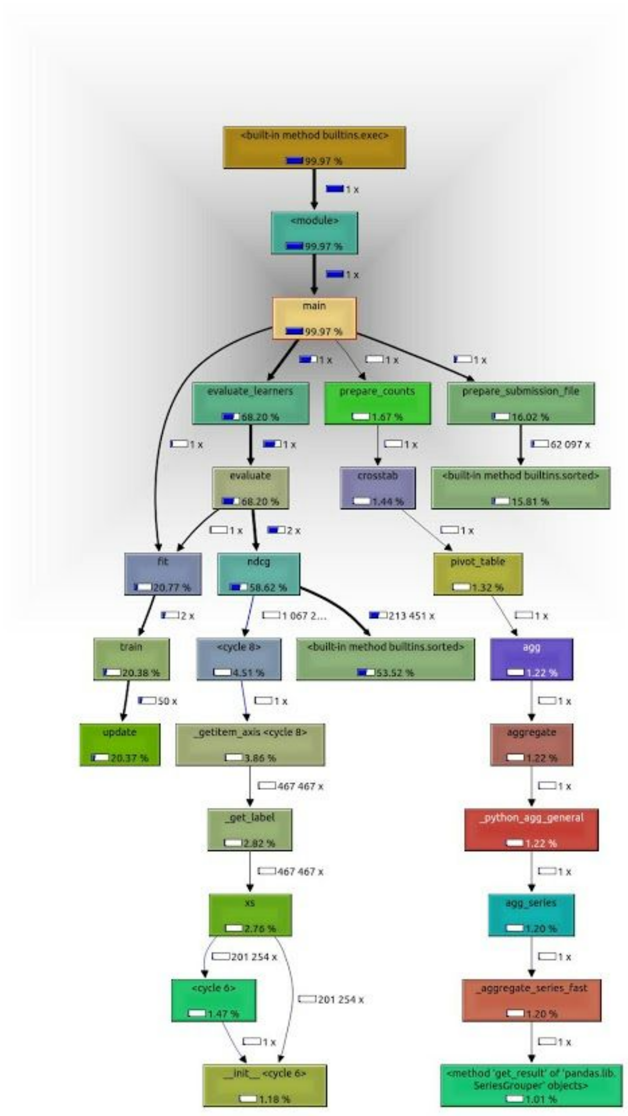
```
age: range, 10, 80; bins, 5; cat
signup_method: cat
signup_flow: cat
language: cat
affiliate_channel: cat
affiliate_provider: cat
first_affiliate_tracked: cat
signup_app: cat
first_device_type: cat
first_browser: cat
country_destination: target
'''
```

Airbnb evaluation metric (NDCG) is not available in scikit learn. So I created a function to evaluate this metric, which is consistent with scikit-learn scoring function API. This allows me to use this metric in other scikit-learn functions (like cross_val_score).

Program Performance

The original version of my program took about 80 minutes for a single run. From the analysis of the call graph generated by KCacheGrind, I found that using python function inside pandas apply unexpectedly took a very large amount of time. So I modified my code to avoid using it, which improved the running time by about 8 times.





In the modified version, I found that in the ndcg metric function, the for loop in python is even much slower than pandas apply. So I replaced the for loop with apply function, which made the ndcg function 6 times faster.

Results

Trend and any seasonality don't make a big difference in validation scores.

The union of users dataset and sessions dataset works much better than the intersection of the two.

I filled the missing values of the feature ‘age’ because it is continuous and it makes more sense to impute this feature. But the result didn’t improve compared to categorized ‘age’ without filling in missing values.

Weighting training examples to put more weight on more recent examples does help.

Conclusion

My very first try using Logistic Regression with only one-hot encoding of categorical features on users dataset achieves a test score of 86.5%, and XGBoost achieves 86.8%. My highest test score is 88.4%, which is ranked 224 out of 1463 on Kaggle.The top test score on Kaggle is 88.7%. The test score for dummy classifier 85.7%. All the scores above are evaluated by NDCG.

I was surprised how close those scores are. I’m not sure that it is too important for airbnb business to improve the score by 0.5%.