Kaggle Competition Report

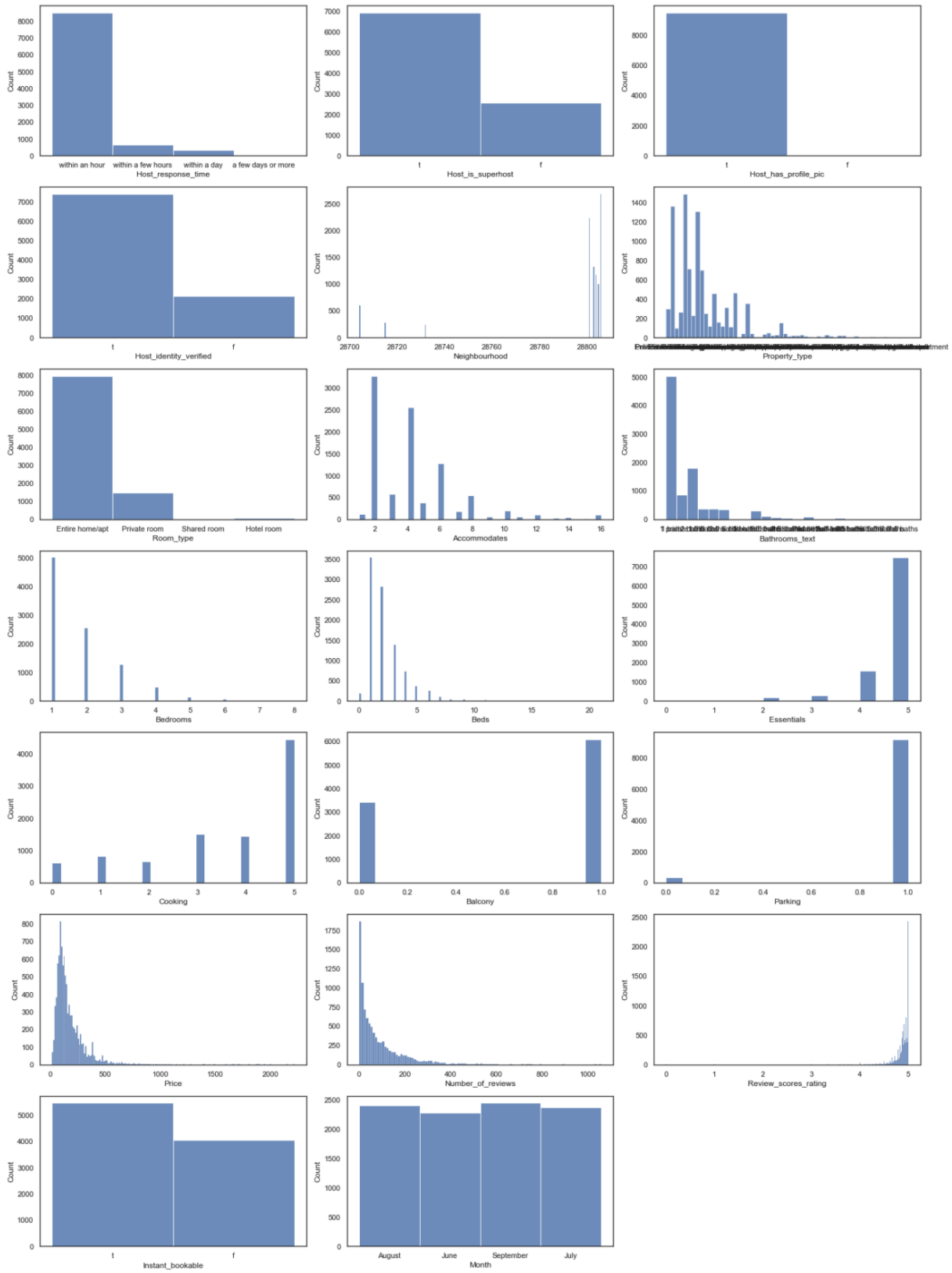
CompSci 671
Zikai Xu(Zikai_Xu_123)

2021 Dec 9th

In this report, I will apply three algorithms - Logistic Regression, Catboost and Neural Networks to classify customers' decision on Airbnb houses.

1 Exploratory Analysis

When choosing a place to live when you are on travelling, what affect people's decisions? Intuitively, we care about the price, the location, the size of the house. Additionally, if we are booking online with the Airbnb platform, we care about the reliability of the householder, so the peer reviews and the landlords' credibility appears to be important. With the data in hand, plot the distribution of the features.I merged the training set and the test set, and the distribution of all information is illustrated in the histograms below:



Observing the histograms, I notice some interesting facts:

1. There are 20 variables in total, 7 of them are numerical, and 13 are categorical (including binary variables).

2. Within numerical type of variables, considerable amount of variations are fund in some important features, such as the *Price, Number of Reviews, Review ratings*. How to deal with these outliers would be a problem, which I will discuss later.

3. Within categorical type of variables, there are also great variations for some of them. For example, *Property type* contains various types, most of them have tiny proportion, indicating that further manipulation and feature engineering is needed.

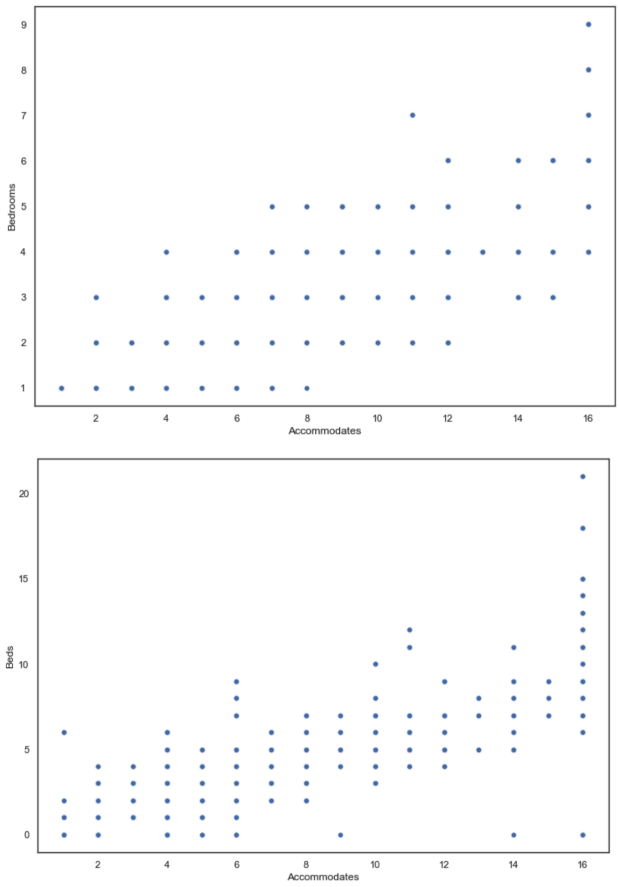
1.1 Missing Data Cleaning

Before feature engineering, I clean up the missing data. Note that in actual training, I will remove all samples with missing data in the training set. Thus, the missing data filling is only for the samples in the test set. Below is the missing data percentage in each variable (aggregate all data):

```
Decision                  24.619110
Host_response_time        11.613359
Host_is_superhost          1.130058
Host_has_profile_pic       1.130058
Host_identity_verified     1.130058
Neighbourhood              0.000000
Property_type              0.000000
Room_type                  0.000000
Accommodates               0.000000
Bathrooms_text             0.000000
Bedrooms                   7.405913
Beds                       0.221976
Essentials                 0.000000
Cooking                    0.000000
Balcony                    0.000000
Parking                    0.000000
Price                      0.000000
Number_of_reviews          0.000000
Review_scores_rating       6.750076
Instant_bookable           0.000000
Month                      0.000000
dtype: float64
```

For categorical missing data: First, the distribution of Host_response_time has revealed a clear fact that almost all landlords will respond within an hour, so I will see the missing value as the mode. Additionally, I treat NaNs in Host_is_superhost, host_has_pic, and host_identity_verified in the same way.

For numerical missing data: First, Beds and Bedrooms are strongly correlated with Accommodates, and there is no missing value in Accommodates, which is great. Below are the scatter plots of the variables.



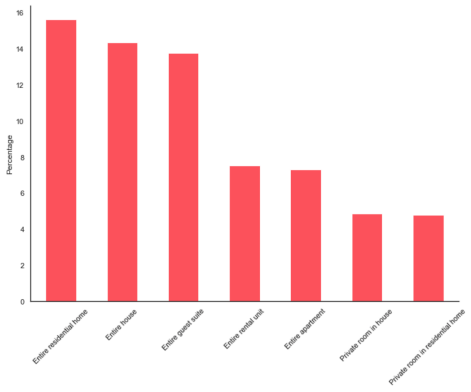
A clear linear relationship is found, I model them with two linear regression models with L-2 regularization. For missing values in Beds and Bedrooms, I will fill in them with the predicted value of the regression model and round them to integers.I consider this step as one of the novel parts of this project.

There are also missing values in review rate scores. To elaborate this, check the distribution of review scores – its mean is around 4.8. I believe the peer review score is of vital importance in terms of consumers' decisions, an extremely low score (say, 2.3) will likely lead to a negative decision. Also, no strong correlation is found between the review score and other variables. I will fill in the missing score with its mean.

1.2 Feature Engineering

For the categorical variables, I decided to do the one-hot encoding. However, it would not be efficient to throw in all the variables produced by one-hot encoding, I will bin some of them depending on the distribution.

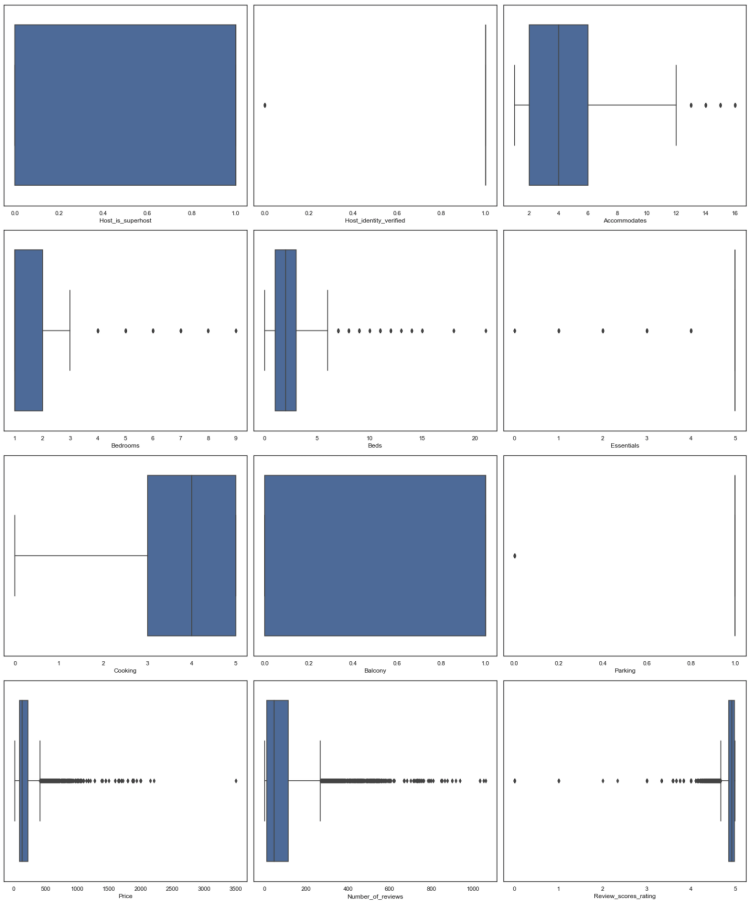
Dealing with property_type, there are 58 unique categories, I keep the top 7 categories of them and consider the rest of them as 'others'. As shown below, the top 7 categories (Entire residential home, Entire house, Entire guest suite, Entire rental unit, Entire apartment, Private room in house, Private room in residential home) covers over 70% of all categories.



Similar feature engineering is conducted to Host response time, Neighbourhood, room type, bathroom text.

1.3 Outliers Cleaning

Visualizing the outliers:



Since logistic regression is one of my methods to apply, dropping outliers becomes more important. As for Catboost, the outlier problem becomes more complicated in practice, which will be discussed in section 3.2. For now, I will recognize samples as outliers with any of these

three conditions that are met: $price > 500, review\_rate < 4, number\_of\_review > 500$. Again, it is one possible attempt, we may come back later to judge this criterion.

2 Methods

2.1 Models

I use three models to classify the data: Logistic regression, Catboost, and Neural Networks.

1. Logistic regression: when it comes to binary classification, trying logistic regression is a natural thought that pops into my mind. Simple, fast, easy to do parameters-tuning, logistic regression is the first shot that I am going to take.

2. Catboost: I expect tree-based models to work well on the task, because to some extent, it imitates the process of decision-making. When people decide for choosing a hotel or Airbnb home, they will likely set up some threshold for price budget and the lowest score they can accept, then do some weight among other factors. The reason I use Catboost is that compared to other boosting models such as XGBoost and LightGBM, it performs better in terms of learning speed and quality. Apart from that, Catboost has good user interaction options, it will also plot the learning curve automatically which helps identify overfitting issues.

3. Neural Networks: Although less intuitive than the previous models, neural networks prowess at classification. but it would be a challenge to design the structure of the network.

2.2 Training

1. Logistic regression: logistic regression is estimating the parameters of a logistic model. Specifically, suppose we have n samples, we calculate the likelihood of the parameters and maximize the log-likelihood using gradient descent.
   The runtime of the selected logistic model is 2.0917 seconds.

2. Catboost: CatBoost is based on gradient boosted decision trees. During training, a set of decision trees is built consecutively. Each successive tree is built with reduced loss compared to the previous trees (https://catboost.ai/en/docs/concepts/algorithm-main-stages). More specifically, Catboost follows the steps:

   (a) In each iteration, a decision tree is built with a limit of maximum depth. This tree is called the oblivious tree, with a small improvement from the last decision tree.

   (b) If this tree has a smaller loss on the validation set compared to all former trees, the parameter for this model is recorded as the best solution.

   (c) Catboost stops and returns the best tree models as the final model after certain iterations.

The training time of the selected Catboost model is 59.217 seconds.

3. Neural Networks: Neural networks are built with layers and nodes in them, since I will use fully connected neural networks only, each pair of nodes in adjacent layers are connected. To train a neural network, one provides inputs and labels, doing forward propagation, then adjust weights by doing backpropagation, by specifying batch size, the training process is doing stochastic gradient descent.
   The training time of the selected Neural Network is 1593.23 seconds.

2.3 Hyper-parameter Selection

2.3.1 Logistic regression

For logistic regression, I use grid search to select the best parameters, the parameters that will be selected include:

1. Solver in ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
2. Regularizations in ['none', 'l1', 'l2', 'elasticnet'] (Note: not all solvers support all regularization terms, as is shown in the resulting code.)
3. C parameter, which controls the penality strength, which can also be effective.

Taking advantage of Sklearn.model_selection.GridSearchCV, the best pair of parameters (concerning cross-validation) will be returned. And I use such parameters as the final parameter.

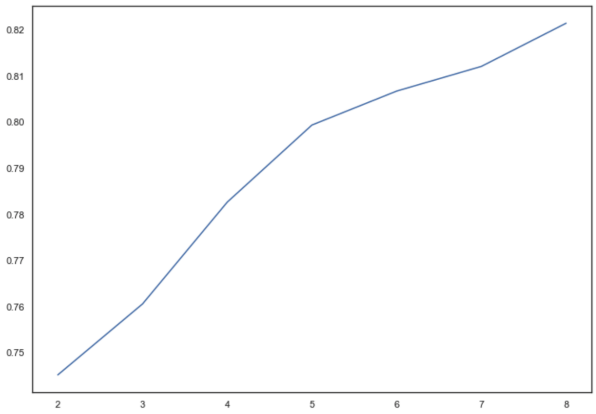
Since data normalization could affect the precision of logistic regression, I also rescaled the data and conduct the same manipulation.

2.3.2 Catboost

I made three attempts to tune the hyper-parameters of Catboost: First, with the *CatBoostClassifier(plot = True)*, I tuned the parameter of *depth* to avoid overfitting; Second, I used *Optuna* package and tuned the parameter; Third, to further imporve the accuracy, I did a base line test to check whether should I use the training data with outliers or not, and repeated *Optuna* tuning. I consider this step as one of my novel parts in the process.

1. Manual tuning: To tune depth, training on parameter *depth*= $[2, 3, 4, 5, 6, 7, 8]$ was conducted, the accuracy on the sub-testing set is shown below:

   The accuracy grows as the maximum depth grows, so should we just use $depth = 8$? No, because overfitting is one thing we should concern. Below is the plot produced by Catboost as the learning curve when $depth = 8$:



When iteration is over 1000, the accuracy on testing set is stable, while the error on the training set goes to zero – it is a pattern of overfitting, I then observed the plot learning curve and pick parameters with depth = 6 and iteration = 1500, with learning rate = 0.01.

2. Manually tuning parameters is not enough for the competition. To tune parameters with programs, I use the Optuna package. Choosing target parameters as loss_function, learning_rate,l2_leaf_reg, colsample_bylevel, depth, boosting_type, bootstrap_type, min_data_in_leaf, one_hot_max_size, random_state, I use the optuna.create_study.optimize() function to let the program searching for best accuracy depending on these parameters, I set the maximize trails as 10000 and the training time is 1 hour. It will output the parameters that make the model has the least mean squared errors, the outcome is in this form:

```
Number of completed trials: 1752
Best trial:
        Best Score: 0.4308514841827461
        Best Params:
    loss_function: CrossEntropy
    learning_rate: 0.11636623012035487
    l2_leaf_reg: 0.06904173006404413
    colsample_bylevel: 0.08496215595673909
    depth: 9
    boosting_type: Plain
    bootstrap_type: Bernoulli
    min_data_in_leaf: 10
    one_hot_max_size: 15
    random_state: 6
    subsample: 0.9838304401710416
```

3. What else can be done to improve the model? Recall that we have dropped over 500 samples as outliers, but did we do it too harshly? Is it possible that some of them carry important information that will improve the model? With these questions in mind, I use Catboost as the baseline model to test to what rate should we abandon outliers. To be specific, [0.8, 0.85, 0.9, 0.95, 0.98, 0.99] are the thresholds that I am curious about – should the houses with a price larger than 80% of others be dropped, or should that rate be 99%?

First, I create a catboost model with parameter iterations = 1500, depth = 5, loss_function='CrossEnt learning_rate= 0.1,leaf_estimation_method = 'Gradient', random_seed = 6, which is a standard setup. Then, I train the model with data selected with the outlier drop rate listed above, I judge the performance of the models with their mean square error. Here is the result:

```
0.8 : 0.25 : 7312
0.85 : 0.17142857142857143 : 7300
0.9 : 0.10256410256410256 : 7278
0.95 : 0.1346153846153846 : 7212
0.98 : 0.07547169811320754 : 7206
0.99 : 0.07407407407407407 : 7204
```

It turns out that setting the drop threshold to be 0.99 performs best, which means that the specific catboost model that we created above favors the dataset where no one is dropped.
Thus, if I ask optuna to tune parameters with complete training data with all outliers, will it give me a better model than if I give it data without outliers? To figure out this, I run the process in step 2 again with the data with outliers. And this time optuna suggests a different set of parameters:

```
Number of completed trials: 1900
Best trial:
        Best Score: 0.6071606606134905
        Best Params:
    loss_function: CrossEntropy
    learning_rate: 0.005496649026380579
    l2_leaf_reg: 0.038845321749756294
    colsample_bylevel: 0.08791720758027873
    depth: 6
    boosting_type: Ordered
    bootstrap_type: Bayesian
    min_data_in_leaf: 2
    one_hot_max_size: 15
    random_state: 12
    bagging_temperature: 1.319388900442126
```

However, when I test predictions with the testing data, the model with parameters suggested by optuna with full data (with outliers) does not behave as well as the model with parameters suggested by optuna with restricted data (without outliers). It violates our mini-experiment, so what is happening?
I think there are two possible explanations: First, the problem is in our mini-experiment design, the baseline Catboost parameter could be problematic, for example, the iteration number is too small and there might be an underfitting problem. Second, the design of the experiment is not valid in terms of scoring and choosing the threshold for abandoned data

2.3.3 Neural Network

When it comes to tuning neural networks, there seems to have no standard operation process for selecting the number of layers and number of nodes in each layer. There is a theoretical finding by Lippmann [Lippmann, 1987] that shows that an MLP with two hidden layers is sufficient for creating classification regions of any desired shape. This is instructive, although it should be noted that no indication of how many nodes to use in each layer or how to learn the weights is given. The way that I tune the parameter is heuristic – Given this amount of data, I firstly fix the number of layers to be four and randomly assign numbers of each node and test their performance, then, I find that there is a certain shape that produces relatively highest accuracy – that is, with the highest number of nodes in the first layer, and reduce the number of nodes as layer number grows.

The activation function is another element to concern, generally speaking, the sigmoid function is used widely, but Relu has been regaining favor in recent years, I tried both functions and decide to use Relu.

3 Results - Prediction

My Kaggle username is Zikai_Xu_123, my score on the Kaggle competition is 0.30327.

According to Kaggle, the (best) accuracy score for each classifier are: Logistic regression with l2 regularization: 0.33196; Catboost: 0.30327 Neural Network: 0.32513

Overall, Catboost performs the best among the three algorithms that I pick. It makes sense since, on the one hand, the decision tree's structure is alike to human reasoning when making a decision, and on the other hand, I have tuned the parameter sophisticatedly.

3.1 Fixing Mistakes

When doing feature engineering. At first, I did one-hot encoding separately - an encoding on the training set and an encoding on the test set. After I trained my model, I tried to fit it to the testing data, but it was not possible to do that due to dimensional problem. It took me a lot of time to figure out what was going on, and it turned out that new type of houses appears in the test data that is not seen in the training set, thus, the one-hot encoding on training data provides less dimension than that provided by the testing set. One way to fix the problem is to merge the two sets together and do one-hot encoding, and that is what I did. However, since I binned the features, the new types appeared in the testing data was just classified as 'other types', so the problem does not have significant effect.

References

[Lippmann, 1987] Lippmann, R. (1987). An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(2):4–22.

In [4]:
```
pip install --upgrade category_encoders
```

```
Requirement already satisfied: category_encoders in /Users/xuzikai/Downloads/ana
conda3/lib/python3.8/site-packages (2.3.0)
Requirement already satisfied: scipy>=1.0.0 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from category_encoders) (1.6.2)
Requirement already satisfied: numpy>=1.14.0 in /Users/xuzikai/Downloads/anacond
a3/lib/python3.8/site-packages (from category_encoders) (1.20.1)
Requirement already satisfied: scikit-learn>=0.20.0 in /Users/xuzikai/Downloads/
anaconda3/lib/python3.8/site-packages (from category_encoders) (1.0.1)
Requirement already satisfied: pandas>=0.21.1 in /Users/xuzikai/Downloads/anacon
da3/lib/python3.8/site-packages (from category_encoders) (1.2.4)
Requirement already satisfied: patsy>=0.5.1 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from category_encoders) (0.5.1)
Requirement already satisfied: statsmodels>=0.9.0 in /Users/xuzikai/Downloads/an
aconda3/lib/python3.8/site-packages (from category_encoders) (0.12.2)
Requirement already satisfied: python-dateutil>=2.7.3 in /Users/xuzikai/Download
s/anaconda3/lib/python3.8/site-packages (from pandas>=0.21.1->category_encoders)
(2.8.1)
Requirement already satisfied: pytz>=2017.3 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from pandas>=0.21.1->category_encoders) (2021.1)
Requirement already satisfied: six in /Users/xuzikai/Downloads/anaconda3/lib/pyt
hon3.8/site-packages (from patsy>=0.5.1->category_encoders) (1.15.0)
Requirement already satisfied: joblib>=0.11 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from scikit-learn>=0.20.0->category_encoders) (1.
0.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /Users/xuzikai/Downloads/
anaconda3/lib/python3.8/site-packages (from scikit-learn>=0.20.0->category_encod
ers) (2.1.0)
Note: you may need to restart the kernel to use updated packages.
```

In [5]:
```
!pip install optuna
```

```
Collecting optuna
  Downloading optuna-2.10.0-py3-none-any.whl (308 kB)
     |████████████████████████████████| 308 kB 2.7 MB/s eta 0:00:01
Requirement already satisfied: scipy!=1.4.0 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from optuna) (1.6.2)
Requirement already satisfied: tqdm in /Users/xuzikai/Downloads/anaconda3/lib/py
thon3.8/site-packages (from optuna) (4.59.0)
Collecting cmaes>=0.8.2
  Downloading cmaes-0.8.2-py3-none-any.whl (15 kB)
Requirement already satisfied: packaging>=20.0 in /Users/xuzikai/Downloads/anaco
nda3/lib/python3.8/site-packages (from optuna) (20.9)
Requirement already satisfied: numpy in /Users/xuzikai/Downloads/anaconda3/lib/p
ython3.8/site-packages (from optuna) (1.20.1)
Requirement already satisfied: sqlalchemy>=1.1.0 in /Users/xuzikai/Downloads/ana
conda3/lib/python3.8/site-packages (from optuna) (1.4.7)
Requirement already satisfied: PyYAML in /Users/xuzikai/Downloads/anaconda3/lib/
python3.8/site-packages (from optuna) (5.4.1)
Collecting colorlog
  Downloading colorlog-6.6.0-py2.py3-none-any.whl (11 kB)
Collecting cliff
  Downloading cliff-3.10.0-py3-none-any.whl (80 kB)
     |████████████████████████████████| 80 kB 24.5 MB/s eta 0:00:01
Collecting alembic
  Downloading alembic-1.7.5-py3-none-any.whl (209 kB)
     |████████████████████████████████| 209 kB 28.3 MB/s eta 0:00:01
Requirement already satisfied: pyparsing>=2.0.2 in /Users/xuzikai/Downloads/anac
onda3/lib/python3.8/site-packages (from packaging>=20.0->optuna) (2.4.7)
Requirement already satisfied: greenlet!=0.4.17 in /Users/xuzikai/Downloads/anac
onda3/lib/python3.8/site-packages (from sqlalchemy>=1.1.0->optuna) (1.0.0)
```

```
Requirement already satisfied: importlib-metadata in /Users/xuzikai/Downloads/an
aconda3/lib/python3.8/site-packages (from alembic->optuna) (4.8.2)
Collecting importlib-resources
  Downloading importlib_resources-5.4.0-py3-none-any.whl (28 kB)
Collecting Mako
  Downloading Mako-1.1.6-py2.py3-none-any.whl (75 kB)
     |████████████████████████████████| 75 kB 12.6 MB/s eta 0:00:01
Collecting PrettyTable>=0.7.2
  Downloading prettytable-2.4.0-py3-none-any.whl (24 kB)
Collecting cmd2>=1.0.0
  Downloading cmd2-2.3.3-py3-none-any.whl (149 kB)
     |████████████████████████████████| 149 kB 20.2 MB/s eta 0:00:01
Collecting stevedore>=2.0.1
  Downloading stevedore-3.5.0-py3-none-any.whl (49 kB)
     |████████████████████████████████| 49 kB 17.2 MB/s eta 0:00:01
Collecting autopage>=0.4.0
  Downloading autopage-0.4.0-py3-none-any.whl (20 kB)
Collecting pbr!=2.1.0,>=2.0.0
  Downloading pbr-5.8.0-py2.py3-none-any.whl (112 kB)
     |████████████████████████████████| 112 kB 29.7 MB/s eta 0:00:01
Requirement already satisfied: wcwidth>=0.1.7 in /Users/xuzikai/Downloads/anacon
da3/lib/python3.8/site-packages (from cmd2>=1.0.0->cliff->optuna) (0.2.5)
Collecting pyperclip>=1.6
  Downloading pyperclip-1.8.2.tar.gz (20 kB)
Requirement already satisfied: attrs>=16.3.0 in /Users/xuzikai/Downloads/anacond
a3/lib/python3.8/site-packages (from cmd2>=1.0.0->cliff->optuna) (20.3.0)
Requirement already satisfied: zipp>=0.5 in /Users/xuzikai/Downloads/anaconda3/l
ib/python3.8/site-packages (from importlib-metadata->alembic->optuna) (3.4.1)
Requirement already satisfied: MarkupSafe>=0.9.2 in /Users/xuzikai/Downloads/ana
conda3/lib/python3.8/site-packages (from Mako->alembic->optuna) (1.1.1)
Building wheels for collected packages: pyperclip
  Building wheel for pyperclip (setup.py) ... done
  Created wheel for pyperclip: filename=pyperclip-1.8.2-py3-none-any.whl size=11
107 sha256=fa87360cad7de782c65069099152014a95bc9cc24be8cb8433af08a99275e0af
  Stored in directory: /Users/xuzikai/Library/Caches/pip/wheels/7f/1a/65/84ff8c3
86bec21fca6d220ea1f5498a0367883a78dd5ba6122
Successfully built pyperclip
Installing collected packages: pyperclip, pbr, stevedore, PrettyTable, Mako, imp
ortlib-resources, cmd2, autopage, colorlog, cmaes, cliff, alembic, optuna
Successfully installed Mako-1.1.6 PrettyTable-2.4.0 alembic-1.7.5 autopage-0.4.0
cliff-3.10.0 cmaes-0.8.2 cmd2-2.3.3 colorlog-6.6.0 importlib-resources-5.4.0 opt
una-2.10.0 pbr-5.8.0 pyperclip-1.8.2 stevedore-3.5.0
```

In [3]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import collections
from copy import deepcopy

from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

from catboost import CatBoostClassifier

import optuna
from optuna.samplers import TPESampler
from sklearn.metrics import mean_squared_error

# Draw inline
%matplotlib inline
```

```
# Set figure aesthetics
sns.set_style('white')
sns.set(rc={'figure.figsize':(11.7,8.27),'figure.facecolor':'white'})
```

Import Data

In [49]:
```
df1 = pd.read_csv('train.csv', index_col=False)
df1.drop(['id'],axis = 1,inplace = True)

df2 = pd.read_csv('test.csv', index_col=False)
idarray = df2['id'].to_numpy()
df2.drop(['id'],axis = 1,inplace = True)


df = pd.concat([df1,df2],axis = 0,ignore_index=True)
```

In [50]:
```
df.head()
```

Out[50]:

	Decision	Host_response_time	Host_is_superhost	Host_has_profile_pic	Host_identity_verified
0	1.0	within an hour	t	t	t
1	1.0	within an hour	t	t	t
2	0.0	within a few hours	t	t	t
3	1.0	within an hour	t	t	t
4	0.0	within an hour	t	t	t

5 rows × 21 columns

In [51]:
```
#checking type of every column in the dataset
df.dtypes
```

Out[51]:
```
Decision                 float64
Host_response_time        object
Host_is_superhost         object
Host_has_profile_pic      object
Host_identity_verified    object
Neighbourhood              int64
Property_type             object
Room_type                 object
Accommodates               int64
Bathrooms_text            object
Bedrooms                 float64
Beds                     float64
Essentials                 int64
Cooking                    int64
```

```
Balcony                    int64
Parking                    int64
Price                     object
Number_of_reviews          int64
Review_scores_rating     float64
Instant_bookable          object
Month                     object
dtype: object
```

In [52]:
```
# Turn Price into floats
PriceValue=[]
for c in df.Price:
    PriceValue.append(float(c[1:].strip().replace(',', '')))
df['Price'] = np.array(PriceValue)
```

Handle the Missing Data

In [53]:
```
# Missing Value Percentage
df_nan = (df.isnull().sum() / df.shape[0]) * 100
df_nan
```

Out[53]:
```
Decision                 24.619110
Host_response_time       11.613359
Host_is_superhost         1.130058
Host_has_profile_pic      1.130058
Host_identity_verified    1.130058
Neighbourhood             0.000000
Property_type             0.000000
Room_type                 0.000000
Accommodates              0.000000
Bathrooms_text            0.000000
Bedrooms                  7.405913
Beds                      0.221976
Essentials                0.000000
Cooking                   0.000000
Balcony                   0.000000
Parking                   0.000000
Price                     0.000000
Number_of_reviews         0.000000
Review_scores_rating      6.750076
Instant_bookable          0.000000
Month                     0.000000
dtype: float64
```

In our case, missing data that is observed does not need too much special treatment. Looking into the nature of our dataset we can state further things: columns "name" and "host_name" are irrelevant and insignificant to our data analysis, columns "Host_response_time" need very simple handling. To elaborate, "last_review" is date; if there were no reviews for the listing - date simply will not exist. In our case, this column is irrelevant and insignificant therefore appending those values is not needed. we can see that in "number_of_review" that column will have a 0, therefore following this logic with 0 total reviews there will be 0.0 rate of reviews per month. Therefore, let's proceed with removing columns that are not important and handling of missing data.

In [54]:
```
df.Host_response_time.describe()
```

Out[54]:
```
count                8760
unique                  4
top       within an hour
freq                 7697
Name: Host_response_time, dtype: object
```

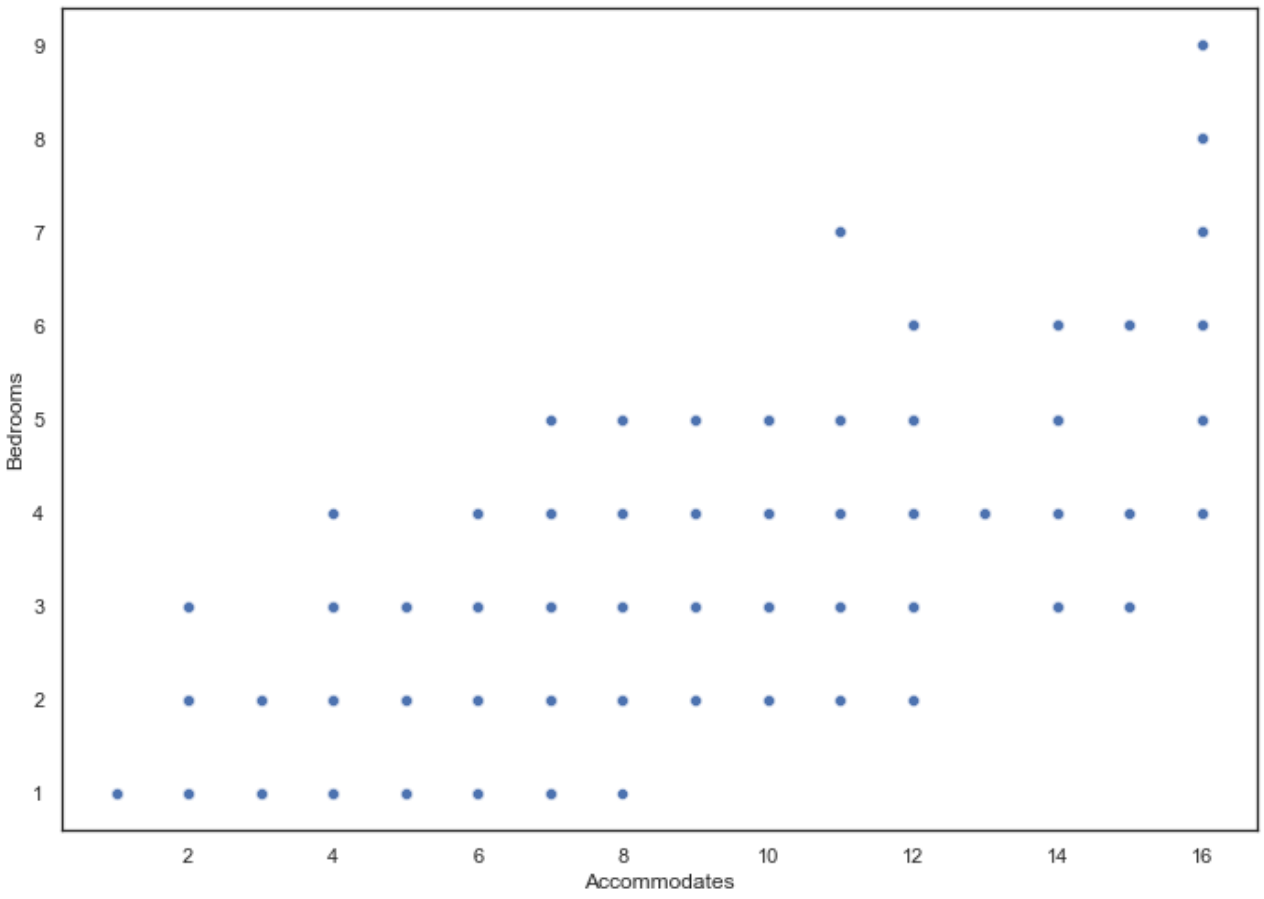
For respose time, 7697/8760 of the owners will respond within an hour, we will just assume the missing value to be 'within an hour'. We will replace missing values in this three columns with the modes.

In [55]:
```
df.fillna({'Host_response_time':'within an hour'}, inplace=True)
df.fillna({'Host_is_superhost':'t'}, inplace=True)
df.fillna({'Host_has_profile_pic':'t'}, inplace=True)
df.fillna({'Host_identity_verified':'t'}, inplace=True)
```

For the missing in Beds and Bedrooms, we will use accomodates to predict them

In [56]:
```
sns.set_style('white')
sns.scatterplot(x='Accommodates',y='Bedrooms',data = df)
```

Out[56]:
```
<AxesSubplot:xlabel='Accommodates', ylabel='Bedrooms'>
```



In [57]:
```
# Predict number of bedrooms
from sklearn.linear_model import LinearRegression
X = df.dropna().Accommodates.to_numpy().reshape(-1,1)
y = df.dropna().Bedrooms.to_numpy().reshape(-1,1)
```

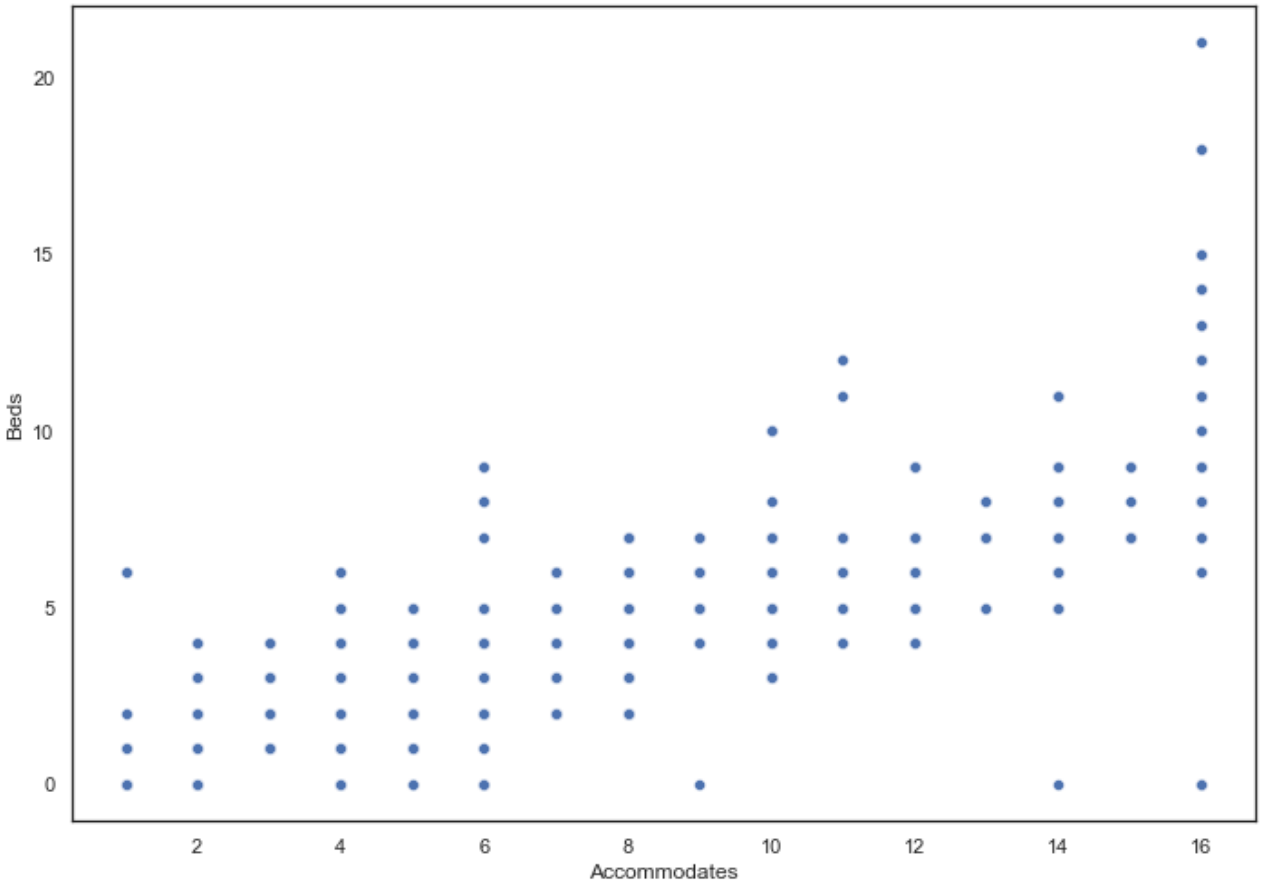
```
reg = LinearRegression().fit(X, y)
reg.score(X,y)
```

Out[57]:
```
0.7830275914335578
```

In [58]:
```
# Update Bedroom NaN with predicted value
pred_bdrms = np.rint(reg.predict(df.Accommodates[df.Bedrooms.isnull()].to_numpy(
a = 0
for i in df.Bedrooms[df.Bedrooms.isnull()].index:
    df.at[i,'Bedrooms'] = pred_bdrms[a]
    a+=1
```

In [59]:
```
sns.scatterplot(y='Beds',x='Accommodates',data = df)
```

Out[59]:
```
<AxesSubplot:xlabel='Accommodates', ylabel='Beds'>
```



In [60]:
```
# Predict number of Beds
X = df.dropna().Accommodates.to_numpy().reshape(-1,1)
y = df.dropna().Beds.to_numpy().reshape(-1,1)
reg = LinearRegression().fit(X, y)
reg.score(X,y)
```

Out[60]:
```
0.7827665440425944
```

In [61]:
```
# Update Beds information
pred_beds = np.rint(reg.predict(df.Accommodates[df.Beds.isnull()].to_numpy().res
```

```
a = 0
for i in df.Beds[df.Beds.isnull()].index:
    df.at[i,'Beds'] = pred_beds[a]
    a+=1
```

Now I will handle the missing value in review score. I believe review score is of vital importance in terms of the choosing decision. So I will drop missing value in the training set and replace the missing value with the overall mean in the testing set

In [62]:
```
df.isnull().sum()
```

Out[62]:
```
Decision                  2440
Host_response_time           0
Host_is_superhost            0
Host_has_profile_pic         0
Host_identity_verified       0
Neighbourhood                0
Property_type                0
Room_type                    0
Accommodates                 0
Bathrooms_text               0
Bedrooms                     0
Beds                         0
Essentials                   0
Cooking                      0
Balcony                      0
Parking                      0
Price                        0
Number_of_reviews            0
Review_scores_rating       669
Instant_bookable             0
Month                        0
dtype: int64
```

In [63]:
```
# Record of number of will be dropped in the training set
train_idx = df[:len(df1)].isnull().sum()['Review_scores_rating']
```

In [64]:
```
# Fill in missing review score with the mean
df["Review_scores_rating"][len(df1):] = df[len(df1):]["Review_scores_rating"].fi
```

```
<ipython-input-64-761f1468e230>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  df["Review_scores_rating"][len(df1):] = df[len(df1):]["Review_scores_rating"].
fillna(df.dropna().Review_scores_rating.mean())
```

In [65]:
```
# Index for training set
train_index = len(df1)-train_idx
train_index
train_label = df.Decision[:train_index]
# There will be 7076 samples in our training set
```

In [66]:
```
# Drop sample with missing review score in the training set
df = df.drop(columns = 'Decision').dropna()
```

```
df.shape
```

Out[66]: (9516, 20)

In [67]:
```
# Please ignore
#df_no_encode = pd.concat([df1,df2],axis = 0,ignore_index=True)
#print(df1.shape, df2.shape)

#PriceValue=[]
#for c in df_no_encode.Price:
    #PriceValue.append(float(c[1:].strip().replace(',', '')))
#df_no_encode['Price'] = np.array(PriceValue)

#df_no_encode.drop(columns = 'Host_response_time')

#a = 0
#for i in df_no_encode.Bedrooms[df.Bedrooms.isnull()].index:
   # df_no_encode.at[i,'Bedrooms'] = pred_bdrms[a]
   # a+=1

#a = 0
#for i in df_no_encode.Beds[df.Beds.isnull()].index:
 #   df_no_encode.at[i,'Beds'] = pred_beds[a]
  #  a+=1

#df_no_encode["Review_scores_rating"] = df_no_encode["Review_scores_rating"].fil

#df_no_encode.fillna({'Host_is_superhost':'t'}, inplace=True)
#df_no_encode.fillna({'Host_has_profile_pic':'t'}, inplace=True)
#df_no_encode.fillna({'Host_identity_verified':'t'}, inplace=True)
```

Exploring and Visualizing Data

Exploring the data by analyzing its statistics and visualizing the values of features and correlations between different features. Explaining the process and the results Now that we are ready for an exploration of our data, we can make a rule that we are going to be working from left to right. The reason some may prefer to do this is due to its set approach - some datasets have a big number of attributes, plus this way we will remember to explore each column individually to make sure we learn as much as we can about our dataset.

Now we will do histogram for the variables, then we will do the variable-wise encoding and outlier wiping. Note that we will do outliers wiping at last, since the df now contains both training sets with label and testing set without lable. We only want to remove outliers that lie in training set.

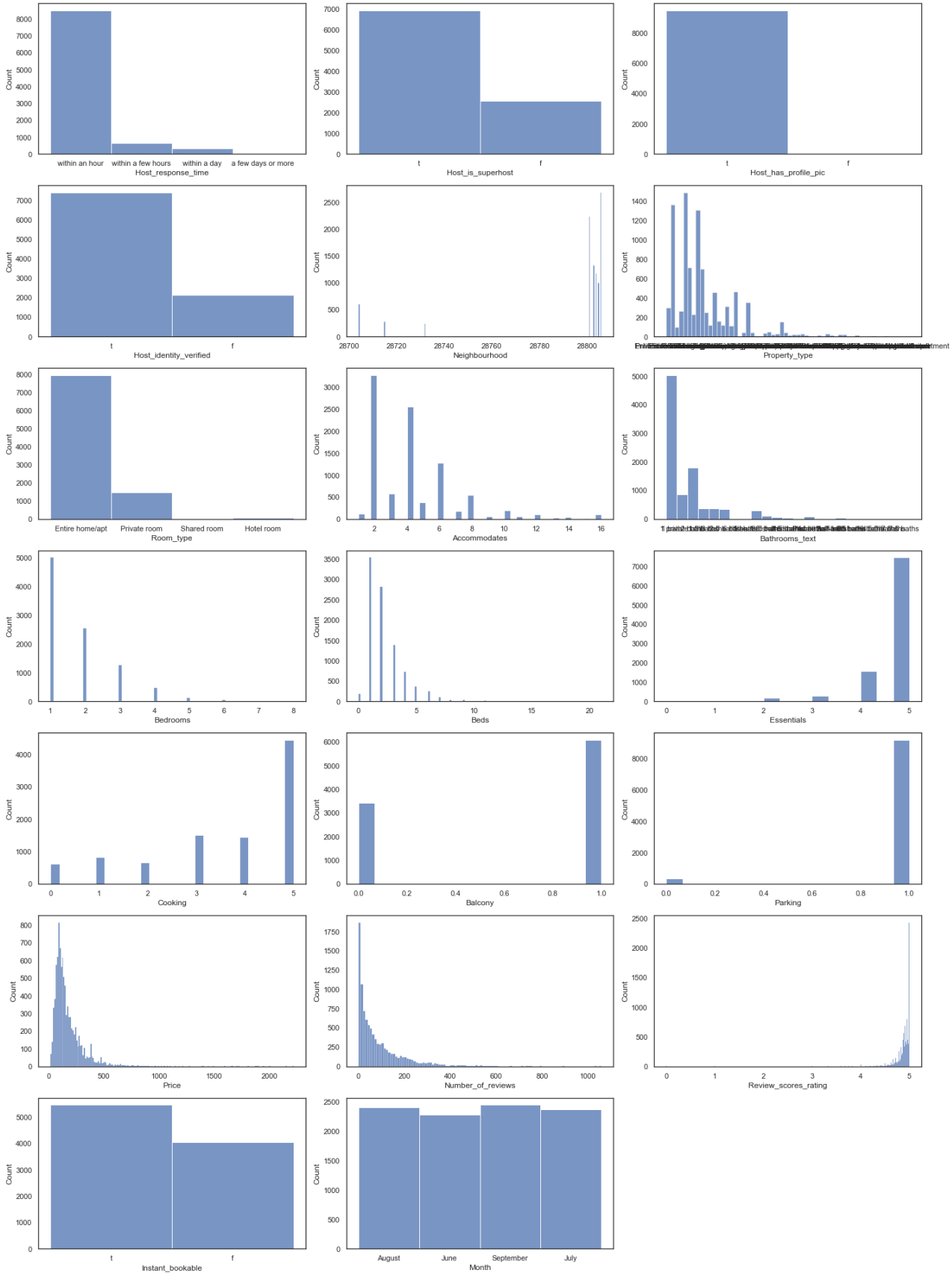
In [68]:
```
df.columns
```

Out[68]:
```
Index(['Host_response_time', 'Host_is_superhost', 'Host_has_profile_pic',
       'Host_identity_verified', 'Neighbourhood', 'Property_type', 'Room_type',
       'Accommodates', 'Bathrooms_text', 'Bedrooms', 'Beds', 'Essentials',
       'Cooking', 'Balcony', 'Parking', 'Price', 'Number_of_reviews',
```

```
       'Review_scores_rating', 'Instant_bookable', 'Month'],
      dtype='object')
```

In [24]:

```
c = 1
nrows = round(len(df.columns) / 3) + 1
fig = plt.figure(figsize=(20,30))
for i in df.columns:
    plt.subplot(nrows, 3, c)
    sns.histplot(df[i])
    c += 1
plt.tight_layout()
plt.show()
```



Handle Binary variables:

Note that four variables are in binary form, and one of them ('Host_has_profile_pic') has only 15 'f' but 9896 't' after our handeling of missing data. So I will drop this variable and encode other three of them.

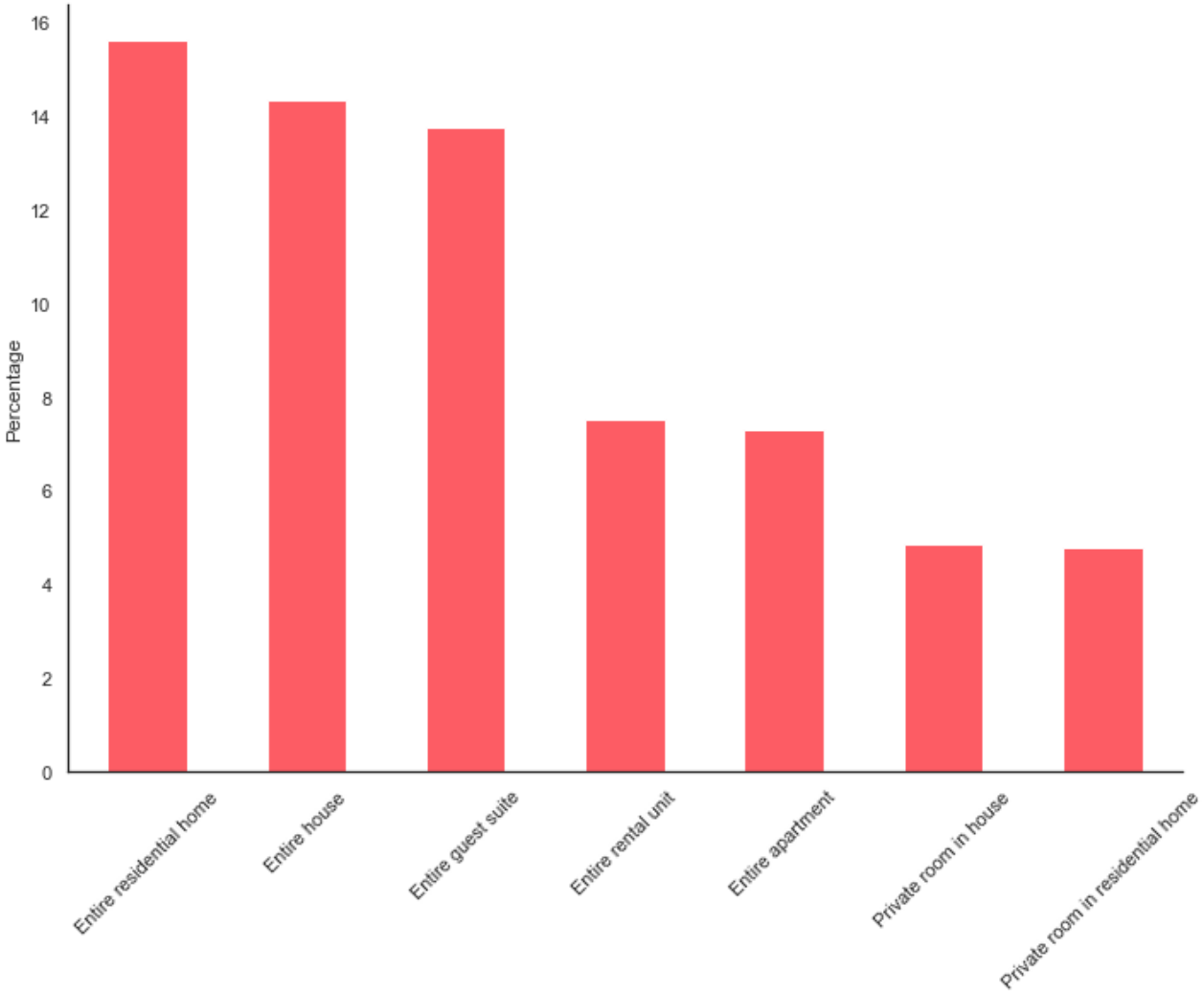
In [69]:
```
# Using 0-1 to denote false or ture
df['Host_is_superhost'] = df['Host_is_superhost'].map({'f':0, 't':1})
df['Host_identity_verified'] = df['Host_identity_verified'].map({'f':0, 't':1})
df['Instant_bookable'] = df['Instant_bookable'].map({'f':0, 't':1})
df['Host_has_profile_pic'] = df['Host_has_profile_pic'].map({'f':0, 't':1})
```

We now turn to encode the left categorical variables, which have multiple outcomes, we will use one-hot encoding. Note that for some of the variables, we don't cover all subsets. For example, 'Property_type' has 58 unique types, it is not meaningful to cover each of them with a dummy variable. Instead, I will make some of the most frequent types dummies and see the others as 'others'. It is convineint to practice since to avoid multicolinearity, I do not have to specify the 'others' specifically.

In [70]:
```
df.Property_type.describe()
```

Out[70]:
```
count                         9516
unique                          58
top       Entire residential home
freq                          1487
Name: Property_type, dtype: object
```

In [71]:
```
proptype_percentage = df.Property_type.value_counts() / df.shape[0] * 100
proptype_percentage2 = proptype_percentage[proptype_percentage>4]
proptype_percentage2.plot(kind='bar',color='#FD5C64', rot=45)
plt.ylabel('Percentage')
sns.despine()
```



Including these 7 categories as dummies will cover about 70 percent of all the cases.

In [72]:
```
# Property_type
dummies_porperty_type = pd.get_dummies(df['Property_type'], prefix= 'is' )
df['is_Entire residential home'] = dummies_porperty_type['is_Entire residential
df['is_Entire house'] = dummies_porperty_type['is_Entire house']
df['is_Entire guest suite'] = dummies_porperty_type['is_Entire guest suite']
df['is_Entire rental unit'] = dummies_porperty_type['is_Entire rental unit']
df['is_Entire apartment'] = dummies_porperty_type['is_Entire apartment']
df['is_Private room in house'] = dummies_porperty_type['is_Private room in house
df['is_Private room in residential home'] = dummies_porperty_type['is_Private ro
```

In [ ]:
```
```

In [73]:
```
# Host_response_time
dummies_response_time = pd.get_dummies(df['Host_response_time'], prefix= 'is' )
df['is_within an hour'] = dummies_response_time['is_within an hour']
df['is_within a few hours'] = dummies_response_time['is_within a few hours']
```

In [74]:
```
# Neighbourhood
```

In [75]:
```
dummies_neighbour = pd.get_dummies(df['Neighbourhood'], prefix= 'is' )
df['is_28805'] = dummies_neighbour['is_28805']
df['is_28804'] = dummies_neighbour['is_28804']
df['is_28806'] = dummies_neighbour['is_28806']
df['is_28803'] = dummies_neighbour['is_28803']
df['is_28704'] = dummies_neighbour['is_28704']
df['is_28801'] = dummies_neighbour['is_28801']
```

In [76]:
```
# Room_type
```

In [77]:
```
room_type_percentage = df.Room_type.value_counts() / df.shape[0] * 100
room_type_percentage
```

Out[77]:
```
Entire home/apt    83.732661
Private room       15.384615
Hotel room          0.546448
Shared room         0.336276
Name: Room_type, dtype: float64
```

In [78]:
```
dummies_room_type = pd.get_dummies(df['Room_type'], prefix= 'is' )
df['is_Entire home/apt'] = dummies_room_type['is_Entire home/apt']
df['is_Private room'] = dummies_room_type['is_Private room']
```

In [79]:
```
# Bathrooms_text
```

In [80]:
```
btrmtext_percentage = df.Bathrooms_text.value_counts() / df.shape[0] * 100
btrmtext_percentage
```

Out[80]:
```
1 bath               52.900378
2 baths              18.883985
1 private bath        8.984868
3 baths               3.898697
1.5 baths             3.804119
2.5 baths             3.741068
1 shared bath         3.142077
3.5 baths             1.208491
4 baths               0.903741
1.5 shared baths      0.567465
2 shared baths        0.409836
4.5 baths             0.346784
2.5 shared baths      0.252207
5 baths               0.252207
0 baths               0.157629
6 baths               0.126103
Half-bath             0.115595
5.5 baths             0.094578
6.5 baths             0.063052
0 shared baths        0.052543
Private half-bath     0.042034
7 baths               0.042034
7.5 baths             0.010509
Name: Bathrooms_text, dtype: float64
```

In [81]:
```
dummies_btrmtext = pd.get_dummies(df['Bathrooms_text'], prefix= 'is' )
df['is_1 bath'] = dummies_btrmtext['is_1 bath']
```

```
df['is_2 baths'] = dummies_btrmtext['is_2 baths']
df['is_1 private bath'] = dummies_btrmtext['is_1 private bath']
df['is_3 baths'] = dummies_btrmtext['is_3 baths']
df['is_1.5 baths'] = dummies_btrmtext['is_1.5 baths']
df['is_2.5 baths'] = dummies_btrmtext['is_2.5 baths']
```

In [82]:
```
# Month
dummies_month = pd.get_dummies(df['Month'], prefix= 'is' )
df['is_August'] = dummies_month['is_August']
df['is_July'] = dummies_month['is_July']
df['is_June'] = dummies_month['is_June']
```

In [83]:
```
# Drop the original categorical data
df_final = df.drop(['Host_has_profile_pic','Property_type', 'Month','Bathrooms_t
```

In [84]:
```
X_raw, Z_raw= df_final[:train_index],df_final[train_index:]

# Add Decisions back
#df_train = pd.read_csv('train.csv', index_col=False)
#label = df_train['Decision'].to_numpy()
#X_raw.loc[:,'Decision'] = pd.Series(train_label, index = X_raw.index)
```

In [85]:
```
df_final.columns
```

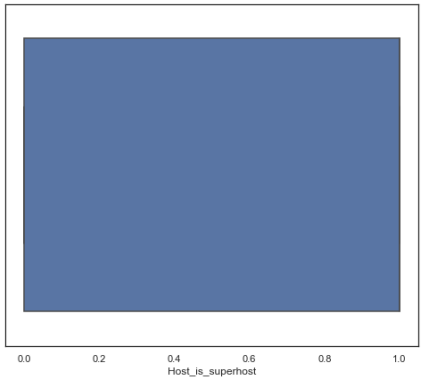
Out[85]:
```
Index(['Host_is_superhost', 'Host_identity_verified', 'Accommodates',
       'Bedrooms', 'Beds', 'Essentials', 'Cooking', 'Balcony', 'Parking',
       'Price', 'Number_of_reviews', 'Review_scores_rating',
       'Instant_bookable', 'is_Entire residential home', 'is_Entire house',
       'is_Entire guest suite', 'is_Entire rental unit', 'is_Entire apartment',
       'is_Private room in house', 'is_Private room in residential home',
       'is_within an hour', 'is_within a few hours', 'is_28805', 'is_28804',
       'is_28806', 'is_28803', 'is_28704', 'is_28801', 'is_Entire home/apt',
       'is_Private room', 'is_1 bath', 'is_2 baths', 'is_1 private bath',
       'is_3 baths', 'is_1.5 baths', 'is_2.5 baths', 'is_August', 'is_July',
       'is_June'],
      dtype='object')
```

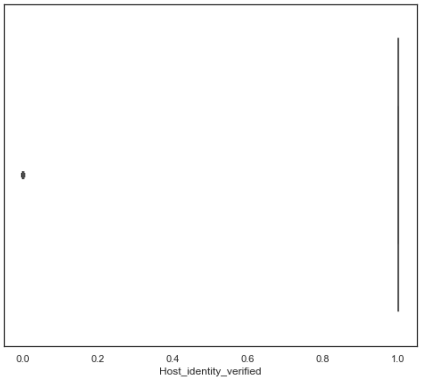
In [ ]:
```

```

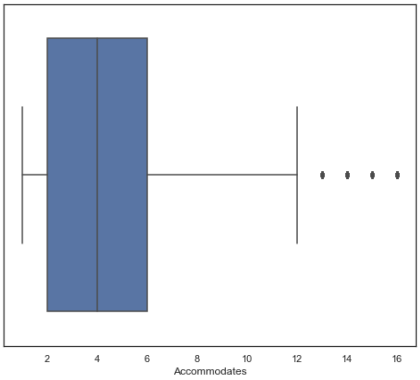
Drop the outliers and split the training and testing data

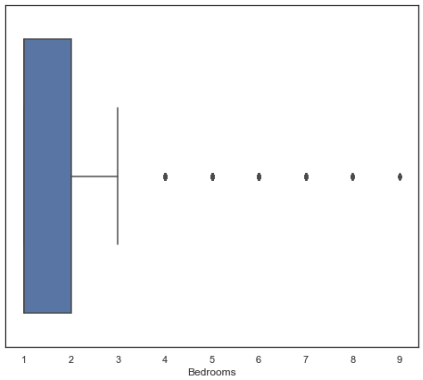
In [97]:
```
# visualizing the outliers

import warnings
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import seaborn as sns

var_num = df_final.select_dtypes(include=['float', 'int']).columns
```

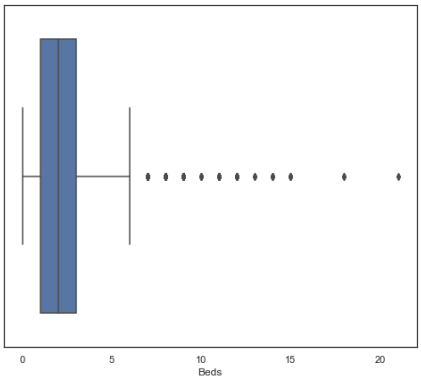
```
c = 1
nrows = round(len(var_num) / 3) + 1
fig = plt.figure(figsize=(20,30))
for i in var_num:
    plt.subplot(nrows, 3, c)
    sns.boxplot(df[i])
    c += 1
plt.tight_layout()
plt.show()
```

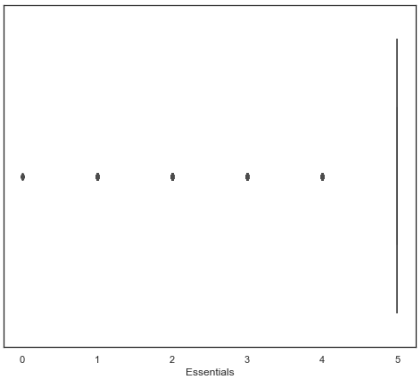

0.0 0.2 0.4 0.6 0.8 1.0
Host_is_superhost

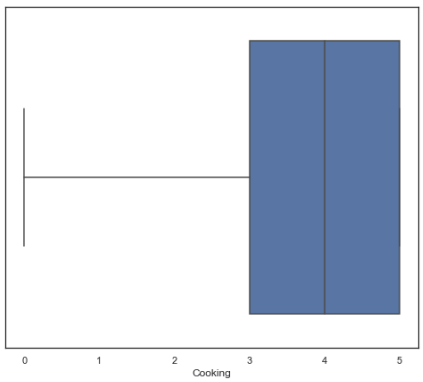

0.0 0.2 0.4 0.6 0.8 1.0
Host_identity_verified

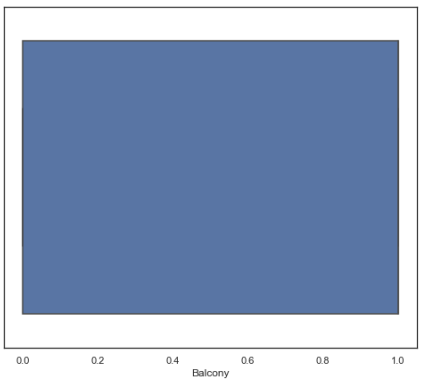

2 4 6 8 10 12 14 16
Accommodates

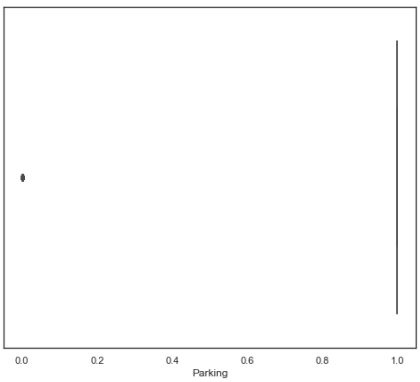

1 2 3 4 5 6 7 8 9
Bedrooms

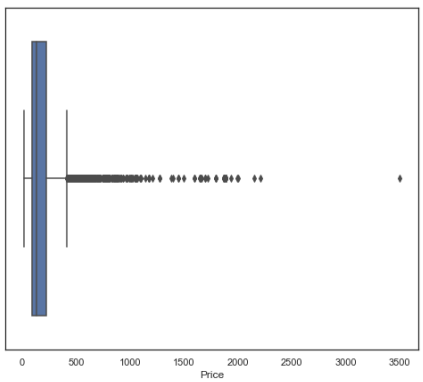

0 5 10 15 20
Beds

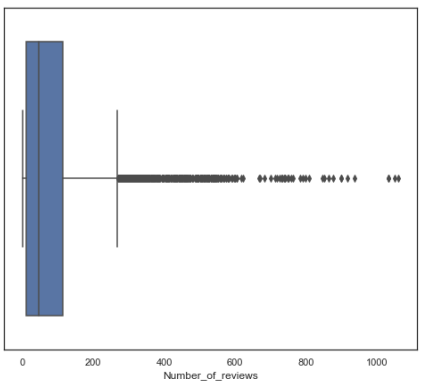

0 1 2 3 4 5
Essentials

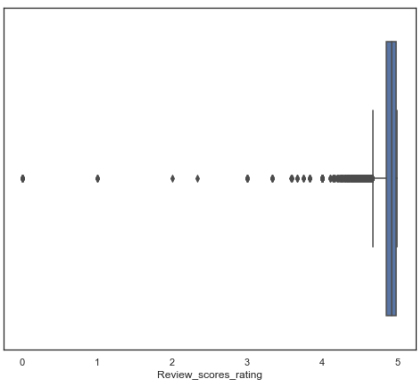

0 1 2 3 4 5
Cooking

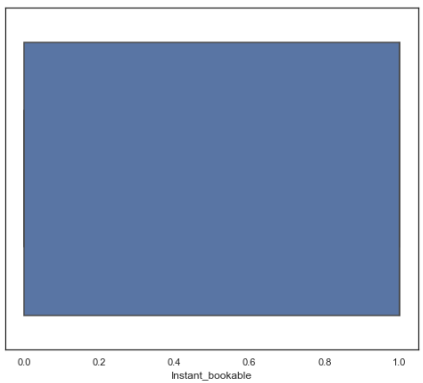

0.0 0.2 0.4 0.6 0.8 1.0
Balcony


0.0 0.2 0.4 0.6 0.8 1.0
Parking


0 500 1000 1500 2000 2500 3000 3500
Price


0 200 400 600 800 1000
Number_of_reviews


0 1 2 3 4 5
Review_scores_rating


0.0 0.2 0.4 0.6 0.8 1.0
Instant_bookable

Now we are left with 10 numerical variables untreated. In the following section, we will drop the outliers in terms of these variables. Specifically, we will drop outliers in Price, Review_Scores and Number of reviews. Samples in the training set with extremely high price, low rating score and high number of reviews will be dropped.

Drop the outliers

In [90]:
```
Xdf = X_raw.drop(X_raw[(df.Price>500) | (df.Review_scores_rating<4) | (df.Number
```

```
<ipython-input-90-65fd2558d804>:1: UserWarning: Boolean Series key will be reind
exed to match DataFrame index.
  Xdf = X_raw.drop(X_raw[(df.Price>500) | (df.Review_scores_rating<4) | (df.Numb
er_of_reviews>500)].index)
```

In [91]:
```
Xdf.shape
```

Out[91]:
```
(6768, 39)
```

Rescaling the data for SVM with Gaussian Kernel

In [93]:
```
X = Xdf.to_numpy()
y = train_label
Z = Z_raw.to_numpy()
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2, random_stat
```

In [215…
```
ytest.shape
```

Out[215…
```
(1406,)
```

Logistic regression

Logistic regression does not really have any critical hyperparameters to tune. Sometimes, we can see useful differences in performance or convergence with different solvers (solver).

In [216…
```
# I conduct a grid searching key hyperparametres for logistic regression
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
# define models and parameters
model = LogisticRegression()
solvers = ['newton-cg', 'lbfgs', 'liblinear']
penalty = ['l2','l1','elasticnet']
c_values = [100, 10, 1.0, 0.1, 0.01]
# define grid search
grid = dict(solver=solvers,penalty=penalty,C=c_values)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, s
grid_result = grid_search.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
```

```
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.696450 using {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}
0.696450 (0.014534) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}
0.692134 (0.012961) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}
0.695454 (0.014738) with: {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'l1', 'solver': 'lbfgs'}
0.695264 (0.014683) with: {'C': 100, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'elasticnet', 'solver': 'newton-
cg'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'elasticnet', 'solver': 'lbfgs'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'elasticnet', 'solver': 'libline
ar'}
0.696308 (0.014600) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
0.691232 (0.012271) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
0.694980 (0.014103) with: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'l1', 'solver': 'lbfgs'}
0.694838 (0.014249) with: {'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'elasticnet', 'solver': 'newton-c
g'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'elasticnet', 'solver': 'lbfgs'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'elasticnet', 'solver': 'liblinea
r'}
0.696213 (0.014560) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.690664 (0.012130) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.694174 (0.013383) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'lbfgs'}
0.693700 (0.013408) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'elasticnet', 'solver': 'newton-
cg'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'elasticnet', 'solver': 'lbfgs'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'elasticnet', 'solver': 'libline
ar'}
0.694980 (0.014363) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.692466 (0.013192) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.695074 (0.011970) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'lbfgs'}
0.691660 (0.012511) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'elasticnet', 'solver': 'newton-
cg'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'elasticnet', 'solver': 'lbfgs'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'elasticnet', 'solver': 'libline
ar'}
0.691469 (0.011392) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
0.691137 (0.011147) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.691564 (0.011014) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'lbfgs'}
0.672975 (0.011306) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'elasticnet', 'solver': 'newton
-cg'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'elasticnet', 'solver': 'lbfg
s'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'elasticnet', 'solver': 'liblin
ear'}
```

In [797…

```
# Try training on normalized data
```

```
# define models and parameters
model = LogisticRegression()
solvers = ['newton-cg', 'lbfgs', 'liblinear']
penalty = ['l2','l1','elasticnet']
c_values = [100, 10, 1.0, 0.1, 0.01]
# define grid search
grid = dict(solver=solvers,penalty=penalty,C=c_values)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, s
grid_result = grid_search.fit(Xdf_norm.drop(columns = ['Decision']).to_numpy(),
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.696545 using {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
0.696355 (0.014451) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}
0.696355 (0.014451) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}
0.696355 (0.014451) with: {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'l1', 'solver': 'lbfgs'}
0.696308 (0.014461) with: {'C': 100, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'elasticnet', 'solver': 'newton-
cg'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'elasticnet', 'solver': 'lbfgs'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'elasticnet', 'solver': 'libline
ar'}
0.696307 (0.014119) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
0.696355 (0.014356) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
0.696545 (0.014154) with: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'l1', 'solver': 'lbfgs'}
0.696165 (0.014455) with: {'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'elasticnet', 'solver': 'newton-c
g'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'elasticnet', 'solver': 'lbfgs'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'elasticnet', 'solver': 'liblinea
r'}
0.696307 (0.014370) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.696260 (0.014315) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.696165 (0.014713) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'lbfgs'}
0.696497 (0.014907) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'elasticnet', 'solver': 'newton-
cg'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'elasticnet', 'solver': 'lbfgs'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'elasticnet', 'solver': 'libline
ar'}
0.686633 (0.011658) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.686633 (0.011658) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.686348 (0.011546) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'lbfgs'}
0.685020 (0.011934) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'elasticnet', 'solver': 'newton-
cg'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'elasticnet', 'solver': 'lbfgs'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'elasticnet', 'solver': 'libline
ar'}
```

```
0.668233 (0.009607) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
0.668233 (0.009607) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.668138 (0.009708) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'lbfgs'}
0.652202 (0.008149) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'elasticnet', 'solver': 'newton
-cg'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'elasticnet', 'solver': 'lbfg
s'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'elasticnet', 'solver': 'liblin
ear'}
```

It seems that logistic regression, though intuitive and direct, doesn't provides us with a satisfying accuracy in this binary classification.

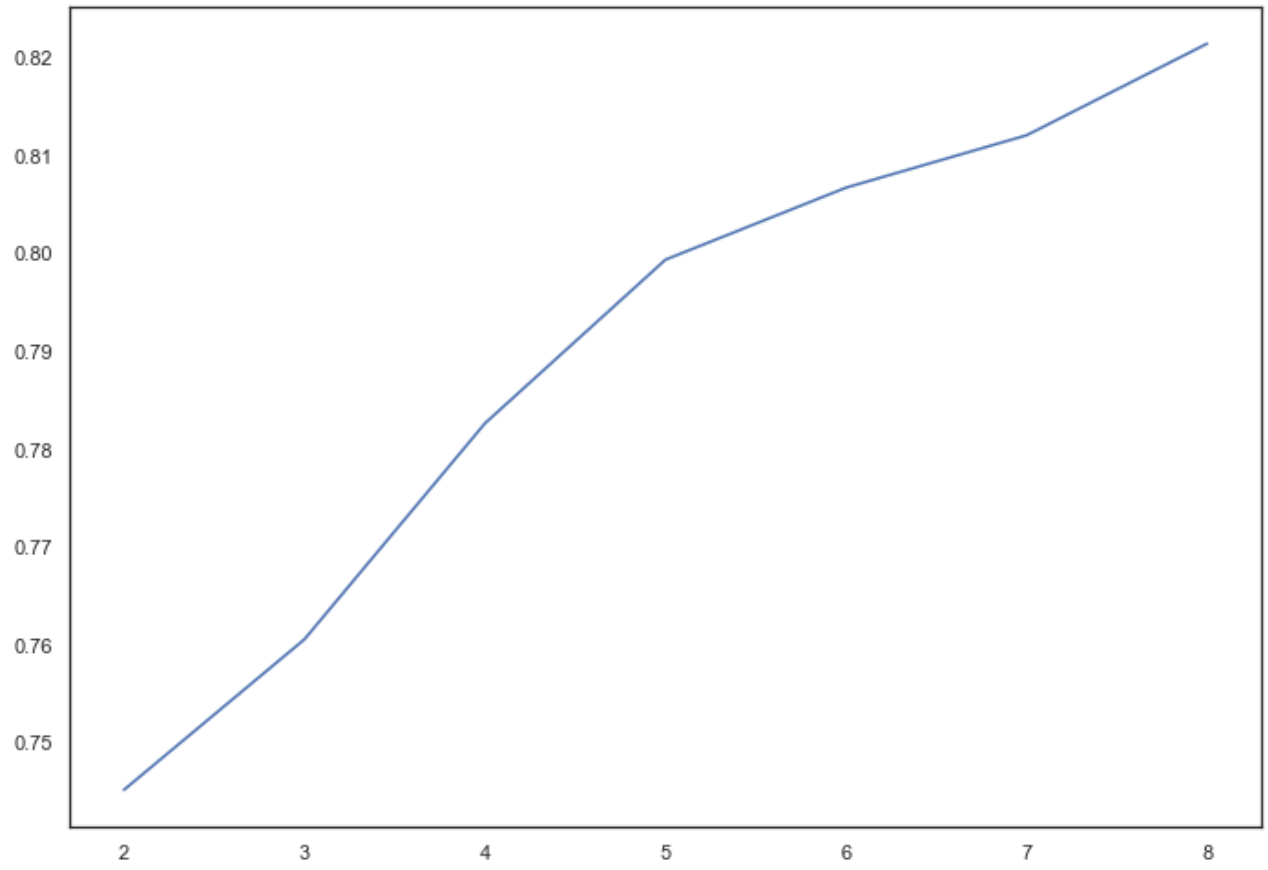
Catboost Part 1 - A simple wonder & Visualization

Let's fist have some fun with Catboost with a tune parameter on the tree depth

In [98]:
```
from catboost import CatBoostClassifier
depth_list = [2, 3, 4, 5, 6, 7, 8]
acc_score_list = []
for depth in depth_list:
    catbst = CatBoostClassifier(iterations = 8000,
                                learning_rate = 0.005,
                                depth = depth,
                                loss_function='Logloss',
                                leaf_estimation_method = 'Gradient',
                                eval_metric = 'Accuracy',
                                custom_metric = 'Accuracy',
                                use_best_model = True,
                                random_seed = 76, verbose = False)
    catbst.fit(Xtrain, ytrain, eval_set=(Xtest, ytest), plot=True)
    y_predict = catbst.predict(Xtest)
    acc_score_list.append(accuracy_score(ytest, y_predict))
    cm = confusion_matrix(ytest, y_predict)
    print(cm)
```

In [857…
```
plt.plot(depth_list, acc_score_list)
```

Out[857… `[<matplotlib.lines.Line2D at 0x7fc1db8f3d60>]`



Catboost Part 2 - hyperparameter tuning

In [95]:

```
# define a function for parameter tuning
# Credit: This function is derived from Competition Notebook: https://www.kaggle

def objective(trial):
    param = {
        "loss_function": trial.suggest_categorical("loss_function", ['Logloss',
        "learning_rate": trial.suggest_loguniform("learning_rate", 1e-5, 1e0),
        "l2_leaf_reg": trial.suggest_loguniform("l2_leaf_reg", 1e-2, 1e0),
        "colsample_bylevel": trial.suggest_float("colsample_bylevel", 0.01, 0.1)
        "depth": trial.suggest_int("depth", 1, 10),
        "boosting_type": trial.suggest_categorical("boosting_type", ["Ordered",
        "bootstrap_type": trial.suggest_categorical("bootstrap_type", ["Bayesian
        "min_data_in_leaf": trial.suggest_int("min_data_in_leaf", 2, 20),
        "one_hot_max_size": trial.suggest_int("one_hot_max_size", 2, 20),
        'random_state': trial.suggest_categorical('random_state', [6, 9, 12, 15]
    }
    # Conditional Hyper-Parameters
    if param["bootstrap_type"] == "Bayesian":
        param["bagging_temperature"] = trial.suggest_float("bagging_temperature"
    elif param["bootstrap_type"] == "Bernoulli":
        param["subsample"] = trial.suggest_float("subsample", 0.1, 1)

    catboost = CatBoostClassifier(**param)
    catboost.fit(Xtrain, ytrain, eval_set=[(Xtest, ytest)], verbose=0, early_sto
    y_pred = catboost.predict(Xtest)
    score = mean_squared_error(ytest, y_pred, squared=False)
    return score
```

```
ata_in_leaf': 10, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.9838
304401710416}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:38:58,608] Trial 1709 finished with value: 0.44466171392340825
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0210554310
05655895, 'l2_leaf_reg': 0.0786310173978312, 'colsample_bylevel': 0.096110453608
8176, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_i
n_leaf': 10, 'one_hot_max_size': 2, 'random_state': 6}. Best is trial 1708 with
 value: 0.4308514841827461.
[I 2021-12-09 09:38:59,404] Trial 1710 finished with value: 0.458055758872357 an
d parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0959678026601
0148, 'l2_leaf_reg': 0.06343194025807607, 'colsample_bylevel': 0.077681154863431
37, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_da
ta_in_leaf': 10, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.96700
69093865207}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:00,289] Trial 1711 finished with value: 0.4710691208760756 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.119568196958
45403, 'l2_leaf_reg': 0.10156446723329157, 'colsample_bylevel': 0.08593965274175
058, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 9, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.70988
97431832616}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:01,421] Trial 1712 finished with value: 0.45180214453827944
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.1495226923
746806, 'l2_leaf_reg': 0.7454626353900002, 'colsample_bylevel': 0.09671805968750
798, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in
_leaf': 10, 'one_hot_max_size': 14, 'random_state': 6}. Best is trial 1708 with
 value: 0.4308514841827461.
[I 2021-12-09 09:39:02,282] Trial 1713 finished with value: 0.46575471416783215
 and parameters: {'loss_function': 'Logloss', 'learning_rate': 0.152671157396777
14, 'l2_leaf_reg': 0.05696147462787618, 'colsample_bylevel': 0.0827745391862787
9, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_dat
a_in_leaf': 10, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.995266
7997458591}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:02,917] Trial 1714 finished with value: 0.49607277032368396
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.2893581350
693737, 'l2_leaf_reg': 0.02903821251128103, 'colsample_bylevel': 0.0932261837757
2725, 'depth': 10, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_
in_leaf': 18, 'one_hot_max_size': 20, 'random_state': 12}. Best is trial 1708 wi
th value: 0.4308514841827461.
[I 2021-12-09 09:39:03,626] Trial 1715 finished with value: 0.461150765422223 an
d parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.1668609103840
8857, 'l2_leaf_reg': 0.031371057110739634, 'colsample_bylevel': 0.09467816401854
26, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_da
ta_in_leaf': 10, 'one_hot_max_size': 20, 'random_state': 15, 'subsample': 0.9243
850638399673}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:04,236] Trial 1716 finished with value: 0.46803970603954903
 and parameters: {'loss_function': 'Logloss', 'learning_rate': 0.135859266158194
82, 'l2_leaf_reg': 0.06727878182911756, 'colsample_bylevel': 0.0973421163424167
5, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_dat
a_in_leaf': 10, 'one_hot_max_size': 19, 'random_state': 6, 'subsample': 0.887374
7477857417}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:05,434] Trial 1717 finished with value: 0.45180214453827944
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0848664479
9526862, 'l2_leaf_reg': 0.0694578640820673, 'colsample_bylevel': 0.0846356646728
0771, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_
data_in_leaf': 9, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.9999
464539581585}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:07,217] Trial 1718 finished with value: 0.4414511180367898 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.032660282600
20565, 'l2_leaf_reg': 0.01657162988834609, 'colsample_bylevel': 0.09798540202627
326, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 10, 'one_hot_max_size': 16, 'random_state': 6, 'subsample': 0.9137
197386778706}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:07,900] Trial 1719 finished with value: 0.4718234354038612 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.167059856436
66849, 'l2_leaf_reg': 0.09784694783563411, 'colsample_bylevel': 0.09698609062039
```

```
037, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 10, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.9709
535454155265}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:10,806] Trial 1720 finished with value: 0.46803970603954903
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0147532796
08098968, 'l2_leaf_reg': 0.07028957946604064, 'colsample_bylevel': 0.09830760851
905353, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bayesian', 'min
_data_in_leaf': 10, 'one_hot_max_size': 16, 'random_state': 6, 'bagging_temperat
ure': 4.568431589059424}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:13,754] Trial 1721 finished with value: 0.44784929389547357
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0187488152
70039715, 'l2_leaf_reg': 0.020149488527973023, 'colsample_bylevel': 0.0932179127
5784841, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_dat
a_in_leaf': 10, 'one_hot_max_size': 16, 'random_state': 6}. Best is trial 1708 w
ith value: 0.4308514841827461.
[I 2021-12-09 09:39:19,169] Trial 1722 finished with value: 0.45727873412480935
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0391777886
03771066, 'l2_leaf_reg': 0.026327683970863828, 'colsample_bylevel': 0.0946581443
9421226, 'depth': 8, 'boosting_type': 'Ordered', 'bootstrap_type': 'MVS', 'min_d
ata_in_leaf': 9, 'one_hot_max_size': 20, 'random_state': 12}. Best is trial 1708
with value: 0.4308514841827461.
[I 2021-12-09 09:39:19,864] Trial 1723 finished with value: 0.45960586739381004
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.1266664967
08297, 'l2_leaf_reg': 0.12163645102273409, 'colsample_bylevel': 0.08205761240305
715, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 10, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.9999
657683045646}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:20,998] Trial 1724 finished with value: 0.45415734109382927
 and parameters: {'loss_function': 'Logloss', 'learning_rate': 0.051309472067510
95, 'l2_leaf_reg': 0.03792842620975192, 'colsample_bylevel': 0.0867091359768270
1, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_dat
a_in_leaf': 9, 'one_hot_max_size': 12, 'random_state': 12, 'subsample': 0.871770
5927033179}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:21,658] Trial 1725 finished with value: 0.4470545296597848 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.2356437680405183
1, 'l2_leaf_reg': 0.5360376147588897, 'colsample_bylevel': 0.09872902298830195,
 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_
in_leaf': 9, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.87460896
1265663}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:22,167] Trial 1726 finished with value: 0.5031903903991437 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.198823787160
4184, 'l2_leaf_reg': 0.03691704140689254, 'colsample_bylevel': 0.092447484648159
7, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_l
eaf': 12, 'one_hot_max_size': 19, 'random_state': 6}. Best is trial 1708 with va
lue: 0.4308514841827461.
[I 2021-12-09 09:39:23,157] Trial 1727 finished with value: 0.4634584567245813 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.099078187809
60544, 'l2_leaf_reg': 0.06309529316892717, 'colsample_bylevel': 0.08056406342402
65, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_da
ta_in_leaf': 11, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.99851
34899113668}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:23,991] Trial 1728 finished with value: 0.4533736350032652 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.188280593342
9441, 'l2_leaf_reg': 0.07247405802431779, 'colsample_bylevel': 0.082882623395096
07, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_da
ta_in_leaf': 10, 'one_hot_max_size': 4, 'random_state': 6, 'subsample': 0.961335
4397912169}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:24,663] Trial 1729 finished with value: 0.4733284581588003 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.142693380292
83543, 'l2_leaf_reg': 0.08205881789089034, 'colsample_bylevel': 0.08007382882551
228, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 10, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.5047
185008450753}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:25,586] Trial 1730 finished with value: 0.4687988950316866 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.324729154100
```

```
00475, 'l2_leaf_reg': 0.7496185558357933, 'colsample_bylevel': 0.099938937189398
99, 'depth': 10, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 5, 'one_hot_max_size': 13, 'random_state': 6, 'subsample': 0.95047
11087756715}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:26,392] Trial 1731 finished with value: 0.5485011258548257 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.025761833544
02464, 'l2_leaf_reg': 0.9254781225374352, 'colsample_bylevel': 0.016084269585977
307, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 11, 'one_hot_max_size': 14, 'random_state': 6, 'subsample': 0.9756
210389260714}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:28,755] Trial 1732 finished with value: 0.5074130411312855 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.000783041569
7746326, 'l2_leaf_reg': 0.01958747333139797, 'colsample_bylevel': 0.083331043809
56705, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min
_data_in_leaf': 9, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.96
50892058985905}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:29,525] Trial 1733 finished with value: 0.48225997132573734
 and parameters: {'loss_function': 'Logloss', 'learning_rate': 0.234874704071138
53, 'l2_leaf_reg': 0.032324454921913194, 'colsample_bylevel': 0.0938636295617551
7, 'depth': 10, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_
leaf': 12, 'one_hot_max_size': 20, 'random_state': 12}. Best is trial 1708 with
 value: 0.4308514841827461.
[I 2021-12-09 09:39:32,493] Trial 1734 finished with value: 0.44784929389547357
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0184554169
33051763, 'l2_leaf_reg': 0.02407714808458731, 'colsample_bylevel': 0.09040706099
605945, 'depth': 10, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'm
in_data_in_leaf': 10, 'one_hot_max_size': 16, 'random_state': 12, 'subsample':
 0.9479951734308099}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:33,241] Trial 1735 finished with value: 0.4422559521015795 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.2515576824976503,
'l2_leaf_reg': 0.050244996217576963, 'colsample_bylevel': 0.09712113622935366,
 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_
in_leaf': 4, 'one_hot_max_size': 17, 'random_state': 6, 'subsample': 0.924270680
1330003}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:35,702] Trial 1736 finished with value: 0.524641718836455 an
d parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 3.3759643776550
074e-05, 'l2_leaf_reg': 0.03183861285490761, 'colsample_bylevel': 0.090318618877
35144, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min
_data_in_leaf': 11, 'one_hot_max_size': 15, 'random_state': 12, 'subsample': 0.9
784565158060944}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:37,535] Trial 1737 finished with value: 0.4510143459537019 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.045631446968
7996, 'l2_leaf_reg': 0.03545907228609139, 'colsample_bylevel': 0.090940212521748
05, 'depth': 10, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 12, 'one_hot_max_size': 20, 'random_state': 12, 'subsample': 0.936
659604647531}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:40,363] Trial 1738 finished with value: 0.4462583499902143 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.021594806772
377417, 'l2_leaf_reg': 0.05819050083205539, 'colsample_bylevel': 0.0984978312384
3384, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_
data_in_leaf': 20, 'one_hot_max_size': 14, 'random_state': 6, 'subsample': 0.957
3378026452546}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:43,103] Trial 1739 finished with value: 0.45883146774112354
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0127393477
81317264, 'l2_leaf_reg': 0.06887789800923706, 'colsample_bylevel': 0.09778079480
31888, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min
_data_in_leaf': 5, 'one_hot_max_size': 20, 'random_state': 15, 'subsample': 0.93
41384093754443}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:43,938] Trial 1740 finished with value: 0.45572071005836723
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.1036122322
6703788, 'l2_leaf_reg': 0.0881095174696684, 'colsample_bylevel': 0.0842166925297
0156, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_
data_in_leaf': 10, 'one_hot_max_size': 16, 'random_state': 6, 'subsample': 0.981
848320981712}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:46,347] Trial 1741 finished with value: 0.46499055497527714
```

```
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.1333251397
0498117, 'l2_leaf_reg': 0.023086248482312135, 'colsample_bylevel': 0.08364530923
921033, 'depth': 8, 'boosting_type': 'Ordered', 'bootstrap_type': 'MVS', 'min_da
ta_in_leaf': 10, 'one_hot_max_size': 19, 'random_state': 6}. Best is trial 1708
 with value: 0.4308514841827461.
[I 2021-12-09 09:39:48,114] Trial 1742 finished with value: 0.44784929389547357
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0506780097
510139, 'l2_leaf_reg': 0.8686970888905844, 'colsample_bylevel': 0.08929540859361
138, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 9, 'one_hot_max_size': 14, 'random_state': 12, 'subsample': 0.9606
052445102159}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:49,168] Trial 1743 finished with value: 0.46651762166385724
 and parameters: {'loss_function': 'Logloss', 'learning_rate': 0.116082718535993
36, 'l2_leaf_reg': 0.04401060344557871, 'colsample_bylevel': 0.0980409722517384
3, 'depth': 10, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_
leaf': 2, 'one_hot_max_size': 18, 'random_state': 6}. Best is trial 1708 with va
lue: 0.4308514841827461.
[I 2021-12-09 09:39:50,956] Trial 1744 finished with value: 0.445460747297274 an
d parameters: {'loss_function': 'Logloss', 'learning_rate': 0.03524911481319979,
'l2_leaf_reg': 0.028321391290745086, 'colsample_bylevel': 0.0915711935322139, 'd
epth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_leaf':
17, 'one_hot_max_size': 19, 'random_state': 12}. Best is trial 1708 with value:
 0.4308514841827461.
[I 2021-12-09 09:39:52,772] Trial 1745 finished with value: 0.45022516889074815
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0340652978
27753464, 'l2_leaf_reg': 0.020832785913769317, 'colsample_bylevel': 0.0953670485
7107445, 'depth': 10, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli',
 'min_data_in_leaf': 3, 'one_hot_max_size': 18, 'random_state': 12, 'subsample':
0.9100222465675335}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:55,449] Trial 1746 finished with value: 0.43659095764327827
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0246633345
08366164, 'l2_leaf_reg': 0.06529442472439408, 'colsample_bylevel': 0.09993502812
375556, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'mi
n_data_in_leaf': 4, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.96
92835094185931}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:57,327] Trial 1747 finished with value: 0.445460747297274 an
d parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.0219783896241
7452, 'l2_leaf_reg': 0.024666312182230306, 'colsample_bylevel': 0.08674492658213
151, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 4, 'one_hot_max_size': 16, 'random_state': 9, 'subsample': 0.96420
06865062869}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:57,943] Trial 1748 finished with value: 0.45883146774112354
 and parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.1916039788
0664506, 'l2_leaf_reg': 0.02717886462451936, 'colsample_bylevel': 0.095106144128
08677, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bayesian', 'min_
data_in_leaf': 8, 'one_hot_max_size': 20, 'random_state': 12, 'bagging_temperatu
re': 0.8029407310466432}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:39:58,857] Trial 1749 finished with value: 0.4703135965391772 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.116883942078
83636, 'l2_leaf_reg': 0.07688029119314121, 'colsample_bylevel': 0.07919512119699
726, 'depth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 11, 'one_hot_max_size': 15, 'random_state': 6, 'subsample': 0.9996
041291546145}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:40:00,441] Trial 1750 finished with value: 0.4398370317620449 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.0492658038260648,
'l2_leaf_reg': 0.0596576045304842, 'colsample_bylevel': 0.09983924156853521, 'de
pth': 9, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_in_l
eaf': 3, 'one_hot_max_size': 15, 'random_state': 12, 'subsample': 0.930745571425
7378}. Best is trial 1708 with value: 0.4308514841827461.
[I 2021-12-09 09:40:01,579] Trial 1751 finished with value: 0.46955685655336044
 and parameters: {'loss_function': 'Logloss', 'learning_rate': 0.105425555574449
64, 'l2_leaf_reg': 0.10034863924213001, 'colsample_bylevel': 0.0855620498356314,
'depth': 4, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_i
n_leaf': 3, 'one_hot_max_size': 16, 'random_state': 6, 'subsample': 0.9813650997
903051}. Best is trial 1708 with value: 0.4308514841827461.
```

```
Number of completed trials: 1752
Best trial:
        Best Score: 0.4308514841827461
        Best Params:
    loss_function: CrossEntropy
    learning_rate: 0.11636623012035487
    l2_leaf_reg: 0.06904173006404413
    colsample_bylevel: 0.08496215595673909
    depth: 9
    boosting_type: Plain
    bootstrap_type: Bernoulli
    min_data_in_leaf: 10
    one_hot_max_size: 15
    random_state: 6
    subsample: 0.9838304401710416
```

In [107…

```
catbst_for_pred = CatBoostClassifier(iterations = 2000,
                                     depth = 9,
                                     loss_function='CrossEntropy',
                                     learning_rate= 0.11636623012035487,
                                     l2_leaf_reg= 0.06904173006404413,
                                     colsample_bylevel= 0.08496215595673909,
                                     min_data_in_leaf= 10,
                                     boosting_type= 'Plain',
                                     bootstrap_type= 'Bernoulli',
                                     one_hot_max_size= 15,
                                     leaf_estimation_method = 'Gradient',
                                     eval_metric = 'Accuracy',
                                     custom_metric = 'Accuracy',
                                     random_seed = 6,
                                     verbose = False)
```

In [108…

```
catbst_for_pred.fit(X, y, plot=True)
y_predict = catbst_for_pred.predict(Z)
```

In [109…

```
catpred = pd.DataFrame({'id':idarray, 'Decision':np.squeeze(y_predict)})
catpred.to_csv('cat_pred_optuna_tuned',index=False)
```

Catboost Part 3 - Improving Catboost

By now Catboost provides me with best score compared to other algorithms. But I believe it could do better, I will do feature engineering again in a way that improves catboost score the most. I will focus on the propotion of outliers that was dropped. Previously we have dropped 500+ outliers, I think it may cause some trouble. First, Catboost, as a tree based model, is not very sensitive to outliers in terms of its accuracy. Second, the outliers may contain some important information that determines decision, for example, if a house price is very high, or a review score is very low. Then the decision may be negative. I want to find a balance between dropping and keeping outliers.

With newly engineered data, I will do the feature Optuna tuning again and see whether there's an improvement in the final score.

In [401…
```
#Collect the variables with outliers
has_outliers = ['Host_identity_verified','Accommodates','Bedrooms',
                'Beds','Essentials','Cooking','Price','Number_of_reviews',
                'Review_scores_rating']
```

Out[401… (9516, 39)

In [242…
```
# Define a function to test different rate
# where the observations above the rate will be deleted
def outliers(rate):

    train = df_final[:train_index]
    train['Decision'] = df_train.Decision
    limit = {}

    for i in has_outliers:
        limit[i] = train[i].quantile(rate)

    for i, j in limit.items():
        train = train[train[i] < j]

    X = train.drop(columns = 'Decision').to_numpy()
    y = train.Decision.to_numpy()
    Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2, random_

    catbst_for_pred = CatBoostClassifier(iterations = 1500,
                                         depth = 5,
                                         loss_function='CrossEntropy',
                                         learning_rate= 0.1,
                                         leaf_estimation_method = 'Gradient',
                                         random_seed = 6,
                                         verbose = False)
    catbst_for_pred.fit(Xtrain,ytrain)
    pred = catbst_for_pred.predict(Xtest)
    score = mean_squared_error(ytest, pred, squared=True)

    return str(rate) + ' : ' + str(score)+ ' : ' + str(df[:train_index].shape[0]
```

In [245…
```
rate = [0.8, 0.85, 0.9, 0.95, 0.98, 0.99, 1]
for i in rate:
    print(outliers(i))
```

```
0.8 : 0.25 : 7312
0.85 : 0.17142857142857143 : 7300
0.9 : 0.10256410256410256 : 7278
0.95 : 0.1346153846153846 : 7212
0.98 : 0.07547169811320754 : 7206
0.99 : 0.07407407407407407 : 7204
1 : 0.07407407407407407 : 7204
```

It seems that outliers has no effect on the prediction for catboost, so we will keep them.

In [418…
```
X_with_otlr = X_raw.to_numpy()
y_with_otlr = train_label.to_numpy()
Xtrain_otlr, Xtest_otlr, ytrain_otlr, ytest_otlr = train_test_split(X_with_otlr,
```

In [419…

```
def objective_with_otlr(trial):
    param = {
        "loss_function": trial.suggest_categorical("loss_function", ['Logloss',
        "learning_rate": trial.suggest_loguniform("learning_rate", 1e-5, 1e0),
        "l2_leaf_reg": trial.suggest_loguniform("l2_leaf_reg", 1e-2, 1e0),
        "colsample_bylevel": trial.suggest_float("colsample_bylevel", 0.01, 0.1)
        "depth": trial.suggest_int("depth", 1, 10),
        "boosting_type": trial.suggest_categorical("boosting_type", ["Ordered",
        "bootstrap_type": trial.suggest_categorical("bootstrap_type", ["Bayesian
        "min_data_in_leaf": trial.suggest_int("min_data_in_leaf", 2, 20),
        "one_hot_max_size": trial.suggest_int("one_hot_max_size", 2, 20),
        'random_state': trial.suggest_categorical('random_state', [6, 9, 12, 15]
    }
    # Conditional Hyper-Parameters
    if param["bootstrap_type"] == "Bayesian":
        param["bagging_temperature"] = trial.suggest_float("bagging_temperature"
    elif param["bootstrap_type"] == "Bernoulli":
        param["subsample"] = trial.suggest_float("subsample", 0.1, 1)

    catboost = CatBoostClassifier(**param)
    catboost.fit(Xtrain_otlr, ytrain_otlr, eval_set=[(Xtest_otlr, ytest_otlr)],
    y_pred = catboost.predict(Xtest_otlr)
    score = mean_squared_error(ytest_otlr, y_pred, squared=False)
    return score
```

In [420…

```
study = optuna.create_study(sampler=TPESampler(), direction="minimize")
study.optimize(objective_with_otlr, n_trials=10000, timeout=3600) # Run for 90 m
print("Number of completed trials: {}".format(len(study.trials)))
print("Best trial:")
trial = study.best_trial

print("\tBest Score: {}".format(trial.value))
print("\tBest Params: ")
for key, value in trial.params.items():
    print("    {}: {}".format(key, value))
```

```
[I 2021-12-09 14:38:42,338] A new study created in memory with name: no-name-b0e
3aeaa-d2b0-4cdd-9661-4bf2cf69d064
[I 2021-12-09 14:38:42,850] Trial 0 finished with value: 0.6100615880736452 and
 parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.20207772928041
78, 'l2_leaf_reg': 0.46462809000585464, 'colsample_bylevel': 0.0868439015511020
5, 'depth': 7, 'boosting_type': 'Ordered', 'bootstrap_type': 'MVS', 'min_data_in
_leaf': 10, 'one_hot_max_size': 5, 'random_state': 9}. Best is trial 0 with valu
e: 0.6100615880736452.
[I 2021-12-09 14:38:46,992] Trial 1 finished with value: 0.6083226916490467 and
 parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.00370101385355
27987, 'l2_leaf_reg': 0.249709761797879, 'colsample_bylevel': 0.0740472340176856
2, 'depth': 9, 'boosting_type': 'Ordered', 'bootstrap_type': 'MVS', 'min_data_in
_leaf': 20, 'one_hot_max_size': 14, 'random_state': 6}. Best is trial 1 with val
ue: 0.6083226916490467.
[I 2021-12-09 14:38:47,192] Trial 2 finished with value: 0.6077419538635421 and
 parameters: {'loss_function': 'Logloss', 'learning_rate': 0.07118128663195478,
 'l2_leaf_reg': 0.051833597827635886, 'colsample_bylevel': 0.0862574153119412,
 'depth': 1, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_lea
f': 10, 'one_hot_max_size': 12, 'random_state': 9}. Best is trial 2 with value:
 0.6077419538635421.
[I 2021-12-09 14:38:47,352] Trial 3 finished with value: 0.6077419538635421 and
 parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.23368430137353
516, 'l2_leaf_reg': 0.04196889427342498, 'colsample_bylevel': 0.0325624732826188
46, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bayesian', 'min_dat
```

```
[I 2021-12-09 15:30:44,872] Trial 1596 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00155388967059984
3, 'l2_leaf_reg': 0.027738652382665138, 'colsample_bylevel': 0.0995344720987569
5, 'depth': 1, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_l
eaf': 15, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with val
ue: 0.6071606606134905.
[I 2021-12-09 15:30:45,770] Trial 1597 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00144582740741151
14, 'l2_leaf_reg': 0.030356942121389265, 'colsample_bylevel': 0.0999437000434624
2, 'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_l
eaf': 15, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with val
ue: 0.6071606606134905.
[I 2021-12-09 15:30:46,639] Trial 1598 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00108294714550937
44, 'l2_leaf_reg': 0.030556619284768296, 'colsample_bylevel': 0.0976188995202078
9, 'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_l
eaf': 16, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with val
ue: 0.6071606606134905.
[I 2021-12-09 15:30:47,745] Trial 1599 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00112162245791860
87, 'l2_leaf_reg': 0.01018008850405752, 'colsample_bylevel': 0.0965519314466918,
'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_lea
f': 16, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with valu
e: 0.6071606606134905.
[I 2021-12-09 15:30:48,657] Trial 1600 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00162916298627649
25, 'l2_leaf_reg': 0.02960517587492722, 'colsample_bylevel': 0.0960765011471263
8, 'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_l
eaf': 15, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with val
ue: 0.6071606606134905.
[I 2021-12-09 15:30:49,544] Trial 1601 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00137659726253690
87, 'l2_leaf_reg': 0.0317971524525236, 'colsample_bylevel': 0.09693399232048679,
'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_lea
f': 19, 'one_hot_max_size': 9, 'random_state': 15}. Best is trial 164 with valu
e: 0.6071606606134905.
[I 2021-12-09 15:30:50,450] Trial 1602 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00154986126191612
53, 'l2_leaf_reg': 0.031657437536160946, 'colsample_bylevel': 0.0959138403580625
7, 'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_l
eaf': 15, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with val
ue: 0.6071606606134905.
[I 2021-12-09 15:30:51,274] Trial 1603 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00097208377283099
77, 'l2_leaf_reg': 0.030558351333186583, 'colsample_bylevel': 0.0306719559278070
13, 'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_
leaf': 19, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with va
lue: 0.6071606606134905.
[I 2021-12-09 15:30:52,173] Trial 1604 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00168426106102314
08, 'l2_leaf_reg': 0.028055257282187065, 'colsample_bylevel': 0.0979655102184894
1, 'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_l
eaf': 15, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with val
ue: 0.6071606606134905.
[I 2021-12-09 15:30:52,962] Trial 1605 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00114629795244992
94, 'l2_leaf_reg': 0.02840530631965779, 'colsample_bylevel': 0.02951311296039015
7, 'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'MVS', 'min_data_in_l
eaf': 15, 'one_hot_max_size': 8, 'random_state': 15}. Best is trial 164 with val
ue: 0.6071606606134905.
[I 2021-12-09 15:30:53,707] Trial 1606 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.00170310866199222
74, 'l2_leaf_reg': 0.02850728337864507, 'colsample_bylevel': 0.03003361415801377
6, 'depth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_dat
a_in_leaf': 15, 'one_hot_max_size': 8, 'random_state': 15, 'subsample': 0.806031
```

```
400659825}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:12,148] Trial 1672 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.386679776979124,
 'l2_leaf_reg': 0.04292402300858492, 'colsample_bylevel': 0.03844870442097602,
 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_
in_leaf': 20, 'one_hot_max_size': 15, 'random_state': 12, 'subsample': 0.3118510
5612827124}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:12,373] Trial 1673 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.5084730730653557,
'l2_leaf_reg': 0.04821865337231929, 'colsample_bylevel': 0.0383709859494296, 'de
pth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_in_l
eaf': 20, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.25776271012
41319}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:12,602] Trial 1674 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.7868959965007768,
'l2_leaf_reg': 0.044660773686882024, 'colsample_bylevel': 0.0387643627422315, 'd
epth': 2, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_in_
leaf': 20, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.2770207378
8875403}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:13,085] Trial 1675 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.008443194766341996
, 'l2_leaf_reg': 0.04909940952373944, 'colsample_bylevel': 0.03895059722780274,
'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_i
n_leaf': 20, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.27167631
684191973}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:13,344] Trial 1676 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.6416208730122228,
'l2_leaf_reg': 0.0439852826622812, 'colsample_bylevel': 0.03896155833427989, 'de
pth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_in_l
eaf': 20, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.32622140130
2866}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:13,813] Trial 1677 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.008166072781078329
, 'l2_leaf_reg': 0.049409135812826806, 'colsample_bylevel': 0.03950977444217853
4, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_dat
a_in_leaf': 20, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.28846
675316154125}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:14,250] Trial 1678 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.009571769432736786
, 'l2_leaf_reg': 0.04722699689319947, 'colsample_bylevel': 0.03880752232987919,
'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_i
n_leaf': 20, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.29876902
34183822}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:14,785] Trial 1679 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.007921259651618502
, 'l2_leaf_reg': 0.045711167822283315, 'colsample_bylevel': 0.039713703095407512
, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_dat
a_in_leaf': 20, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.98900
22063030044}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:15,334] Trial 1680 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.008335455296794959
, 'l2_leaf_reg': 0.05082743671691996, 'colsample_bylevel': 0.03982239723919643,
'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_i
n_leaf': 20, 'one_hot_max_size': 17, 'random_state': 12, 'subsample': 0.92901868
3386356}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:15,865] Trial 1681 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.007433264665873432
, 'l2_leaf_reg': 0.04112170235942176, 'colsample_bylevel': 0.0397071423330838,
 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_data_
in_leaf': 20, 'one_hot_max_size': 16, 'random_state': 12, 'subsample': 0.9259428
586359411}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:31:16,495] Trial 1682 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'Logloss', 'learning_rate': 0.005840309081734121
, 'l2_leaf_reg': 0.04057741629375485, 'colsample_bylevel': 0.039435186331020486
, 'depth': 8, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_dat
```

```
data_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.68
22800617788096}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:12,616] Trial 1748 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.003348185125
1532645, 'l2_leaf_reg': 0.3422625485736026, 'colsample_bylevel': 0.0442682132679
9228, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_
data_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.72
1367436614216}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:13,692] Trial 1749 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.002671926372
0738603, 'l2_leaf_reg': 0.3424380303095717, 'colsample_bylevel': 0.0461070893045
63975, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min
_data_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.7
135421548362513}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:14,739] Trial 1750 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.001890292161
659896, 'l2_leaf_reg': 0.3392936097570707, 'colsample_bylevel': 0.04510277386416
4766, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_
data_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.66
34122488676153}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:15,771] Trial 1751 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.002272641880
985781, 'l2_leaf_reg': 0.31454347908028935, 'colsample_bylevel': 0.0479312706610
75574, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min
_data_in_leaf': 12, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.6
837725939342946}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:16,623] Trial 1752 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.003607790892
279266, 'l2_leaf_reg': 0.306643676059009, 'colsample_bylevel': 0.046749423947398
214, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.627
1916833605107}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:17,464] Trial 1753 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.003658618512
5906984, 'l2_leaf_reg': 0.32571881297701133, 'colsample_bylevel': 0.047197732735
47029, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min
_data_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.6
421504746513212}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:18,540] Trial 1754 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.001861308855
0492882, 'l2_leaf_reg': 0.3330635194937729, 'colsample_bylevel': 0.0475737953257
5428, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_
data_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.66
15991055707149}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:19,373] Trial 1755 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.004054503557
871216, 'l2_leaf_reg': 0.3259011490123874, 'colsample_bylevel': 0.04795259076288
3496, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_
data_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.60
85739299828318}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:20,314] Trial 1756 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.003893459045
5661825, 'l2_leaf_reg': 0.33192366871660023, 'colsample_bylevel': 0.046885663155
35047, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min
_data_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.6
625561437363789}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:21,516] Trial 1757 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.001831963731
828587, 'l2_leaf_reg': 0.33198742646026175, 'colsample_bylevel': 0.0468568985678
329, 'depth': 6, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'min_d
ata_in_leaf': 13, 'one_hot_max_size': 18, 'random_state': 12, 'subsample': 0.715
956020382841}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:32:22,536] Trial 1758 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 0.003047871440
2390004, 'l2_leaf_reg': 0.30960354502540594, 'colsample_bylevel': 0.046726961963
```

```
74, 'depth': 6, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'min_d
ata_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_temperatur
e': 0.018579607935062692}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:38,319] Trial 1824 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 8.592119921133
254e-05, 'l2_leaf_reg': 0.22915467622222552, 'colsample_bylevel': 0.049462804704
48179, 'depth': 6, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'mi
n_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_tempera
ture': 0.7362537902114852}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:40,508] Trial 1825 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 6.691919673832
302e-05, 'l2_leaf_reg': 0.23734697703076735, 'colsample_bylevel': 0.056005653721
263024, 'depth': 6, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'm
in_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_temper
ature': 0.29394739426888955}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:42,700] Trial 1826 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 6.058220142570
027e-05, 'l2_leaf_reg': 0.23522179367571547, 'colsample_bylevel': 0.055555678252
18395, 'depth': 6, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'mi
n_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_tempera
ture': 0.4912557312740491}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:44,871] Trial 1827 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 4.761004063793
867e-05, 'l2_leaf_reg': 0.22158365611063233, 'colsample_bylevel': 0.055414726786
48644, 'depth': 6, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'mi
n_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_tempera
ture': 0.9035129705560778}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:47,217] Trial 1828 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 7.345687945348
251e-05, 'l2_leaf_reg': 0.22334712641405172, 'colsample_bylevel': 0.057173170683
1534, 'depth': 6, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'min
_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_temperat
ure': 0.6439879830576938}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:49,446] Trial 1829 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 9.254210758893
793e-05, 'l2_leaf_reg': 0.2277467237824431, 'colsample_bylevel': 0.0557436923163
3681, 'depth': 6, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'min
_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_temperat
ure': 0.3707144810057763}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:51,708] Trial 1830 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 6.162147632485
27e-05, 'l2_leaf_reg': 0.22282499418865442, 'colsample_bylevel': 0.0553402153774
5744, 'depth': 6, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'min
_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_temperat
ure': 0.47544497687755544}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:55,377] Trial 1831 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 5.441870293590
9865e-05, 'l2_leaf_reg': 0.2160257754094967, 'colsample_bylevel': 0.055494003467
28165, 'depth': 9, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'mi
n_data_in_leaf': 11, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_tempera
ture': 0.8726486275722242}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:33:58,923] Trial 1832 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 4.663702753016
282e-05, 'l2_leaf_reg': 0.22774264232013514, 'colsample_bylevel': 0.055274971311
880625, 'depth': 9, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'm
in_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_temper
ature': 0.4194191981771238}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:34:02,633] Trial 1833 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 2.942826922523
0383e-05, 'l2_leaf_reg': 0.20920916736457715, 'colsample_bylevel': 0.05600651103
8836276, 'depth': 9, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian',
 'min_data_in_leaf': 10, 'one_hot_max_size': 10, 'random_state': 6, 'bagging_tem
perature': 1.1873285854446372}. Best is trial 164 with value: 0.607160660613490
5.
[I 2021-12-09 15:34:06,305] Trial 1834 finished with value: 0.6077419538635421 a
```

```
37826, 'depth': 9, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'mi
n_data_in_leaf': 3, 'one_hot_max_size': 20, 'random_state': 6, 'bagging_temperat
ure': 4.496237676701765}. Best is trial 164 with value: 0.6071606606134905.
[I 2021-12-09 15:38:46,901] Trial 1899 finished with value: 0.6077419538635421 a
nd parameters: {'loss_function': 'CrossEntropy', 'learning_rate': 1.369493304179
38e-05, 'l2_leaf_reg': 0.16564503404537487, 'colsample_bylevel': 0.0628742976983
6873, 'depth': 9, 'boosting_type': 'Ordered', 'bootstrap_type': 'Bayesian', 'min
_data_in_leaf': 3, 'one_hot_max_size': 19, 'random_state': 6, 'bagging_temperatu
re': 4.144582599108663}. Best is trial 164 with value: 0.6071606606134905.
Number of completed trials: 1900
Best trial:
        Best Score: 0.6071606606134905
        Best Params:
    loss_function: CrossEntropy
    learning_rate: 0.005496649026380579
    l2_leaf_reg: 0.038845321749756294
    colsample_bylevel: 0.08791720758027873
    depth: 6
    boosting_type: Ordered
    bootstrap_type: Bayesian
    min_data_in_leaf: 2
    one_hot_max_size: 15
    random_state: 12
    bagging_temperature: 1.319388900442126
```

In [438…

```
catbst_for_pred = CatBoostClassifier(iterations = 50000,
                                     depth = 6,
                                     loss_function='CrossEntropy',
                                     learning_rate= 0.005496649026380579,
                                     l2_leaf_reg= 0.038845321749756294,
                                     colsample_bylevel= 0.08791720758027873,
                                     min_data_in_leaf= 2,
                                     boosting_type= 'Ordered',
                                     bootstrap_type= 'Bayesian',
                                     one_hot_max_size= 15,
                                     leaf_estimation_method = 'Gradient',
                                     eval_metric = 'Accuracy',
                                     custom_metric = 'Accuracy',
                                     random_seed = 12,
                                     bagging_temperature= 1.319388900442126,
                                     verbose = False)
```

In [439…

```
catbst_for_pred.fit(X_with_otlr, y_with_otlr, plot=True)
y_predict = catbst_for_pred.predict(Z)
```

```
Custom logger is already specified. Specify more than one logger at same time is
not thread safe.
```

In [440…

```
catpred = pd.DataFrame({'id':idarray, 'Decision':np.squeeze(y_predict)})
catpred.to_csv('cat_pred_optuna_tuned_V5',index=False)
```

In [ ]:

Neural Networks

In [802…

```
!pip install tensorflow
!pip install scikeras
```

```
Requirement already satisfied: tensorflow in /Users/xuzikai/Downloads/anaconda3/
lib/python3.8/site-packages (2.7.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /Users/xuzikai/Downloads/a
naconda3/lib/python3.8/site-packages (from tensorflow) (1.42.0)
Requirement already satisfied: protobuf>=3.9.2 in /Users/xuzikai/Downloads/anaco
nda3/lib/python3.8/site-packages (from tensorflow) (3.19.1)
Requirement already satisfied: wheel<1.0,>=0.32.0 in /Users/xuzikai/Downloads/an
aconda3/lib/python3.8/site-packages (from tensorflow) (0.36.2)
Requirement already satisfied: opt-einsum>=2.3.2 in /Users/xuzikai/Downloads/ana
conda3/lib/python3.8/site-packages (from tensorflow) (3.3.0)
Requirement already satisfied: absl-py>=0.4.0 in /Users/xuzikai/Downloads/anacon
da3/lib/python3.8/site-packages (from tensorflow) (1.0.0)
Requirement already satisfied: termcolor>=1.1.0 in /Users/xuzikai/Downloads/anac
onda3/lib/python3.8/site-packages (from tensorflow) (1.1.0)
Requirement already satisfied: keras-preprocessing>=1.1.1 in /Users/xuzikai/Down
loads/anaconda3/lib/python3.8/site-packages (from tensorflow) (1.1.2)
Requirement already satisfied: numpy>=1.14.5 in /Users/xuzikai/Downloads/anacond
a3/lib/python3.8/site-packages (from tensorflow) (1.20.1)
Requirement already satisfied: tensorboard~=2.6 in /Users/xuzikai/Downloads/anac
onda3/lib/python3.8/site-packages (from tensorflow) (2.7.0)
Requirement already satisfied: astunparse>=1.6.0 in /Users/xuzikai/Downloads/ana
conda3/lib/python3.8/site-packages (from tensorflow) (1.6.3)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.21.0 in /Users/xu
zikai/Downloads/anaconda3/lib/python3.8/site-packages (from tensorflow) (0.22.0)
Requirement already satisfied: keras<2.8,>=2.7.0rc0 in /Users/xuzikai/Downloads/
anaconda3/lib/python3.8/site-packages (from tensorflow) (2.7.0)
Requirement already satisfied: flatbuffers<3.0,>=1.12 in /Users/xuzikai/Download
s/anaconda3/lib/python3.8/site-packages (from tensorflow) (2.0)
Requirement already satisfied: h5py>=2.9.0 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from tensorflow) (2.10.0)
Requirement already satisfied: tensorflow-estimator<2.8,~=2.7.0rc0 in /Users/xuz
ikai/Downloads/anaconda3/lib/python3.8/site-packages (from tensorflow) (2.7.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /Users/xuzikai/Downlo
ads/anaconda3/lib/python3.8/site-packages (from tensorflow) (3.7.4.3)
Requirement already satisfied: gast<0.5.0,>=0.2.1 in /Users/xuzikai/Downloads/an
aconda3/lib/python3.8/site-packages (from tensorflow) (0.4.0)
Requirement already satisfied: six>=1.12.0 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from tensorflow) (1.15.0)
Requirement already satisfied: google-pasta>=0.1.1 in /Users/xuzikai/Downloads/a
naconda3/lib/python3.8/site-packages (from tensorflow) (0.2.0)
Requirement already satisfied: libclang>=9.0.1 in /Users/xuzikai/Downloads/anaco
nda3/lib/python3.8/site-packages (from tensorflow) (12.0.0)
Requirement already satisfied: wrapt>=1.11.0 in /Users/xuzikai/Downloads/anacond
a3/lib/python3.8/site-packages (from tensorflow) (1.12.1)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /Users/x
uzikai/Downloads/anaconda3/lib/python3.8/site-packages (from tensorboard~=2.6->t
ensorflow) (0.6.1)
Requirement already satisfied: google-auth<3,>=1.6.3 in /Users/xuzikai/Download
s/anaconda3/lib/python3.8/site-packages (from tensorboard~=2.6->tensorflow) (2.
3.3)
Requirement already satisfied: requests<3,>=2.21.0 in /Users/xuzikai/Downloads/a
naconda3/lib/python3.8/site-packages (from tensorboard~=2.6->tensorflow) (2.25.
1)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /Users/xuzika
i/Downloads/anaconda3/lib/python3.8/site-packages (from tensorboard~=2.6->tensor
flow) (0.4.6)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /Users/xuzikai/D
ownloads/anaconda3/lib/python3.8/site-packages (from tensorboard~=2.6->tensorflo
w) (1.8.0)
Requirement already satisfied: setuptools>=41.0.0 in /Users/xuzikai/Downloads/an
aconda3/lib/python3.8/site-packages (from tensorboard~=2.6->tensorflow) (52.0.0.
```

```
post20210125)
Requirement already satisfied: werkzeug>=0.11.15 in /Users/xuzikai/Downloads/ana
conda3/lib/python3.8/site-packages (from tensorboard~=2.6->tensorflow) (1.0.1)
Requirement already satisfied: markdown>=2.6.8 in /Users/xuzikai/Downloads/anaco
nda3/lib/python3.8/site-packages (from tensorboard~=2.6->tensorflow) (3.3.6)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /Users/xuzikai/Download
s/anaconda3/lib/python3.8/site-packages (from google-auth<3,>=1.6.3->tensorboard
~=2.6->tensorflow) (4.2.4)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /Users/xuzikai/Download
s/anaconda3/lib/python3.8/site-packages (from google-auth<3,>=1.6.3->tensorboard
~=2.6->tensorflow) (0.2.8)
Requirement already satisfied: rsa<5,>=3.1.4 in /Users/xuzikai/Downloads/anacond
a3/lib/python3.8/site-packages (from google-auth<3,>=1.6.3->tensorboard~=2.6->te
nsorflow) (4.8)
Requirement already satisfied: requests-oauthlib>=0.7.0 in /Users/xuzikai/Downlo
ads/anaconda3/lib/python3.8/site-packages (from google-auth-oauthlib<0.5,>=0.4.1
->tensorboard~=2.6->tensorflow) (1.3.0)
Requirement already satisfied: importlib-metadata>=4.4 in /Users/xuzikai/Downloa
ds/anaconda3/lib/python3.8/site-packages (from markdown>=2.6.8->tensorboard~=2.6
->tensorflow) (4.8.2)
Requirement already satisfied: zipp>=0.5 in /Users/xuzikai/Downloads/anaconda3/l
ib/python3.8/site-packages (from importlib-metadata>=4.4->markdown>=2.6.8->tenso
rboard~=2.6->tensorflow) (3.4.1)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /Users/xuzikai/Downloads/
anaconda3/lib/python3.8/site-packages (from pyasn1-modules>=0.2.1->google-auth<
3,>=1.6.3->tensorboard~=2.6->tensorflow) (0.4.8)
Requirement already satisfied: certifi>=2017.4.17 in /Users/xuzikai/Downloads/an
aconda3/lib/python3.8/site-packages (from requests<3,>=2.21.0->tensorboard~=2.6-
>tensorflow) (2020.12.5)
Requirement already satisfied: idna<3,>=2.5 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from requests<3,>=2.21.0->tensorboard~=2.6->tenso
rflow) (2.10)
Requirement already satisfied: chardet<5,>=3.0.2 in /Users/xuzikai/Downloads/ana
conda3/lib/python3.8/site-packages (from requests<3,>=2.21.0->tensorboard~=2.6->
tensorflow) (4.0.0)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /Users/xuzikai/Download
s/anaconda3/lib/python3.8/site-packages (from requests<3,>=2.21.0->tensorboard~=
2.6->tensorflow) (1.26.4)
Requirement already satisfied: oauthlib>=3.0.0 in /Users/xuzikai/Downloads/anaco
nda3/lib/python3.8/site-packages (from requests-oauthlib>=0.7.0->google-auth-oau
thlib<0.5,>=0.4.1->tensorboard~=2.6->tensorflow) (3.1.1)
Requirement already satisfied: keras in /Users/xuzikai/Downloads/anaconda3/lib/p
ython3.8/site-packages (2.7.0)
Requirement already satisfied: scikeras in /Users/xuzikai/Downloads/anaconda3/li
b/python3.8/site-packages (0.6.0)
Requirement already satisfied: scikit-learn>=1.0.0 in /Users/xuzikai/Downloads/a
naconda3/lib/python3.8/site-packages (from scikeras) (1.0.1)
Requirement already satisfied: packaging<22.0,>=0.21 in /Users/xuzikai/Download
s/anaconda3/lib/python3.8/site-packages (from scikeras) (20.9)
Requirement already satisfied: pyparsing>=2.0.2 in /Users/xuzikai/Downloads/anac
onda3/lib/python3.8/site-packages (from packaging<22.0,>=0.21->scikeras) (2.4.7)
Requirement already satisfied: joblib>=0.11 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from scikit-learn>=1.0.0->scikeras) (1.0.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /Users/xuzikai/Downloads/
anaconda3/lib/python3.8/site-packages (from scikit-learn>=1.0.0->scikeras) (2.1.
0)
Requirement already satisfied: numpy>=1.14.6 in /Users/xuzikai/Downloads/anacond
a3/lib/python3.8/site-packages (from scikit-learn>=1.0.0->scikeras) (1.20.1)
Requirement already satisfied: scipy>=1.1.0 in /Users/xuzikai/Downloads/anaconda
3/lib/python3.8/site-packages (from scikit-learn>=1.0.0->scikeras) (1.6.2)
```

In [724…

```
from keras.models import Sequential
from keras.layers import Dense
```

```
from sklearn.model_selection import KFold
from keras.wrappers.scikit_learn import KerasClassifier
```

In fact, there is a theoretical finding by Lippmann in the 1987 paper “An introduction to computing with neural nets” that shows that an MLP with two hidden layers is sufficient for creating classification regions of any desired shape. This is instructive, although it should be noted that no indication of how many nodes to use in each layer or how to learn the weights is given.

Since Neural Networks is not likely to be affected by outliers when the propotion of outliers is small, we will use the data with outliers to train it.

In [851…

```
# define the keras model
model = Sequential()
model.add(Dense(120, input_dim=X_with_otlr.shape[1], activation='relu'))
model.add(Dense(80, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']

# fit the keras model on the dataset
model.fit(X_with_otlr, y_raw, epochs=800, batch_size=10)

# evaluate the keras model
_, accuracy = model.evaluate(X_with_otlr, y_raw)
print('Accuracy: %.2f' % (accuracy*100))

# make class predictions with the model
NN_predict = (model.predict(Z) > 0.5).astype(int)

# export
NN_pred = pd.DataFrame({'id':idarray, 'Decision':np.squeeze(NN_predict)})
NN_pred.to_csv('NN_pred',index=False)
```

```
Epoch 1/800
748/748 [==============================] - 1s 939us/step - loss: 0.7355 - accura
cy: 0.6225
Epoch 2/800
748/748 [==============================] - 1s 942us/step - loss: 0.6447 - accura
cy: 0.6583
Epoch 3/800
748/748 [==============================] - 1s 940us/step - loss: 0.6216 - accura
cy: 0.6751
Epoch 4/800
748/748 [==============================] - 1s 937us/step - loss: 0.6272 - accura
cy: 0.6713
Epoch 5/800
748/748 [==============================] - 1s 940us/step - loss: 0.6112 - accura
cy: 0.6786
Epoch 6/800
748/748 [==============================] - 1s 936us/step - loss: 0.6061 - accura
cy: 0.6836
Epoch 7/800
748/748 [==============================] - 1s 934us/step - loss: 0.5983 - accura
cy: 0.6849
Epoch 8/800
```

```
Epoch 550/800
748/748 [==============================] - 1s 915us/step - loss: 0.1616 - accura
cy: 0.9277
Epoch 551/800
748/748 [==============================] - 1s 930us/step - loss: 0.1534 - accura
cy: 0.9315
Epoch 552/800
748/748 [==============================] - 1s 917us/step - loss: 0.1642 - accura
cy: 0.9258
Epoch 553/800
748/748 [==============================] - 1s 915us/step - loss: 0.1842 - accura
cy: 0.9213
Epoch 554/800
748/748 [==============================] - 1s 918us/step - loss: 0.1586 - accura
cy: 0.9304
Epoch 555/800
748/748 [==============================] - 1s 919us/step - loss: 0.1721 - accura
cy: 0.9254
Epoch 556/800
748/748 [==============================] - 1s 914us/step - loss: 0.1743 - accura
cy: 0.9225
Epoch 557/800
748/748 [==============================] - 1s 917us/step - loss: 0.1536 - accura
cy: 0.9299
Epoch 558/800
748/748 [==============================] - 1s 916us/step - loss: 0.1661 - accura
cy: 0.9264
Epoch 559/800
748/748 [==============================] - 1s 916us/step - loss: 0.1548 - accura
cy: 0.9301
Epoch 560/800
748/748 [==============================] - 1s 919us/step - loss: 0.1657 - accura
cy: 0.9300
Epoch 561/800
748/748 [==============================] - 1s 975us/step - loss: 0.1554 - accura
cy: 0.9320
Epoch 562/800
748/748 [==============================] - 1s 914us/step - loss: 0.1680 - accura
cy: 0.9289
Epoch 563/800
748/748 [==============================] - 1s 918us/step - loss: 0.1575 - accura
cy: 0.9291
Epoch 564/800
748/748 [==============================] - 1s 916us/step - loss: 0.1597 - accura
cy: 0.9297
Epoch 565/800
748/748 [==============================] - 1s 919us/step - loss: 0.1659 - accura
cy: 0.9289
Epoch 566/800
748/748 [==============================] - 1s 917us/step - loss: 0.1530 - accura
cy: 0.9305
Epoch 567/800
748/748 [==============================] - 1s 920us/step - loss: 0.1616 - accura
cy: 0.9297
Epoch 568/800
748/748 [==============================] - 1s 926us/step - loss: 0.1948 - accura
cy: 0.9174
Epoch 569/800
748/748 [==============================] - 1s 924us/step - loss: 0.1596 - accura
cy: 0.9323
Epoch 570/800
748/748 [==============================] - 1s 921us/step - loss: 0.1700 - accura
cy: 0.9277
Epoch 571/800
748/748 [==============================] - 1s 922us/step - loss: 0.1539 - accura
```

```
cy: 0.9316
Epoch 572/800
748/748 [==============================] - 1s 930us/step - loss: 0.1670 - accura
cy: 0.9276
Epoch 573/800
748/748 [==============================] - 1s 913us/step - loss: 0.1605 - accura
cy: 0.9308
Epoch 574/800
748/748 [==============================] - 1s 930us/step - loss: 0.1447 - accura
cy: 0.9354
Epoch 575/800
748/748 [==============================] - 1s 916us/step - loss: 0.1699 - accura
cy: 0.9295
Epoch 576/800
748/748 [==============================] - 1s 916us/step - loss: 0.1759 - accura
cy: 0.9234
Epoch 577/800
748/748 [==============================] - 1s 918us/step - loss: 0.1595 - accura
cy: 0.9311
Epoch 578/800
748/748 [==============================] - 1s 917us/step - loss: 0.1583 - accura
cy: 0.9300
Epoch 579/800
748/748 [==============================] - 1s 918us/step - loss: 0.1579 - accura
cy: 0.9300
Epoch 580/800
748/748 [==============================] - 1s 921us/step - loss: 0.1599 - accura
cy: 0.9293
Epoch 581/800
748/748 [==============================] - 1s 912us/step - loss: 0.1999 - accura
cy: 0.9141
Epoch 582/800
748/748 [==============================] - 1s 914us/step - loss: 0.1731 - accura
cy: 0.9277
Epoch 583/800
748/748 [==============================] - 1s 913us/step - loss: 0.1491 - accura
cy: 0.9324
Epoch 584/800
748/748 [==============================] - 1s 923us/step - loss: 0.1679 - accura
cy: 0.9287
Epoch 585/800
748/748 [==============================] - 1s 948us/step - loss: 0.1429 - accura
cy: 0.9349
Epoch 586/800
748/748 [==============================] - 1s 968us/step - loss: 0.1809 - accura
cy: 0.9220
Epoch 587/800
748/748 [==============================] - 1s 917us/step - loss: 0.1528 - accura
cy: 0.9341
Epoch 588/800
748/748 [==============================] - 1s 922us/step - loss: 0.1503 - accura
cy: 0.9356
Epoch 589/800
748/748 [==============================] - 1s 920us/step - loss: 0.1574 - accura
cy: 0.9316
Epoch 590/800
748/748 [==============================] - 1s 914us/step - loss: 0.1544 - accura
cy: 0.9317
Epoch 591/800
748/748 [==============================] - 1s 918us/step - loss: 0.1529 - accura
cy: 0.9327
Epoch 592/800
748/748 [==============================] - 1s 916us/step - loss: 0.1423 - accura
cy: 0.9358
Epoch 593/800
```

```
748/748 [==============================] - 1s 918us/step - loss: 0.1664 - accura
cy: 0.9288
Epoch 594/800
748/748 [==============================] - 1s 914us/step - loss: 0.1671 - accura
cy: 0.9275
Epoch 595/800
748/748 [==============================] - 1s 919us/step - loss: 0.1476 - accura
cy: 0.9358
Epoch 596/800
748/748 [==============================] - 1s 914us/step - loss: 0.1818 - accura
cy: 0.9217
Epoch 597/800
748/748 [==============================] - 1s 917us/step - loss: 0.1416 - accura
cy: 0.9382
Epoch 598/800
748/748 [==============================] - 1s 916us/step - loss: 0.1592 - accura
cy: 0.9315
Epoch 599/800
748/748 [==============================] - 1s 931us/step - loss: 0.1495 - accura
cy: 0.9336
Epoch 600/800
748/748 [==============================] - 1s 914us/step - loss: 0.1714 - accura
cy: 0.9242
Epoch 601/800
748/748 [==============================] - 1s 915us/step - loss: 0.1504 - accura
cy: 0.9341
Epoch 602/800
748/748 [==============================] - 1s 914us/step - loss: 0.1503 - accura
cy: 0.9348
Epoch 603/800
748/748 [==============================] - 1s 913us/step - loss: 0.1457 - accura
cy: 0.9387
Epoch 604/800
748/748 [==============================] - 1s 915us/step - loss: 0.1525 - accura
cy: 0.9317
Epoch 605/800
748/748 [==============================] - 1s 917us/step - loss: 0.1717 - accura
cy: 0.9262
Epoch 606/800
748/748 [==============================] - 1s 919us/step - loss: 0.1734 - accura
cy: 0.9246
Epoch 607/800
748/748 [==============================] - 1s 924us/step - loss: 0.1550 - accura
cy: 0.9320
Epoch 608/800
748/748 [==============================] - 1s 917us/step - loss: 0.1507 - accura
cy: 0.9364
Epoch 609/800
748/748 [==============================] - 1s 913us/step - loss: 0.1464 - accura
cy: 0.9325
Epoch 610/800
748/748 [==============================] - 1s 1ms/step - loss: 0.1539 - accurac
y: 0.9328
Epoch 611/800
748/748 [==============================] - 1s 931us/step - loss: 0.1722 - accura
cy: 0.9254
Epoch 612/800
748/748 [==============================] - 1s 924us/step - loss: 0.1512 - accura
cy: 0.9340
Epoch 613/800
748/748 [==============================] - 1s 924us/step - loss: 0.1563 - accura
cy: 0.9291
Epoch 614/800
748/748 [==============================] - 1s 914us/step - loss: 0.1364 - accura
cy: 0.9386
```

```
Epoch 615/800
748/748 [==============================] - 1s 917us/step - loss: 0.1695 - accura
cy: 0.9266
Epoch 616/800
748/748 [==============================] - 1s 915us/step - loss: 0.1412 - accura
cy: 0.9379
Epoch 617/800
748/748 [==============================] - 1s 917us/step - loss: 0.1639 - accura
cy: 0.9304
Epoch 618/800
748/748 [==============================] - 1s 916us/step - loss: 0.1437 - accura
cy: 0.9388
Epoch 619/800
748/748 [==============================] - 1s 918us/step - loss: 0.1809 - accura
cy: 0.9264
Epoch 620/800
748/748 [==============================] - 1s 918us/step - loss: 0.1519 - accura
cy: 0.9323
Epoch 621/800
748/748 [==============================] - 1s 917us/step - loss: 0.1481 - accura
cy: 0.9349
Epoch 622/800
748/748 [==============================] - 1s 914us/step - loss: 0.1665 - accura
cy: 0.9266
Epoch 623/800
748/748 [==============================] - 1s 916us/step - loss: 0.1335 - accura
cy: 0.9402
Epoch 624/800
748/748 [==============================] - 1s 914us/step - loss: 0.1610 - accura
cy: 0.9288
Epoch 625/800
748/748 [==============================] - 1s 916us/step - loss: 0.1420 - accura
cy: 0.9355
Epoch 626/800
748/748 [==============================] - 1s 920us/step - loss: 0.1585 - accura
cy: 0.9325
Epoch 627/800
748/748 [==============================] - 1s 919us/step - loss: 0.1425 - accura
cy: 0.9370
Epoch 628/800
748/748 [==============================] - 1s 927us/step - loss: 0.1735 - accura
cy: 0.9249
Epoch 629/800
748/748 [==============================] - 1s 921us/step - loss: 0.1427 - accura
cy: 0.9354
Epoch 630/800
748/748 [==============================] - 1s 918us/step - loss: 0.1763 - accura
cy: 0.9250
Epoch 631/800
748/748 [==============================] - 1s 918us/step - loss: 0.1372 - accura
cy: 0.9370
Epoch 632/800
748/748 [==============================] - 1s 914us/step - loss: 0.1682 - accura
cy: 0.9283
Epoch 633/800
748/748 [==============================] - 1s 918us/step - loss: 0.1532 - accura
cy: 0.9307
Epoch 634/800
748/748 [==============================] - 1s 914us/step - loss: 0.1543 - accura
cy: 0.9300
Epoch 635/800
748/748 [==============================] - 1s 913us/step - loss: 0.1329 - accura
cy: 0.9390
Epoch 636/800
748/748 [==============================] - 1s 911us/step - loss: 0.1705 - accura
```

```
cy: 0.9287
Epoch 637/800
748/748 [==============================] - 1s 917us/step - loss: 0.1414 - accura
cy: 0.9354
Epoch 638/800
748/748 [==============================] - 1s 916us/step - loss: 0.1624 - accura
cy: 0.9323
Epoch 639/800
748/748 [==============================] - 1s 911us/step - loss: 0.1413 - accura
cy: 0.9366
Epoch 640/800
748/748 [==============================] - 1s 913us/step - loss: 0.1482 - accura
cy: 0.9359
Epoch 641/800
748/748 [==============================] - 1s 917us/step - loss: 0.1584 - accura
cy: 0.9321
Epoch 642/800
748/748 [==============================] - 1s 916us/step - loss: 0.1450 - accura
cy: 0.9362
Epoch 643/800
748/748 [==============================] - 1s 923us/step - loss: 0.1539 - accura
cy: 0.9328
Epoch 644/800
748/748 [==============================] - 1s 913us/step - loss: 0.1590 - accura
cy: 0.9333
Epoch 645/800
748/748 [==============================] - 1s 920us/step - loss: 0.1466 - accura
cy: 0.9355
Epoch 646/800
748/748 [==============================] - 1s 916us/step - loss: 0.1474 - accura
cy: 0.9332
Epoch 647/800
748/748 [==============================] - 1s 915us/step - loss: 0.1398 - accura
cy: 0.9358
Epoch 648/800
748/748 [==============================] - 1s 964us/step - loss: 0.1515 - accura
cy: 0.9341
Epoch 649/800
748/748 [==============================] - 1s 917us/step - loss: 0.1651 - accura
cy: 0.9271
Epoch 650/800
748/748 [==============================] - 1s 915us/step - loss: 0.1337 - accura
cy: 0.9392
Epoch 651/800
748/748 [==============================] - 1s 919us/step - loss: 0.1582 - accura
cy: 0.9316
Epoch 652/800
748/748 [==============================] - 1s 917us/step - loss: 0.1653 - accura
cy: 0.9277
Epoch 653/800
748/748 [==============================] - 1s 919us/step - loss: 0.1387 - accura
cy: 0.9396
Epoch 654/800
748/748 [==============================] - 1s 916us/step - loss: 0.1530 - accura
cy: 0.9313
Epoch 655/800
748/748 [==============================] - 1s 923us/step - loss: 0.1402 - accura
cy: 0.9349
Epoch 656/800
748/748 [==============================] - 1s 918us/step - loss: 0.1438 - accura
cy: 0.9363
Epoch 657/800
748/748 [==============================] - 1s 919us/step - loss: 0.1438 - accura
cy: 0.9352
Epoch 658/800
```

```
748/748 [==============================] - 1s 929us/step - loss: 0.1667 - accura
cy: 0.9273
Epoch 659/800
748/748 [==============================] - 1s 916us/step - loss: 0.1347 - accura
cy: 0.9387
Epoch 660/800
748/748 [==============================] - 1s 920us/step - loss: 0.1591 - accura
cy: 0.9312
Epoch 661/800
748/748 [==============================] - 1s 916us/step - loss: 0.1346 - accura
cy: 0.9396
Epoch 662/800
748/748 [==============================] - 1s 913us/step - loss: 0.1564 - accura
cy: 0.9312
Epoch 663/800
748/748 [==============================] - 1s 922us/step - loss: 0.1598 - accura
cy: 0.9295
Epoch 664/800
748/748 [==============================] - 1s 918us/step - loss: 0.1277 - accura
cy: 0.9428
Epoch 665/800
748/748 [==============================] - 1s 916us/step - loss: 0.1467 - accura
cy: 0.9343
Epoch 666/800
748/748 [==============================] - 1s 910us/step - loss: 0.1467 - accura
cy: 0.9376
Epoch 667/800
748/748 [==============================] - 1s 913us/step - loss: 0.1549 - accura
cy: 0.9327
Epoch 668/800
748/748 [==============================] - 1s 912us/step - loss: 0.1482 - accura
cy: 0.9347
Epoch 669/800
748/748 [==============================] - 1s 918us/step - loss: 0.1401 - accura
cy: 0.9380
Epoch 670/800
748/748 [==============================] - 1s 916us/step - loss: 0.1479 - accura
cy: 0.9347
Epoch 671/800
748/748 [==============================] - 1s 918us/step - loss: 0.1321 - accura
cy: 0.9392
Epoch 672/800
748/748 [==============================] - 1s 923us/step - loss: 0.1504 - accura
cy: 0.9367
Epoch 673/800
748/748 [==============================] - 1s 918us/step - loss: 0.1590 - accura
cy: 0.9275
Epoch 674/800
748/748 [==============================] - 1s 918us/step - loss: 0.1671 - accura
cy: 0.9304
Epoch 675/800
748/748 [==============================] - 1s 914us/step - loss: 0.1326 - accura
cy: 0.9400
Epoch 676/800
748/748 [==============================] - 1s 913us/step - loss: 0.1532 - accura
cy: 0.9351
Epoch 677/800
748/748 [==============================] - 1s 911us/step - loss: 0.1482 - accura
cy: 0.9341
Epoch 678/800
748/748 [==============================] - 1s 913us/step - loss: 0.1451 - accura
cy: 0.9362
Epoch 679/800
748/748 [==============================] - 1s 915us/step - loss: 0.1268 - accura
cy: 0.9435
```

```
Epoch 680/800
748/748 [==============================] - 1s 915us/step - loss: 0.1553 - accura
cy: 0.9331
Epoch 681/800
748/748 [==============================] - 1s 915us/step - loss: 0.1653 - accura
cy: 0.9299
Epoch 682/800
748/748 [==============================] - 1s 913us/step - loss: 0.1481 - accura
cy: 0.9347
Epoch 683/800
748/748 [==============================] - 1s 915us/step - loss: 0.1375 - accura
cy: 0.9363
Epoch 684/800
748/748 [==============================] - 1s 923us/step - loss: 0.1602 - accura
cy: 0.9312
Epoch 685/800
748/748 [==============================] - 1s 917us/step - loss: 0.1382 - accura
cy: 0.9402
Epoch 686/800
748/748 [==============================] - 1s 917us/step - loss: 0.1519 - accura
cy: 0.9359
Epoch 687/800
748/748 [==============================] - 1s 920us/step - loss: 0.1397 - accura
cy: 0.9356
Epoch 688/800
748/748 [==============================] - 1s 918us/step - loss: 0.1325 - accura
cy: 0.9412
Epoch 689/800
748/748 [==============================] - 1s 920us/step - loss: 0.1483 - accura
cy: 0.9366
Epoch 690/800
748/748 [==============================] - 1s 914us/step - loss: 0.1709 - accura
cy: 0.9327
Epoch 691/800
748/748 [==============================] - 1s 917us/step - loss: 0.1285 - accura
cy: 0.9424
Epoch 692/800
748/748 [==============================] - 1s 919us/step - loss: 0.1507 - accura
cy: 0.9354
Epoch 693/800
748/748 [==============================] - 1s 921us/step - loss: 0.1341 - accura
cy: 0.9390
Epoch 694/800
748/748 [==============================] - 1s 918us/step - loss: 0.1561 - accura
cy: 0.9327
Epoch 695/800
748/748 [==============================] - 1s 913us/step - loss: 0.1419 - accura
cy: 0.9367
Epoch 696/800
748/748 [==============================] - 1s 915us/step - loss: 0.1399 - accura
cy: 0.9402
Epoch 697/800
748/748 [==============================] - 1s 912us/step - loss: 0.1353 - accura
cy: 0.9407
Epoch 698/800
748/748 [==============================] - 1s 918us/step - loss: 0.1462 - accura
cy: 0.9351
Epoch 699/800
748/748 [==============================] - 1s 912us/step - loss: 0.1305 - accura
cy: 0.9400
Epoch 700/800
748/748 [==============================] - 1s 914us/step - loss: 0.1558 - accura
cy: 0.9341
Epoch 701/800
748/748 [==============================] - 1s 914us/step - loss: 0.1451 - accura
```

```
cy: 0.9378
Epoch 702/800
748/748 [==============================] - 1s 926us/step - loss: 0.1770 - accura
cy: 0.9257
Epoch 703/800
748/748 [==============================] - 1s 914us/step - loss: 0.1294 - accura
cy: 0.9400
Epoch 704/800
748/748 [==============================] - 1s 912us/step - loss: 0.1217 - accura
cy: 0.9467
Epoch 705/800
748/748 [==============================] - 1s 917us/step - loss: 0.1386 - accura
cy: 0.9396
Epoch 706/800
748/748 [==============================] - 1s 916us/step - loss: 0.1378 - accura
cy: 0.9382
Epoch 707/800
748/748 [==============================] - 1s 915us/step - loss: 0.1605 - accura
cy: 0.9295
Epoch 708/800
748/748 [==============================] - 1s 914us/step - loss: 0.1508 - accura
cy: 0.9356
Epoch 709/800
748/748 [==============================] - 1s 916us/step - loss: 0.1544 - accura
cy: 0.9337
Epoch 710/800
748/748 [==============================] - 1s 916us/step - loss: 0.1367 - accura
cy: 0.9404
Epoch 711/800
748/748 [==============================] - 1s 916us/step - loss: 0.1447 - accura
cy: 0.9372
Epoch 712/800
748/748 [==============================] - 1s 920us/step - loss: 0.1480 - accura
cy: 0.9336
Epoch 713/800
748/748 [==============================] - 1s 914us/step - loss: 0.1375 - accura
cy: 0.9400
Epoch 714/800
748/748 [==============================] - 1s 917us/step - loss: 0.1333 - accura
cy: 0.9404
Epoch 715/800
748/748 [==============================] - 1s 916us/step - loss: 0.1559 - accura
cy: 0.9363
Epoch 716/800
748/748 [==============================] - 1s 921us/step - loss: 0.1373 - accura
cy: 0.9376
Epoch 717/800
748/748 [==============================] - 1s 915us/step - loss: 0.1369 - accura
cy: 0.9380
Epoch 718/800
748/748 [==============================] - 1s 914us/step - loss: 0.1335 - accura
cy: 0.9387
Epoch 719/800
748/748 [==============================] - 1s 916us/step - loss: 0.1519 - accura
cy: 0.9348
Epoch 720/800
748/748 [==============================] - 1s 916us/step - loss: 0.1197 - accura
cy: 0.9449
Epoch 721/800
748/748 [==============================] - 1s 914us/step - loss: 0.1803 - accura
cy: 0.9264
Epoch 722/800
748/748 [==============================] - 1s 914us/step - loss: 0.1260 - accura
cy: 0.9412
Epoch 723/800
```

```
748/748 [==============================] - 1s 921us/step - loss: 0.1271 - accura
cy: 0.9408
Epoch 724/800
748/748 [==============================] - 1s 916us/step - loss: 0.1566 - accura
cy: 0.9309
Epoch 725/800
748/748 [==============================] - 1s 913us/step - loss: 0.1427 - accura
cy: 0.9386
Epoch 726/800
748/748 [==============================] - 1s 916us/step - loss: 0.1246 - accura
cy: 0.9436
Epoch 727/800
748/748 [==============================] - 1s 913us/step - loss: 0.1629 - accura
cy: 0.9305
Epoch 728/800
748/748 [==============================] - 1s 918us/step - loss: 0.1422 - accura
cy: 0.9372
Epoch 729/800
748/748 [==============================] - 1s 923us/step - loss: 0.1446 - accura
cy: 0.9371
Epoch 730/800
748/748 [==============================] - 1s 918us/step - loss: 0.1402 - accura
cy: 0.9375
Epoch 731/800
748/748 [==============================] - 1s 912us/step - loss: 0.1425 - accura
cy: 0.9384
Epoch 732/800
748/748 [==============================] - 1s 923us/step - loss: 0.1488 - accura
cy: 0.9349
Epoch 733/800
748/748 [==============================] - 1s 915us/step - loss: 0.1370 - accura
cy: 0.9383
Epoch 734/800
748/748 [==============================] - 1s 915us/step - loss: 0.1488 - accura
cy: 0.9308
Epoch 735/800
748/748 [==============================] - 1s 965us/step - loss: 0.1293 - accura
cy: 0.9395
Epoch 736/800
748/748 [==============================] - 1s 912us/step - loss: 0.1472 - accura
cy: 0.9366
Epoch 737/800
748/748 [==============================] - 1s 916us/step - loss: 0.1591 - accura
cy: 0.9339
Epoch 738/800
748/748 [==============================] - 1s 1ms/step - loss: 0.1231 - accurac
y: 0.9446
Epoch 739/800
748/748 [==============================] - 1s 916us/step - loss: 0.1490 - accura
cy: 0.9368
Epoch 740/800
748/748 [==============================] - 1s 916us/step - loss: 0.1330 - accura
cy: 0.9388
Epoch 741/800
748/748 [==============================] - 1s 915us/step - loss: 0.1417 - accura
cy: 0.9367
Epoch 742/800
748/748 [==============================] - 1s 921us/step - loss: 0.1435 - accura
cy: 0.9375
Epoch 743/800
748/748 [==============================] - 1s 913us/step - loss: 0.1507 - accura
cy: 0.9333
Epoch 744/800
748/748 [==============================] - 1s 910us/step - loss: 0.1328 - accura
cy: 0.9388
```

```
Epoch 745/800
748/748 [==============================] - 1s 916us/step - loss: 0.1289 - accura
cy: 0.9431
Epoch 746/800
748/748 [==============================] - 1s 920us/step - loss: 0.1347 - accura
cy: 0.9386
Epoch 747/800
748/748 [==============================] - 1s 918us/step - loss: 0.1380 - accura
cy: 0.9388
Epoch 748/800
748/748 [==============================] - 1s 920us/step - loss: 0.1441 - accura
cy: 0.9371
Epoch 749/800
748/748 [==============================] - 1s 925us/step - loss: 0.1259 - accura
cy: 0.9427
Epoch 750/800
748/748 [==============================] - 1s 919us/step - loss: 0.1531 - accura
cy: 0.9356
Epoch 751/800
748/748 [==============================] - 1s 916us/step - loss: 0.1245 - accura
cy: 0.9426
Epoch 752/800
748/748 [==============================] - 1s 919us/step - loss: 0.1439 - accura
cy: 0.9391
Epoch 753/800
748/748 [==============================] - 1s 917us/step - loss: 0.1371 - accura
cy: 0.9396
Epoch 754/800
748/748 [==============================] - 1s 916us/step - loss: 0.1495 - accura
cy: 0.9366
Epoch 755/800
748/748 [==============================] - 1s 911us/step - loss: 0.1213 - accura
cy: 0.9445
Epoch 756/800
748/748 [==============================] - 1s 917us/step - loss: 0.1539 - accura
cy: 0.9340
Epoch 757/800
748/748 [==============================] - 1s 917us/step - loss: 0.1346 - accura
cy: 0.9395
Epoch 758/800
748/748 [==============================] - 1s 917us/step - loss: 0.1231 - accura
cy: 0.9426
Epoch 759/800
748/748 [==============================] - 1s 920us/step - loss: 0.1421 - accura
cy: 0.9380
Epoch 760/800
748/748 [==============================] - 1s 960us/step - loss: 0.1369 - accura
cy: 0.9364
Epoch 761/800
748/748 [==============================] - 1s 924us/step - loss: 0.1265 - accura
cy: 0.9422
Epoch 762/800
748/748 [==============================] - 1s 925us/step - loss: 0.1599 - accura
cy: 0.9320
Epoch 763/800
748/748 [==============================] - 1s 926us/step - loss: 0.1116 - accura
cy: 0.9474
Epoch 764/800
748/748 [==============================] - 1s 913us/step - loss: 0.1599 - accura
cy: 0.9304
Epoch 765/800
748/748 [==============================] - 1s 915us/step - loss: 0.1118 - accura
cy: 0.9474
Epoch 766/800
748/748 [==============================] - 1s 913us/step - loss: 0.1482 - accura
```

```
cy: 0.9344
Epoch 767/800
748/748 [==============================] - 1s 916us/step - loss: 0.1316 - accura
cy: 0.9388
Epoch 768/800
748/748 [==============================] - 1s 912us/step - loss: 0.1259 - accura
cy: 0.9428
Epoch 769/800
748/748 [==============================] - 1s 911us/step - loss: 0.1294 - accura
cy: 0.9416
Epoch 770/800
748/748 [==============================] - 1s 916us/step - loss: 0.1426 - accura
cy: 0.9378
Epoch 771/800
748/748 [==============================] - 1s 918us/step - loss: 0.1323 - accura
cy: 0.9418
Epoch 772/800
748/748 [==============================] - 1s 916us/step - loss: 0.1308 - accura
cy: 0.9419
Epoch 773/800
748/748 [==============================] - 1s 921us/step - loss: 0.1284 - accura
cy: 0.9449
Epoch 774/800
748/748 [==============================] - 1s 918us/step - loss: 0.1472 - accura
cy: 0.9368
Epoch 775/800
748/748 [==============================] - 1s 918us/step - loss: 0.1490 - accura
cy: 0.9380
Epoch 776/800
748/748 [==============================] - 1s 925us/step - loss: 0.1281 - accura
cy: 0.9427
Epoch 777/800
748/748 [==============================] - 1s 915us/step - loss: 0.1356 - accura
cy: 0.9396
Epoch 778/800
748/748 [==============================] - 1s 916us/step - loss: 0.1301 - accura
cy: 0.9443
Epoch 779/800
748/748 [==============================] - 1s 918us/step - loss: 0.1317 - accura
cy: 0.9386
Epoch 780/800
748/748 [==============================] - 1s 920us/step - loss: 0.1319 - accura
cy: 0.9426
Epoch 781/800
748/748 [==============================] - 1s 919us/step - loss: 0.1531 - accura
cy: 0.9351
Epoch 782/800
748/748 [==============================] - 1s 914us/step - loss: 0.1159 - accura
cy: 0.9461
Epoch 783/800
748/748 [==============================] - 1s 914us/step - loss: 0.1242 - accura
cy: 0.9439
Epoch 784/800
748/748 [==============================] - 1s 914us/step - loss: 0.1487 - accura
cy: 0.9348
Epoch 785/800
748/748 [==============================] - 1s 921us/step - loss: 0.1227 - accura
cy: 0.9451
Epoch 786/800
748/748 [==============================] - 1s 917us/step - loss: 0.1339 - accura
cy: 0.9408
Epoch 787/800
748/748 [==============================] - 1s 913us/step - loss: 0.1321 - accura
cy: 0.9403
Epoch 788/800
```

```
748/748 [==============================] - 1s 916us/step - loss: 0.1602 - accura
cy: 0.9299
Epoch 789/800
748/748 [==============================] - 1s 914us/step - loss: 0.1340 - accura
cy: 0.9379
Epoch 790/800
748/748 [==============================] - 1s 918us/step - loss: 0.1342 - accura
cy: 0.9398
Epoch 791/800
748/748 [==============================] - 1s 928us/step - loss: 0.1267 - accura
cy: 0.9447
Epoch 792/800
748/748 [==============================] - 1s 912us/step - loss: 0.1422 - accura
cy: 0.9382
Epoch 793/800
748/748 [==============================] - 1s 912us/step - loss: 0.1352 - accura
cy: 0.9394
Epoch 794/800
748/748 [==============================] - 1s 914us/step - loss: 0.1275 - accura
cy: 0.9439
Epoch 795/800
748/748 [==============================] - 1s 918us/step - loss: 0.1331 - accura
cy: 0.9379
Epoch 796/800
748/748 [==============================] - 1s 915us/step - loss: 0.1342 - accura
cy: 0.9391
Epoch 797/800
748/748 [==============================] - 1s 918us/step - loss: 0.1364 - accura
cy: 0.9391
Epoch 798/800
748/748 [==============================] - 1s 916us/step - loss: 0.1335 - accura
cy: 0.9428
Epoch 799/800
748/748 [==============================] - 1s 913us/step - loss: 0.1142 - accura
cy: 0.9483
Epoch 800/800
748/748 [==============================] - 1s 911us/step - loss: 0.1477 - accura
cy: 0.9371
234/234 [==============================] - 0s 552us/step - loss: 0.1188 - accura
cy: 0.9451
Accuracy: 94.51
```

In [842…

```
# four hidden layers: case one
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
batch_size = 10
no_epochs = 200
num_folds = 3
acc_per_fold = []
loss_per_fold = []

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):

    # define the keras model
    model = Sequential()
```

```
    model.add(Dense(60, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.7251912951469421; accuracy of 73.23943376541138%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.796196460723877; accuracy of 73.87964129447937%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.7453776597976685; accuracy of 72.64190912246704%
Mean Accuracy73.25366139411926
```

In [843…

```
# Four hidden layers: Case 2
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
batch_size = 10
no_epochs = 200
num_folds = 3
acc_per_fold = []
loss_per_fold = []

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):
```

```
    # define the keras model
    model = Sequential()
    model.add(Dense(120, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(80, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.9293689727783203; accuracy of 72.21511006355286%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 1.0047789812088013; accuracy of 74.86128807067871%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 1.0315414667129517; accuracy of 73.92232418060303%
Mean Accuracy is 73.66624077161153
```

In [844…

```
# Four hidden layers: Case 3
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
batch_size = 10
no_epochs = 200
num_folds = 3
acc_per_fold = []
loss_per_fold = []

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)
```

```
# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):

    # define the keras model
    model = Sequential()
    model.add(Dense(90, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(3, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.6571944355964661; accuracy of 72.08706736564636%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.7887710332870483; accuracy of 71.95902466773987%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.6944047212600708; accuracy of 71.14810347557068%
Mean Accuracy is 71.73139850298564
```

In [845…

```
# Four hidden layers: Case 4
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
batch_size = 10
no_epochs = 200
num_folds = 3
acc_per_fold = []
loss_per_fold = []
```

```
# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):

    # define the keras model
    model = Sequential()
    model.add(Dense(30, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(5, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.6686562895774841; accuracy of 72.59923219680786%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.7394677400588989; accuracy of 73.15407395362854%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.7210723757743835; accuracy of 71.83098793029785%
Mean Accuracy is 72.52809802691142
```

In [846…

```
# Four hidden layers: Case 5
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
batch_size = 10
no_epochs = 200
num_folds = 3
```

```
acc_per_fold = []
loss_per_fold = []

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):

    # define the keras model
    model = Sequential()
    model.add(Dense(5, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(5, activation='relu'))
    model.add(Dense(5, activation='relu'))
    model.add(Dense(2, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.5824558734893799; accuracy of 70.72129845619202%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.6587501168251038; accuracy of 63.038837909698486%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.6617707014083862; accuracy of 62.48399615287781%
Mean Accuracy is 65.41471083958943
```

In [847…

```
# Four hidden layers: Case 6
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
```

```
batch_size = 10
no_epochs = 200
num_folds = 3
acc_per_fold = []
loss_per_fold = []

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):

    # define the keras model
    model = Sequential()
    model.add(Dense(120, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(80, activation='relu'))
    model.add(Dense(80, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.9761484265327454; accuracy of 73.23943376541138%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.9579718708992004; accuracy of 73.15407395362854%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.746872067451477; accuracy of 73.53819608688354%
Mean Accuracy is 73.31056793530782
```

In [848…

```
# Five hidden layers based on case 2
# fix random seed for reproducibility
```

```
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
batch_size = 10
no_epochs = 200
num_folds = 3
acc_per_fold = []
loss_per_fold = []

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):

    # define the keras model
    model = Sequential()
    model.add(Dense(120, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(80, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(3, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.6574210524559021; accuracy of 63.294923305511475%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.6587494015693665; accuracy of 63.038837909698486%
------------------------------------------------------------------------
Training for fold 3 ...
```

```
Score for fold 3: loss of 0.6617264151573181; accuracy of 62.48399615287781%
Mean Accuracy is 62.93925245602926
```

In [849…

```
# Three hidden layers based on case 2
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
batch_size = 10
no_epochs = 200
num_folds = 3
acc_per_fold = []
loss_per_fold = []

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):

    # define the keras model
    model = Sequential()
    model.add(Dense(120, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(80, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 1.1548067331314087; accuracy of 73.32479953765869%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.9965495467185974; accuracy of 73.5808789730072%
```

```
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.9283364415168762; accuracy of 72.81263470649719%
Mean Accuracy is 73.23943773905437
```

In [850…

```
# Five hidden layers based on case 2
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# Define hyperparameters
batch_size = 10
no_epochs = 200
num_folds = 3
acc_per_fold = []
loss_per_fold = []

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(X, y):

    # define the keras model
    model = Sequential()
    model.add(Dense(80, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(80, activation='relu'))
    model.add(Dense(80, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))


  # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accura

  # Generate a print
    print('------------------------------------------------------------------------
    print(f'Training for fold {fold_no} ...')

  # Fit data to model
    history = model.fit(X[train], y[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=0)

  # Generate generalization metrics
    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

  # Increase fold number
    fold_no = fold_no + 1

print('Mean Accuracy is '+str(np.array(acc_per_fold).mean()))
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.9335806965827942; accuracy of 73.15407395362854%
------------------------------------------------------------------------
```

```
Training for fold 2 ...
Score for fold 2: loss of 0.9412785768508911; accuracy of 74.13572072982788%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.8984785676002502; accuracy of 71.83098793029785%
Mean Accuracy is 73.04026087125142
```

In [841…

```
print(np.array(acc_per_fold).mean())
#print('Mean Accuracy'+str(np.array(acc_per_fold)/num_folds))
from sklearn.model_selection import RepeatedKFold

# prepare the cross-validation procedure
#cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
#scores_NN = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
#print('Accuracy: %.3f (%.3f)' % (mean(scores_NN), std(scores_NN)))
```

```
69.54047481218974
```

In [705…

```
# Create function returning a compiled network
def create_network():
    network = models.Sequential()
    # Add fully connected layer with a ReLU activation function
    network.add(layers.Dense(units=50, activation='relu', input_shape=(number_of
    # Add fully connected layer with a ReLU activation function
    network.add(layers.Dense(units=20, activation='relu'))
    network.add(layers.Dense(units=5, activation='relu'))
    # Add fully connected layer with a sigmoid activation function
    network.add(layers.Dense(units=1, activation='sigmoid'))
    # Compile neural network
    network.compile(loss='binary_crossentropy', # Cross-entropy
                    optimizer='adam',
                    metrics=['accuracy']) # Accuracy performance metric

    # Return compiled network
    return network
```

In [727…

```
model_NN = KerasClassifier(build_fn=create_network,
                                epochs=10,
                                batch_size=100,
                                verbose=0)
```

```
<ipython-input-727-3be39911f496>:1: DeprecationWarning: KerasClassifier is depre
cated, use Sci-Keras (https://github.com/adriangb/scikeras) instead.
  model_NN = KerasClassifier(build_fn=create_network,
```

In [ ]: