Team 10: Nigel Wang, Susie Li,Faustin Nzitonda, Jennifer Kim

Professor: George S. Easton

ISOM_674

4 December 2022

Prediction of Clicks for On-line Advertisements

1. Introduction

This project addresses the predictions of clicks for on-line advertisements based on features of different advertisements. The data collected is ranging from Oct. 21 2014 to Oct.29 2014, 9 days data about advertisements. Our goal is to build different models to predict whether an advertisement will be clicked based on its all features, the model we built includes: logistic regression, XGBoost, CATBoost and Neural Network, and our evaluation metric is log loss of the predicted probability, because the target variable has very imbalanced categories (0:1 = 5:1), so we don't use accuracy as our evaluation metric.

2. Model Method

2.1 Logistic Regression

Logistic Regression (LR) is one of classification models which could predict the outcome of binary response variables from finding relationship between features and probability of particular outcome. It works well especially in server imbalance dataset and performs simply and efficiently. Here, since our response variable 'Click or not' has two categories, 1 or 0, we can use logistic regression to solve this problem.

```
▼ LogisticRegression
LogisticRegression(C=0.01, random_state=42)
```

2.2 XGBoost

XGBoost is a tree-based ensemble method for prediction. It uses gradient boosting calculation interiorly to get a classification output by looking over multiple iterations of the training data. For each of the iterations, it will get a small decision tree and aggregate the performance in the final step.

2.3 Neural Network

A Neural Network consists of a lot of parameters (neurons), and each of them could be regarded as a function to process with multiple inputs then getting the one or multiple output

**** all functions briefly explained in Appendix***

from each of the layers. Then the outputs will be passed to the next layer until the last one we set and the final neuron will consider all of the input logistically to get the binary result.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense_5 (Dense)              (None, 20)                1080

 dense_6 (Dense)              (None, 15)                315

 dense_7 (Dense)              (None, 10)                160

 leaky_re_lu_1 (LeakyReLU)    (None, 10)                0

 dense_8 (Dense)              (None, 5)                 55

 dense_9 (Dense)              (None, 1)                 6

=================================================================
Total params: 1,616
Trainable params: 1,616
Non-trainable params: 0
_________________________________________________________________
```

2.4 CATBoost

CATBoost is an algorithm for gradient boosting on decision trees. When the categorical variables in a dataset play a large role, CATBoost will give significant and undeniable improvement compared with other gradient boosting algorithms. CATBoost can cleverly handle all categorical variables with one-hot encoding (specified by *one_hot_max_size*). Because the majority of our variables are categorical, the CATBoost is a good method to be considered.

3. Evaluation Metric

We used *neg_log_loss* as our scoring metric when we GridSearch for the best parameters for each model as it represents the performance of a model to predict a binary class problem. When comparing the performance of each model, we took one part of the training dataset that was not used for training the model out, and get the predict the log loss of the predicted clicks for that

**** all functions briefly explained in Appendix***

unleaked part of data, and then we choose the model giving us the lowest **log loss** as our optimal model. The formula embedded in the function **log_loss** is:

$$\text{Log Loss} = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log p_i + (1-y_i)\log(1-p_i)\right)$$

3. EDA and Data Preprocessing

3.1 Preprocessing Steps

a. Raw Data Manipulation: due to the large data set, we shuffled the training data with function ***shaffle_data(df)*** and split the training data and test data to 10 parts with the function ***split_data(df, part)*** to train models and predict test data results more efficiently.
b. Interpret Datetime: Convert the feature 'hour' to two numerical features: 'weekday' and 'hour' with defined function ***conver_hour(df)***.

3.2 Exploratory Data Analysis

a. Categorical Variable

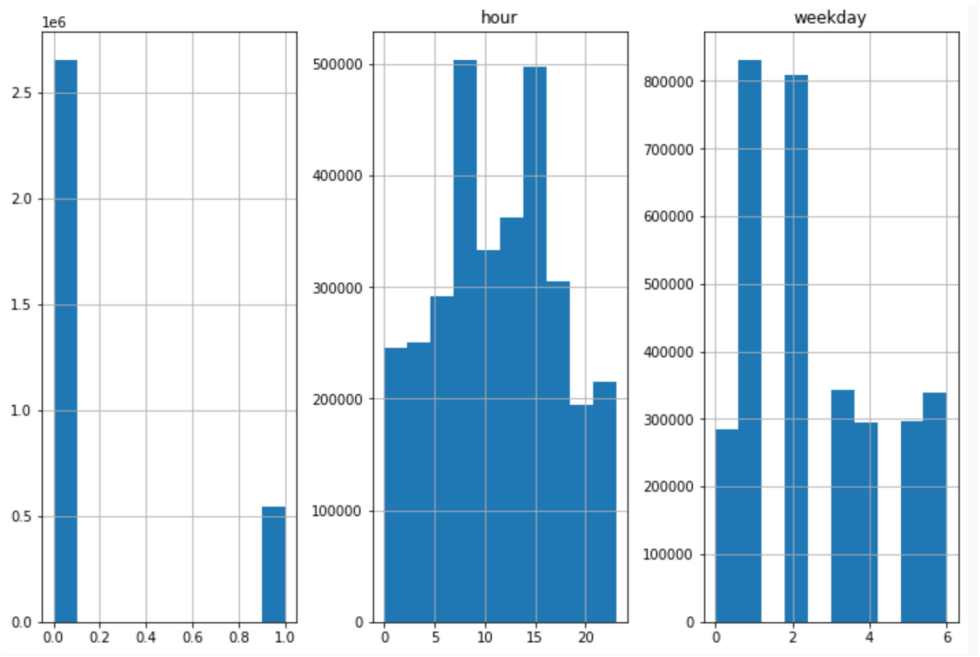
   After removing meaningless columns (like 'id'), there are still a great number of categorical variables, and some of them have lots of dimensions (hour, site_id, site_domain, site_category, app_id, app_domain, device_id, device_ip, device_model, C14, C17, C19, C20, C21), we created a summary table for number of categories in training data and test data separately by function defined function ***count_categories(df)***, which helps us get a overview of numerics for each categorical feature.

**** all functions briefly explained in Appendix***

	feature	train	test
0	hour	24	24
1	C1	7	7
2	banner_pos	7	7
3	site_id	3292	2656
4	site_domain	4041	2902
5	site_category	22	22
6	app_id	4507	3366
7	app_domain	282	198
8	app_category	26	28
9	device_id	417996	161398
10	device_ip	1348033	660498
11	device_model	6208	5197
12	device_type	5	4
13	device_conn_type	4	4
14	C14	2279	2412
15	C15	8	8
16	C16	9	9
17	C17	403	462
18	C18	4	4
19	C19	66	67
20	C20	167	163
21	C21	55	61
22	weekday	7	7

b. Numerical Variables and Target Variable: we only have two numerical variables ‘hour’ and ‘weekday’, due to the nature of datetime, we took it as continuous numerical values. The target variable only has binary outcome ‘1’ and ‘0, therefore we visualize the distribution of them to validate the reasonability.

**** all functions briefly explained in Appendix***



3.3 Feature Engineering

a. Feature Encoding for Logistic Regression, XGBoost, and Neural Network ***freq_encode(X, testDF, columns, threshold =10)***:

For both training data X and test data testDF:

(1) if the category dimension is smaller than 10, create dummy variables

(2) if the category dimension is more than 10, take frequency encoding: encodes each category to their corresponding frequencies.

(3) Rationale:

1. We don't use one-hot encoding here because some categorical variables have millions of categories and the model is high-likely to overfit the training model, and the efficiency will be greatly influenced by the exploding feature size. Therefore one-hot encoding is not feasible in this case.

2. We also considered setting a threshold and grouping all minority categories with frequency lower than the threshold as 'other', and keeping the majority group, however, this method may result in losing information of categories distribution of the minority group, which has a negative impact on model performance.

b. Change duplicated columns name for dummies variables: some categories in C14 - C21 are represented by numbers, so after converting them to string, we changed the repeated column name by the defined function ***drop_dup_colname(df).***

c. Scale numerical variables for Logistic Regression and Neural Network with defined function ***scale_num_var(df,col):*** we used MinMaxScaler to scale two numerical features: 'weekday' and 'hour' to 0-1 scale, and fit into these two models.

**** all functions briefly explained in Appendix***

d. Feature preprocessing for CATBoost: we convert the data type of all categorical variables to 'category'. Because CATBoost will encode all categorical variables automatically, we don't encode categorical features but only specify them.

4. Modeling

4.1 Process

a. We took the first part out of ten parts of data to give us a glance of each model's performance, based on the performance we decided to tune hyperparameters for XGBoost (log loss: 0.3961) and CATBoost (log loss: 0.3931).

b. After tuning the hyperparameters of the best-performed models, we fit the model to the first part (1/10) of data, then we took another part of training data that wasn't used for fitting model as our validation data (in the function ***pred_test_fun(part, method)***, if the training data is one part from 1-5, takes part 10 as validation data, if the training data is one part from 6-10, take part 1 as validation data). The defined function ***pred_test_fun(part, method)*** gives the trained model, best parameters (for XGBoost and CATBoost), and log loss on validation data. The grids of hyperparameters are shown below:

	Logistic Regression	XGBoost	CatBoost	Neural Network
	GridSearch CV	GridSearchCV	GridSearchCV	7 Layers

**** all functions briefly explained in Appendix***

Parameter	LogisticRegression(C=0.01, random_state=42, solver='lbfgs',penalty = 'l2')	param_grid = {'max_depth': range (2, 10, 1), 'n_estimators': range(60, 220, 40), 'learning_rate': [1,0.1, 0.01 , 0.05], 'tree_method':['hist'], 'eval_metric':['logloss']}	parameters = {'depth' : [4,10,15,20], 'learning_rate' : [0.02,0.1,0.3,0.8], 'iterations' : [50,100,300], 'one_hot_max_size': [5,15,50] }	SpiralNN.add(Dense(units=20,input_shape=(X.shape[1],),activation="relu",use_bias=True)) SpiralNN.add(Dense(units=15,activation="relu", use_bias=True)) SpiralNN.add(Dense(units=10,use_bias=True)) SpiralNN.add(LeakyReLU(alpha=0.05)) SpiralNN.add(Dense(units=5,activation="relu",use_bias=True)) SpiralNN.add(Dense(units=1,activation="sigmoid",use_bias=True)) SpiralNN.compile(loss='binary_crossentropy', optimizer=Optimizer,metrics=['binary_crossentropy','accuracy'])

c. We fit both models with optimal parameters for each part of data (1-10) and got their corresponding validation log loss, and applied the models with best parameters to predict the corresponding part of test data (1-10).

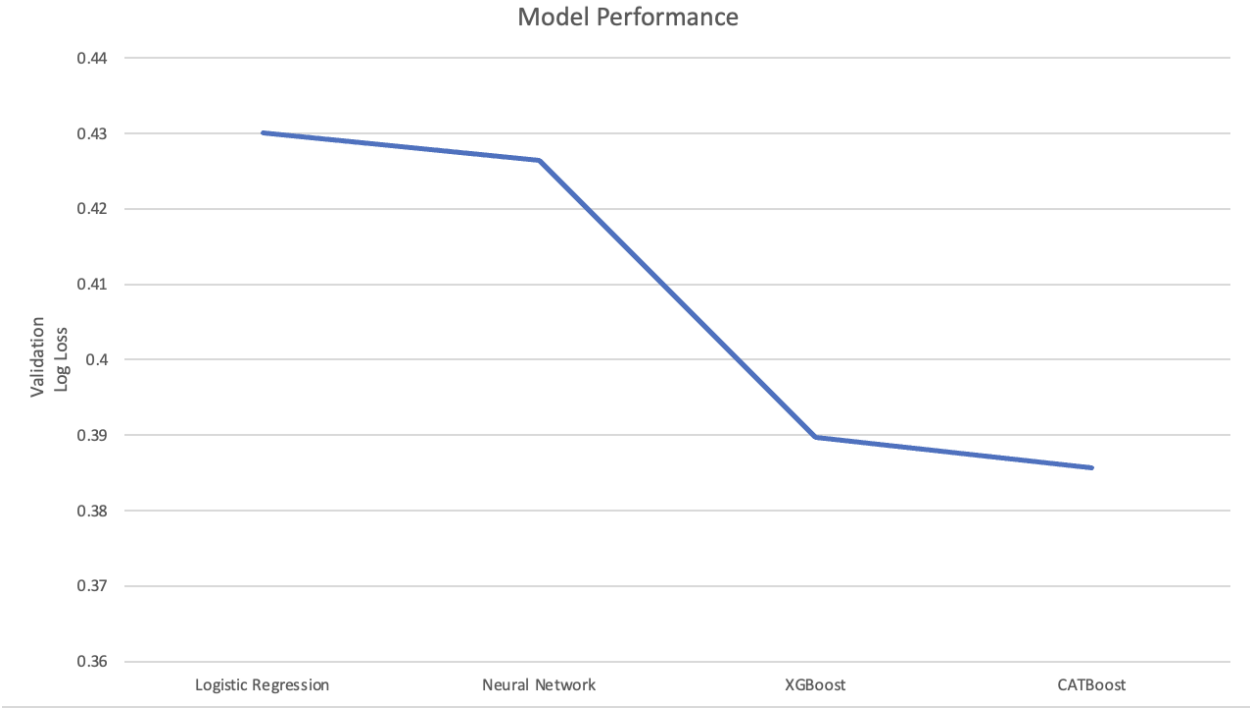
4.2 Model Result & Comparison

We used the defined function ***xgboost_bp(X,y)*** to specify the process to tune the hyperparameters in the grid, which will return the best parameters trained by one tenth of the training data. Defined function ***nn_model(NEpochs, BatchSize, Optimizer, X, y,X_test)***. We took one part of

**** all functions briefly explained in Appendix***

training data that wasn't used for training models as our validation data, and we got the log loss on the validation data for each model to compare the performance. The ***pred_test_fun(part, method)*** gave us all needed output for each part and model (predicted probability of positive label, log loss of validation data, best parameter for the specified model). The result is below:

	Logistic Regression	XGBoost	CatBoost	Neural Network
Part 1	0.4294	0.3894	0.3857	0.4173
Part 2	0.4308	0.3904	0.3855	0.4164
Part 3	0.4298	0.3895	0.3851	0.4515
Part 4	0.4295	0.3889	0.3856	0.4176
Part 5	0.4306	0.3897	0.3859	nan
Part 6	0.4310	0.3906	0.3858	0.4164
Part 7	0.4309	0.3908	0.3851	nan
Part 8	0.4303	0.3896	0.3858	0.4555
Part 9	0.4290	0.3883	0.3860	0.4190
Part 10	0.4295	0.3893	0.2784	0.4175



**** all functions briefly explained in Appendix***

5. Submission & Conclusion

We did the same data preprocessing for the test dataset after importing (***convert_hour(df)*** and change data type), and then we used tuned CATBoost to predict the probability for all the test cases.

Based on the results of different models for different parts, CATBoost gave best performance with moderate efficiency. XGBoost gave the second best performance, and neural network and logistic regression gave the worst performances. However, we can witness the tradeoff between model's performance and efficiency, the XGBoost performed a little bit inferior to CATBoost but five times more efficiently. We choose CATBoost because the majority of our features are categorical and we want to optimize our performance, in other use cases, we should consider the cost of time when we decide the model to implement.

	Min Execution Time	Max Execution Time
Logistic Regression	1 min 32 secs	2 min 46 secs
XGBoost	32 min 24 secs	52 min 53 secs
CATBoost	168 mins 12 secs	247 mins 27 secs
Neural Network	28 mins 37 secs	34 mins 17 secs

**** all functions briefly explained in Appendix***

Appendix

Shuffle the order of row for the data frame

```
def shaffle_data(df):
    df = df.sample(frac=1) # shuffle the data to split them to parts
    df = df.reset_index(drop=True)
    return df
```

Split df to how many ‘part’

```
def split_data(df, part):
    data_dic ={}

    # store splitted data in a dictionary
    for i in range(1,part+1):
        split_data = df.iloc[int(np.ceil(df.shape[0]/part)*(i-1)) :
int(np.ceil(df.shape[0]/part)*i),:]
        data_dic['data'+str(i)] = split_data

    return data_dic
```

convert the feature ‘hour’ in df to two numerical feature ‘hour’ and ‘weekday’

```
def convert_hour(df): # convert the feature 'hour' to two numerical variables
    df['hour'] = pd.to_datetime(df['hour'], format = '%y%m%d%H')
    df['weekday'] = df['hour'].dt.weekday
    df['hour'] = df['hour'].dt.hour
    return df
```


**** all functions briefly explained in Appendix***

enumerate all columns to count unique categories in that column

```
def count_categories(df): # count the number of categories for each variables
    lst = []
    for i in df.columns:
        col = i
        num_cat = (df[i].nunique() )
        lst.append(num_cat)
    return lst
```

after get the transformed and cleansed dataset, get features dataset, label dataset and test data set without 'id'

```
def get_X_y_test(part_of_data):
    traindf = 'data'+str(part_of_data)
    testdf = 'data'+str(part_of_data)

    df = all_tr[traindf]

    X = df.loc[:,['hour', 'C1', 'banner_pos', 'site_id', 'site_domain',
        'site_category', 'app_id', 'app_domain', 'app_category', 'device_id',
        'device_ip', 'device_model', 'device_type', 'device_conn_type', 'C14',
        'C15', 'C16', 'C17', 'C18', 'C19', 'C20', 'C21', 'weekday']]
    y = df['click']

    testDF = all_test[testdf]

    return X, y,testDF
```


**** all functions briefly explained in Appendix***

Based on the ‘threshold’ we set, for categorical variables with more than ‘threshold’ number of categories, we encode those categories with their frequency, otherwise we keep the original categories for that column.

```
### according to the cardinality of different categorical variables,
### we can use frequency encoder to encode those categorical variables.

### encoding for the rest categorical features by creating dummies
### (no relationship between categories, so we use one-hot encoding)

def freq_encode(X,testDF,columns, threshold=10):

    total_df = pd.concat([X,testDF])

    for i in columns:

        # if the categories for this variable is larger than threshold, use frequency encoding
        if total_df[i].nunique() > threshold:
            freq = (total_df.groupby(i).size())/len(total_df) # get the frequency
            total_df[i] = total_df[i].apply(lambda x: freq[x]) # apply frequency to the categorical variable colume

        # if categories is less than threshold, create dummies for this variable
        if total_df[i].nunique() < threshold:
            temp_dummy = pd.get_dummies(total_df[i],drop_first=True) # create a temporary dataframe for the created dummies
            total_df.drop(i,axis=1, inplace=True) # drop the varibles from original dataset and append the dummies to it
            total_df = pd.concat([total_df,temp_dummy],axis=1)
```


**** all functions briefly explained in Appendix***

```
    # split the dataset to training and test data
    X_final = total_df[0:X.shape[0]]
    testDF_final = total_df[X.shape[0]:]

    return X_final, testDF_final
```

scale the numerical variables with MinMaxScaler to 0-1 scale.

```
def scale_num_var(df, col):
    # standardize the numerical variables with MinMaxScaler for train and test data
    scaler = MinMaxScaler()
    scaler.fit(df[col])

    # transform the numerical column
    df[col] = scaler.transform(df[col])
    return df
```

enumerate all columns name, if there is repeated column name, change the later one to new one

```
def drop_dup_colname(df):
    cols=pd.Series(df.columns)

    for dup in cols[cols.duplicated()].unique():
        cols[cols[cols == dup].index.values.tolist()] = [str(dup) + '.' + str(i) if i
!= 0 else dup for i in range(sum(cols == dup))]

    # rename the columns with the cols list.
    df.columns=cols
```


**** all functions briefly explained in Appendix***

```
    return df
```

build four dense neural network, one of them is leaky ReLU, return the neural network model

```
def nn_model(NEpochs, BatchSize, Optimizer, X, y,X_test):
    SpiralNN = Sequential()

SpiralNN.add(Dense(units=20,input_shape=(X.shape[1],),activation="relu",use_bias=True
))
    SpiralNN.add(Dense(units=15,activation="relu",use_bias=True))
    SpiralNN.add(Dense(units=10,use_bias=True))
    SpiralNN.add(LeakyReLU(alpha=0.05))
    SpiralNN.add(Dense(units=5,activation="relu",use_bias=True))
    SpiralNN.add(Dense(units=1,activation="sigmoid",use_bias=True))

    SpiralNN.compile(loss='binary_crossentropy',
optimizer=Optimizer,metrics=['binary_crossentropy','accuracy'])

    StopRule =
EarlyStopping(monitor='binary_crossentropy',mode='min',verbose=0,patience=100,min_del
ta=0.0)
    FitHist = SpiralNN.fit(X,y,\
                           epochs=NEpochs,batch_size=BatchSize,verbose=0, \
                           callbacks=[StopRule])

    pred_test = SpiralNN.predict(X_test,batch_size=X.shape[0])

    return pred_test, SpiralNN
```


**** all functions briefly explained in Appendix***

get all default parameters of XGBoost, then tune them based on the parameters in the grid, return the best parameters trained from the given dataset. However, only return model and log loss on validation dataset for neural network and logistic regression because we didn't tune them.

```
def xgboost_bp(X,y):

    xgbc0 = xgb.XGBClassifier(objective='binary:logistic',

                              booster='gbtree',

                              eval_metric='logloss',

                              tree_method='hist',

                              grow_policy='lossguide',

                              use_label_encoder=False)

    default_params = {}

    gparams = xgbc0.get_params()


    #default parameters have to be wrapped in lists - even single values - so
GridSearchCV can take them as inputs

    for key in gparams.keys():

        gp = gparams[key]

        default_params[key] = [gp]


    clf0 = GridSearchCV(estimator=xgbc0, scoring='neg_log_loss',
param_grid=default_params, return_train_score=True, verbose=1, cv=3)

    clf0.fit(X, y.values.ravel())


    params = deepcopy(default_params)


    param_grid = {'max_depth': range (2, 10, 1),

                  'n_estimators': range(60, 220, 40),

                  'learning_rate': [1,0.1, 0.01 , 0.05],

                  'tree_method':['hist'],
```


**** all functions briefly explained in Appendix***

```
                'eval_metric':['logloss']}

    gcvj = np.cumsum([len(x) for x in param_grid.values()])[-1]

#iteration loop. Each selected parameter iterated separately
    for i,grid_key in enumerate(param_grid.keys()):

        #creating param_grid argument for GridSearchCV:
        #listing grid values of current iterable parameter and wrapping non-iterable
parameter single values in list
        for param_key in params.keys():
            if param_key == grid_key:
                params[param_key] = param_grid[grid_key]
            else:
                #use best parameters of last iteration
                try:
                    param_value = [clf.best_params_[param_key]]
                    params[param_key] = param_value
                #use benchmark model parameters for first iteration
                except:
                    param_value = [clf0.best_params_[param_key]]
                    params[param_key] = param_value

        #classifier instance of current iteration
        xgbc = xgb.XGBClassifier(**default_params)

        #GridSearch instance of current iteration
```


**** all functions briefly explained in Appendix***

```
        clf = GridSearchCV(estimator=xgbc, param_grid=params, scoring='neg_log_loss',
return_train_score=True, verbose=0, cv=5)
        gs_clf = clf.fit(X, y.values.ravel())

        #best parameters
        bp = gs_clf.best_params_

    return bp
```

Select a specific ‘part’ of data and model specified by ‘method’. Return the tuned model, best parameters, and log loss on validation dataset.

```
def pred_test_fun(part,method):
    X,y,testDF = get_X_y_test(part)

    columns = ['C1', 'banner_pos', 'site_id', 'site_domain', 'site_category',
    'app_id', 'app_domain', 'app_category', 'device_id', 'device_ip',
    'device_model', 'device_type', 'device_conn_type', 'C14', 'C15', 'C16',
    'C17', 'C18', 'C19', 'C20', 'C21']

    if part <= 5: # if we took one part of the first five part of data, we take the
10th part of data as the validation data
        X_val,y_val,testDF_val = get_X_y_test(10)
    if part >5:
        X_val,y_val,testDF_val = get_X_y_test(1)
    X_val_final, testDF_val_final = freq_encode(X_val,testDF_val,columns,
threshold=10) # Encode categorical variables with high cardinality
    X_val_final.columns = X_val_final.columns.astype(str)
    X_val_final = drop_dup_colname(X_val_final)
```


**** all functions briefly explained in Appendix***

```
    if method == 'CATboost':

        X,y,testDF = get_X_y_test(part)


        X[columns] = X[columns].astype('category') # change the data type of
categorical variables to category


        X.columns = X.columns.astype(str) # change all column name to string



        # rename duplicated col names

        X_final = drop_dup_colname(X)

        testDF_final = drop_dup_colname(testDF)


        #X_train, X_test,y_train, y_test =
train_test_split(X_final,y,test_size=0.3,random_state=42) # split this part of data
to train and test data


        CBC = CatBoostClassifier(cat_features=columns)



        parameters = {'depth'   : [10,15,20],

                      'learning_rate' : [0.02,0.1,0.3],

                      'iterations'    : [50,300],

                      'one_hot_max_size': [2,10,15]

                     }


        Grid_CBC = GridSearchCV(estimator=CBC,scoring='neg_log_loss',param_grid =
parameters, cv = 3, n_jobs=-1)

        Grid_CBC.fit(X_final, y)
```


**** all functions briefly explained in Appendix***

```
        #pred_test = Grid_CBC.predict(testDF_final)

        logloss = log_loss(y_val,Grid_CBC.predict_proba(X_val_final))
        best_params_cat = Grid_CBC.best_params_

        return Grid_CBC,best_params_cat,logloss



    if method == 'XGboost':
        X_final, testDF_final = freq_encode(X,testDF,columns, threshold=10) # Encode
categorical variables with high cardinality
        X_final.columns = X_final.columns.astype(str)



        # rename duplicated col names
        X_final = drop_dup_colname(X_final)
        testDF_final = drop_dup_colname(testDF_final)

        # rename duplicated col names
        X_final = drop_dup_colname(X_final)
        testDF_final = drop_dup_colname(testDF_final)



        # get the Best_Parameter for xgboost
        Best_Parameter = xgboost_bp(X_final,y)
```


**** all functions briefly explained in Appendix***

```
        xgbc0 = xgb.XGBClassifier(**Best_Parameter)
        xgbc = xgbc0.fit(X_final,y)

        #pred_test = xgbc.predict_proba(testDF_final)[:,1]

        logloss = log_loss(y_val,xgbc.predict_proba(X_val_final)[:,1])
        return xgbc,Best_Parameter,logloss

    if method =='logisitc_regression':
        X_final, testDF_final = freq_encode(X,testDF,columns, threshold=10) # Encode
categorical variables with high cardinality
        X_final.columns = X_final.columns.astype(str)

        # rename duplicated col names
        X_final = drop_dup_colname(X_final)
        testDF_final = drop_dup_colname(testDF_final)

        # Standardize the data
        col = ['hour','weekday']
        X_final_sc = scale_num_var(X_final, col) # transform training dataset
        X_testDF_sc = scale_num_var(testDF_final, col) # transform test dataset

        X_train_sc, X_test_sc,y_train_sc, y_test_sc =
train_test_split(X_final_sc,y,test_size=0.3,random_state=42) # split this part of
data to train and test data

        lr = LogisticRegression(C=0.01, random_state=42, solver='lbfgs',penalty =
'l2') # build the logistic model
```


**** all functions briefly explained in Appendix***

```
        lr = lr.fit(X_final_sc,y)


        #pred_test = lr.predict_proba(X_testDF_sc)
        logloss = log_loss(y_val,lr.predict_proba(X_val_final)[:,1])
        return lr,logloss


    if method == 'neural_network':
        X_final, testDF_final = freq_encode(X,testDF,columns, threshold=10) # Encode
categorical variables with high cardinality
        X_final.columns = X_final.columns.astype(str)


        # rename duplicated col names
        X_final = drop_dup_colname(X_final)
        testDF_final = drop_dup_colname(testDF_final)


        # Standardize the data
        col = ['hour','weekday']
        X_final_sc = scale_num_var(X_final, col) # transform training dataset
        X_testDF_sc = scale_num_var(testDF_final, col) # transform test dataset
        X_train_sc, X_test_sc,y_train_sc, y_test_sc =
train_test_split(X_final_sc,y,test_size=0.3,random_state=42) # split this part of
data to train and test data


        # get the output and NN model from the defined function
        #pred_test,SpiralNN =
nn_model(250,400,optimizers.RMSprop(learning_rate=0.001),X_final_sc,y,X_testDF_sc)


        logloss =
log_loss(y_val,SpiralNN.predict(X_val_final,batch_size=X_final_sc.shape[0]))
```


**** all functions briefly explained in Appendix***

```
        return SpiralNN,logloss
```


**** all functions briefly explained in Appendix***