EMORY | GOIZUETA BUSINESS SCHOOL

ISOM 674 Machine Learning I

Final Project

Team6

Maggie Chen, Santiago Suarez, Ethan Nadeau, Shannon Huang, Yufei Lu

12.18.2021

Exploratory Data Analysis

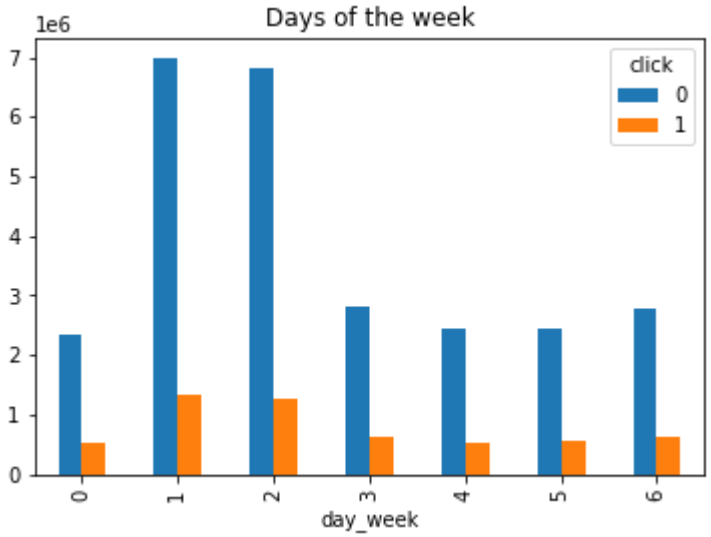
The original data set contains 23 columns with 31991089 entries. We converted the column 'hour' to DateTime datatype for further analysis.

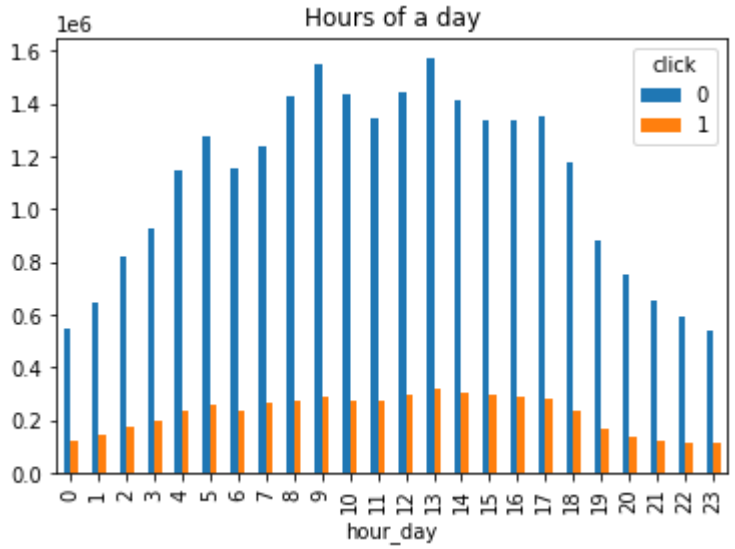
```
print(train.info())
print(train.isnull().sum().sum())
```

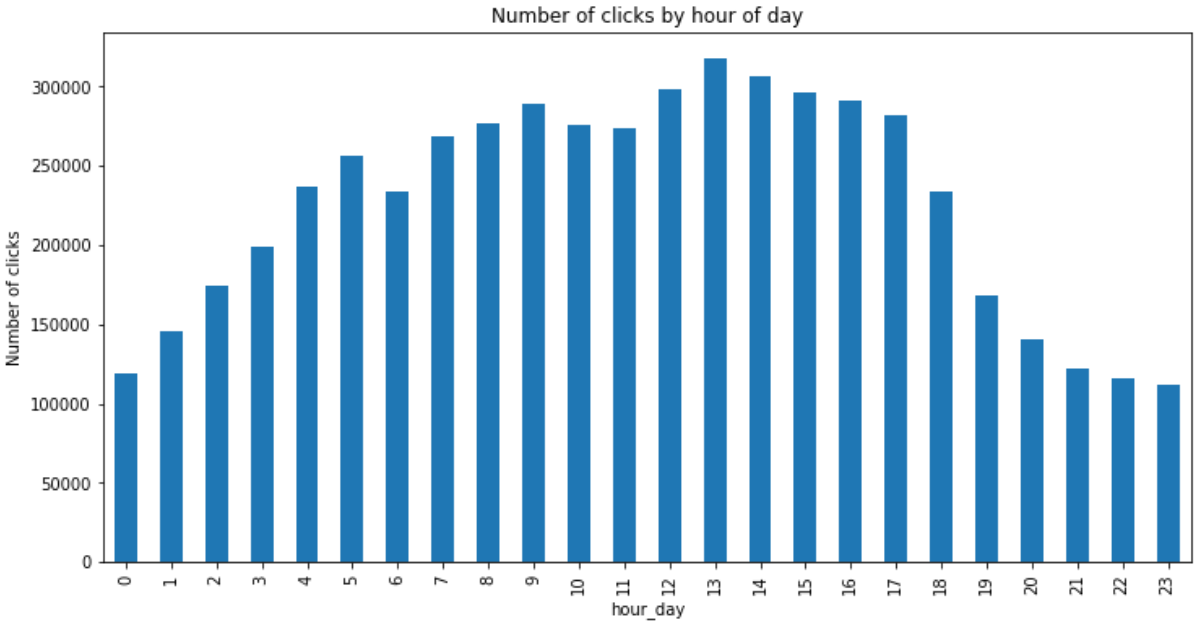
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31991090 entries, 0 to 31991089
Data columns (total 23 columns):
 #   Column            Dtype
---  ------            -----
 0   click             int64
 1   hour              datetime64[ns]
 2   C1                int64
 3   banner_pos        int64
 4   site_id           object
 5   site_domain       object
 6   site_category     object
 7   app_id            object
 8   app_domain        object
 9   app_category      object
 10  device_id         object
 11  device_ip         object
 12  device_model      object
 13  device_type       int64
 14  device_conn_type  int64
 15  C14               int64
 16  C15               int64
 17  C16               int64
 18  C17               int64
 19  C18               int64
 20  C19               int64
 21  C20               int64
 22  C21               int64
dtypes: datetime64[ns](1), int64(13), object(9)
memory usage: 5.5+ GB
None
0
```

Timestamp - Hour

We added two columns which are 'Days of the week' and 'Hours of the day' to see the distribution of clicks throughout the day and the week. From the plot, we can see that the most click happen on Monday and Tuesday of the week. And from 9:00 am to 1:00 pm during the day.

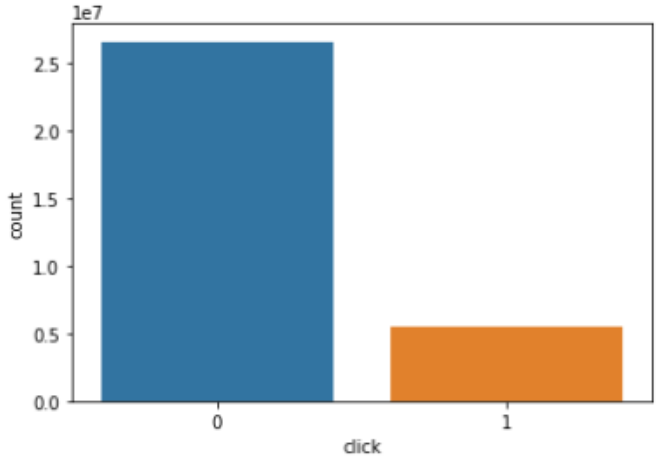






CTR

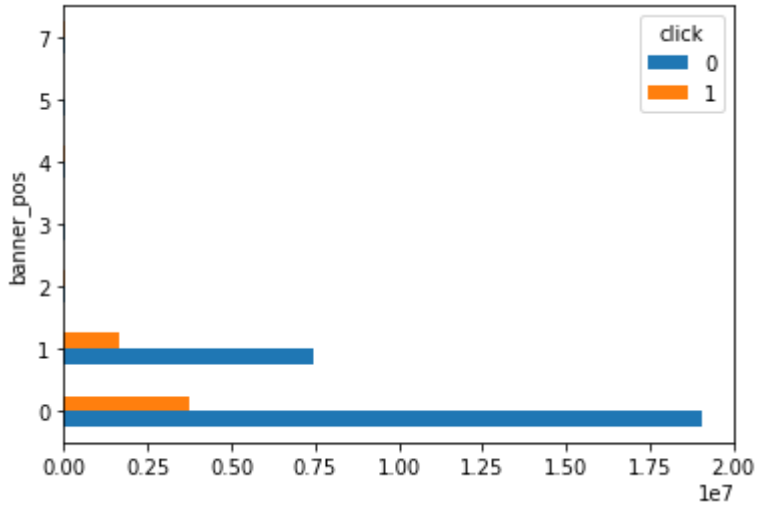
This plot shows that the number of clicks is very less as compared to the number of non-clicks. The click-through rate is almost 17% of the training data.

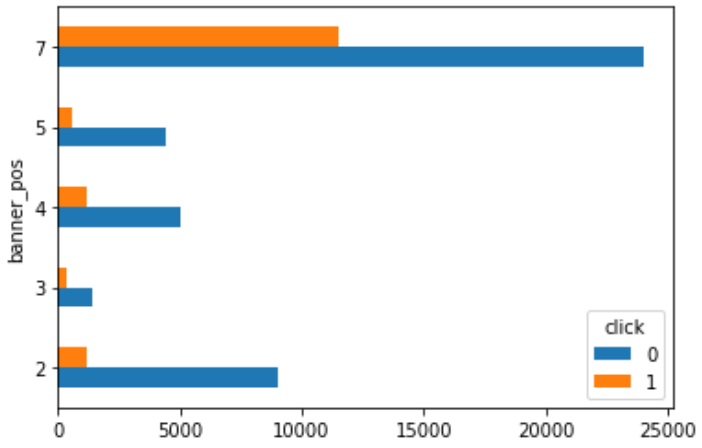



```
0    0.83011
1    0.16989
Name: click, dtype: float64
```

Banner Position

From the plot we can see that the ads are being clicked on the most when the banner position is 7.





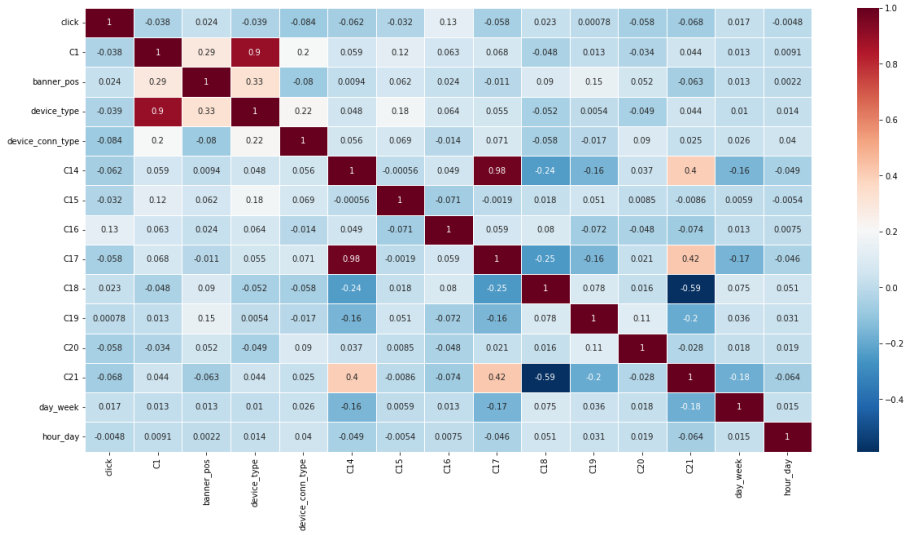
Site Features

```
site_features = ['site_id', 'site_domain', 'site_category']
train[site_features].describe()
```

	site_id	site_domain	site_category
count	31991090	31991090	31991090
unique	4581	7341	26
top	85f751fd	c4e18dd6	50e219e0
freq	11170690	11598113	12731910

Feature Selection

We can remove the features that are irrelevant and not useful to the data before building our models.



Data Preparation

Sample Selection:

Since the original data size is huge, we created a chunk with 5,000,000 rows of data that follows the original data distribution. In the following models, we are using this chunk to train models.

Feature Engineering

At the start of the project, we analyzed the dataset to find certain combinations or transformations of variables that are more informative than the raw data. We focused on two engineered features: (1) a variable encoding time with cyclicality and (2) removing duplicated information in the site_id and app_id variables. For (1), we encoded days to reflect an arbitrary distance from the first instance. For the time in hours, we first encoded it to reflect the number of seconds since midnight, then we used the sin() and cos() functions to encode the data with cyclicality. This better represents how “time works”. For instance, if we encoded each time variable from midnight to noon based on a 24 hour

clock, a machine learning model would perceive a time at hour 24 and at hour 0 as the maximum distance away from each other. In actuality, they are right next to each other, so we wanted the data to reflect that. For (2), we noticed that there was a repeated string in both the app_id and site_id columns. We realized that it must reflect a placeholder for when either an ad is seen on an app or on a website. Therefore, we removed both site_id and app_id and replaced the values with a binary column.

Feature Encoding

Besides what we did before, our models require extra feature encoding. The model fitting process built in the Python package 'Sklearn' does not take non-numerical values as column 'site_domain', 'site_category', 'app_domain', and so on. To solve the problem, we applied LabelEncoder from sklearn.preprocessing to encode labels with values between 0 and the number of total levels -1.

```
from sklearn import preprocessing

def feature_encoding(dataframe):
    feats_to_encode = ['C1', 'banner_pos', 'site_id', 'site_domain', 'site_category',
                       'app_id', 'app_domain', 'app_category', 'device_id', 'device_ip', 'device_model',
                      'C14', 'C15', 'C16', 'C17', 'C18', 'C19', 'C20', 'C21', 'Day', 'Weekday', 'Hour', 'Weekday']
    for column in feats_to_encode:
        enc = preprocessing.LabelEncoder()
        dataframe[column] = enc.fit_transform(dataframe[column])

    print(dataframe.info())
    print(dataframe.head())

    return dataframe

df3 = feature_encoding(df2)
```

Standardization:

Furthermore, we wanted to control the effect of variable scaling. So, we scaled the dataset using the function z=(x-u)/s, where u is the mean and s is the standard deviation.

```
def standardize_data(dataframe):
    scaler = preprocessing.StandardScaler().fit(dataframe)
    dataframe_scaled = scaler.transform(dataframe)
    return pd.DataFrame(dataframe_scaled, columns=dataframe.columns)
```

Logistic Regression

Since we are predicting the probability of a binary variable, click, logistic regression is one of the models that we came up with. We split the training data into training and testing set. This action could

help us to get a better understanding of how the model performed before applying models to the real test dataset.

Hyperparameter Tuning:

After data preparation, we fitted logistic regression with the default parameters. We also did hyperparameter tuning by Grid Search CV to optimize the model. The tuning parameters are the regularization method (L1 or L2) and the penalty strength (C). We used the log loss value as a standard to evaluate the model performance on the test data.

```
from sklearn.metrics import log_loss,make_scorer

def perform_lr(X_train, y_train, penaltys):
    C_s = [0.01,0.1]
    # C_s = [0.00001,0.0001,0.001, 0.01, 0.1, 1]
    parameters_to_tune = dict(penalty=penaltys, C=C_s)
    model = LogisticRegression(solver='liblinear')
    LogLoss = make_scorer(log_loss, greater_is_better=False, needs_proba=True)
    grid = GridSearchCV(model, parameters_to_tune, cv=5, scoring=LogLoss, verbose=2)
    grid.fit(X_train, y_train)

    print('Best score: ', -grid.best_score_)
    print('Best params: ', grid.best_params_)

    return grid

best_grid = perform_lr(X_train, y_train, penaltys = ['l1'])
```

```
from sklearn import metrics

lrtest=perform_lr(x_Train,y_Train,penaltys=['l2'])
y_pred=lrtest.predict_proba(x_Test)
log_loss= metrics.log_loss(y_Test,y_pred,eps=1e-15,normalize=True,sample_weight=None, labels=None)
log_loss
```

```
Fitting 5 folds for each of 2 candidates, totalling 10 fits
[CV] END ................................C=0.01, penalty=l2; total time=   0.1s
[CV] END ................................C=0.01, penalty=l2; total time=   0.0s
[CV] END ................................C=0.01, penalty=l2; total time=   0.0s
[CV] END ................................C=0.01, penalty=l2; total time=   0.0s
[CV] END ................................C=0.01, penalty=l2; total time=   0.0s
[CV] END .................................C=0.1, penalty=l2; total time=   0.0s
[CV] END .................................C=0.1, penalty=l2; total time=   0.0s
[CV] END .................................C=0.1, penalty=l2; total time=   0.1s
[CV] END .................................C=0.1, penalty=l2; total time=   0.1s
[CV] END .................................C=0.1, penalty=l2; total time=   0.1s
Best score:  0.44423229965775174
Best params:  {'C': 0.1, 'penalty': 'l2'}

0.4432072114770645
```

The best parameters of logistic regression are C=0.1 and penalty= L2, which stands for Ridge Regression. The best log loss score for the logistic regression model is 0.443. It was not so bad per se, but we still need to compare with other models to decide on the final one that we are going to use for prediction.

Random Forest

Sample Selection:

Because the random forest is a method of ensemble learning. In this model, we created a smaller sample to do hyperparameter tuning in grid search. The new sample size is 50,000, with the distribution of the original data.

Feature Selection:

Because the ID's in the dataset is measured as categorical data, which leads to both long computation time and overfitting issues in random forest data, so we dropped these columns: "site_id", "app_id", "device_id" and "device_ip". It has significantly improved the calculation speed.

```
id 49905
click 2
hour 33
C1 7
banner_pos 6
site_id 1000
site_domain 842
site_category 18
app_id 1000
app_domain 76
app_category 22
device_id 9012
device_ip 40095
device_model 2558
device_type 4
device_conn_type 4
C14 571
C15 7
C16 8
C17 181
C18 4
C19 43
C20 155
C21 38
```

Hyperparameter Tuning:

After data preparation, we fitted the random forest model with the default parameters. As a result, the log loss is 0.535. However, with the gridsearchCV we tuned the model with n_estimators and min_samples_leaf, we achieved a much lower log loss.
The tuning process is exhibited below:

```
def perform_rf(X_train, y_train):
    param_grid = {
    'n_estimators': [1000,1500,2000],
    'min_samples_leaf':[3,5,7]
     }

    model = RandomForestClassifier(n_jobs = 3)
    grid = GridSearchCV(model, param_grid=param_grid, cv=5, scoring=scorer, verbose=2 )
    grid.fit(X_train, y_train)



    print('Best score: ', -grid.best_score_)
    print('Best params: ', grid.best_params_)
    print(grid.cv_results_)
    return grid

best_grid = perform_rf(X_train, y_train)
```

The best score be got in the gridsearchcv is 0.41 with the parameter combination:
{'min_samples_leaf': 7, 'n_estimators': 1000}
In the validation set, the best model gives a log loss value of 0.42 and thus we got a better result than logistic regression.

However, a huge drawback of random forest is its running time. Even if we have taken a smaller sample, the running time of gridsearchCV also exceeds 10 minutes. Thus, the model has poor application in the larger sample.

XGBoost

We used the same sample selection and feature selection as Random Forest and logistics regression. XGboost takes a while to run, especially when we are doing hyperparameter tuning or cross-validation scores. Therefore, we first used 50000 rows to run and tune and choose the best Xgb model, and then we fitted the best model with the best parameters on all 5,000,000 rows. One of the disadvantages of XGBoost is that it is very hard to tune because it has so many parameters so we tried 3 different approaches.

1.- Baseline

We first fit base XGBclassifier to have a baseline of what we have to improve.

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
              gamma=0, gpu_id=-1, importance_type=None,
              interaction_constraints='', learning_rate=0.300000012,
              max_delta_step=0, max_depth=6, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=100, n_jobs=12,
              num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

We used 10 folds of cross-validation and out-of-sample testing to evaluate the model. The first score was .59 but improved to .419 after the last encoding we did.

```
from sklearn.metrics import log_loss

y_pred = xgbc.predict_proba(x_test)

log_loss = sklearn.metrics.log_loss(y_test, y_pred, eps=1e-15, normalize=True, sample_weight=None, labels=None)

log_loss
```

```
0.4192677120835939
```

```
scores = cross_val_score(xgbc, X, y, cv=5, scoring = 'neg_log_loss') # cross-validation scores

print("Cross validation log loss: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2)) #estimate mean and va

scores
```

2.- Hyperparameter Tuning using randomized search

Here we use a pipeline and randomize search to tune the model. It works by giving a list of parameters we want to tune. The more parameters we included the more it takes to run. Up to 2 hours running. We found out that the fewer parameters we tune the better the model was performing. We found the

parameters here and we tried to find the best range based on our loss function. Also, note that we can add an eval metric that xgboost will try to minimize or maximize depending on the metric.

https://xgboost.readthedocs.io/en/stable/parameter.html

```
steps = [('scaler', StandardScaler()), ('XGBoost', xgb.XGBClassifier(eval_metric = 'logloss', random_state = 42))]
```

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline(steps)
```

```
parameters = {'XGBoost__booster': ['gbtree', 'gblinear', 'dart'],
              'XGBoost__max_depth': np.arange(3, 18, 1),
              'XGBoost__reg_alpha': [1e-5, 1e-2, 0.1, 1],
              'XGBoost__learning_rate': np.arange(0.1,1,.2),
              'XGBoost__eval_metric' : ['logloss'],
              'XGBoost__colsample_bytree': np.arange(0.1, 1.05, 0.05),
              'XGBoost__objective': ['reg:logistic','binary:logistic']}
```

```
from sklearn.model_selection import RandomizedSearchCV
model = RandomizedSearchCV(pipeline, parameters, n_iter=5, scoring= "neg_log_loss", cv=4)
```

We also used out-of-sample performance and cross-validation to evaluate the model. There was definitely a big improvement.

```
print('Tuned Model Parameters: {}'.format(model.best_params_))
```

```
Tuned Model Parameters: {'XGBoost__reg_alpha': 0.1, 'XGBoost__objective': 'reg:logistic', 'XGBoost__max_depth': 12, 'XGBoost__l
earning_rate': 0.1, 'XGBoost__eval_metric': 'logloss', 'XGBoost__colsample_bytree': 0.3500000000000001, 'XGBoost__booster': 'da
rt'}
```

```
elapsed_time = (time.time() - start_time)/60
print('Time to tune the model: {0:0.2f} min.'.format(elapsed_time))
```

```
Time to tune the model: 153.58 min.
```

3.- Parameter Tuning with Gridsearch

Next approach is tuning parameters one by one using gridsearch. We focused on max _depth, min_child_weight, gamma, subsample, colsample_bytree, and reg_alpha. We go in order and we update the parameters that have been tuned on the calculations of the new parameters. Therefore we estimate the best parameters taking into consideration the previous ones.

```
param_test1 = {
 'max_depth':range(3,10,2),
 'min_child_weight':range(1,6,2)
}
gsearch1 = GridSearchCV(estimator = XGBClassifier( learning_rate =0.1, n_estimators=140, max_depth=5,
 min_child_weight=1, gamma=0, subsample=0.8, colsample_bytree=0.8,
 objective= 'binary:logistic', nthread=4, scale_pos_weight=1, seed=27),
 param_grid = param_test1, scoring='neg_log_loss',n_jobs=4, cv=5)
gsearch1.fit(x_train,y_train)
gsearch1.best_params_, gsearch1.best_score_
```

```
C:\Users\santi\Anaconda3\lib\site-packages\xgboost\sklearn.py:1224: UserWarning: The use of label encoder in XGBClassifier is d
eprecated and will be removed in a future release. To remove this warning, do the following: 1) Pass option use_label_encoder=F
alse when constructing XGBClassifier object; and 2) Encode your labels (y) as integers starting with 0, i.e. 0, 1, 2, ..., [num
_class - 1].
  warnings.warn(label_encoder_deprecation_msg, UserWarning)

[13:01:23] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.0/src/learner.cc:1115: Starting in XGBoost 1.3.
0, the default evaluation metric used with the objective 'binary:logistic' was changed from 'error' to 'logloss'. Explicitly se
t eval_metric if you'd like to restore the old behavior.

({'max_depth': 5, 'min_child_weight': 5}, -0.41215822752764647)
```

Once we have tuned all of the parameters mentioned above, we fit the model and evaluate it. Then we compare it with the other two models. Third model gave us the best result.

```
In [94]: import sklearn
         from sklearn.metrics import log_loss

         y_pred = xgb2.predict_proba(x_test)

         log_loss = sklearn.metrics.log_loss(y_test, y_pred, eps=1e-15, normalize=True, sample_weight=None, labels=None)

         log_loss

Out[94]: 0.4115084456340523
```

However, now that we had tuned the models, we refitted them but now with all 5000000 rows. Model 2 performed the best with .394.

```
In [133]: import sklearn
          from sklearn.metrics import log_loss

          y_pred = xgb2.predict_proba(x_test)

          log_loss = sklearn.metrics.log_loss(y_test, y_pred, eps=1e-15, normalize=True, sample_weight=None, labels=None)

          log_loss

Out[133]: 0.4052140240763463
```

```
In [182]: import sklearn
          from sklearn.metrics import log_loss

          y_pred = model.predict_proba(x_test)

          log_loss = sklearn.metrics.log_loss(y_test, y_pred, eps=1e-15, normalize=True, sample_weight=None, labels=None)

In [183]: log_loss

Out[183]: 0.39376112972468896
```

Model results

Model	Out of Sample Log Loss
Neural Nets	**.69**
Random Forest	**.42**
Logistic Regression	**.44**
XGboost	**.39**

XGboost yielded the best Log Loss and therefore we decided to use our XGboost model to make predictions on the Test data. Predictions were formatted in an excel file as requested.

Note: We tried a Neural Network method; however, we quickly realized that XGboost systematically outperformed NN methods. Therefore, we diverted our attention to perfecting the XGBoost method.