Introduction:

The rise of social media and digital technology enables marketers to narrow down target audience of their ads. They want to show the ads to those who may generate genuine incentives to buy or take other actions to increase conversion rate and making the whole process cost-effective. Since most digital display ads are paid by per click, fraud clicks would be a waste their budget. Being the largest mobile market in the world with over 1 billion smart devices in active use every month, China suffers from large scales of fraudulent traffic. The data I used for this project is from China’s largest independent big data service platform TalkingData. It handles 3 billion clicks per day and 90% are potentially fraudulent. The goal of this project is to build algorithm to detect fraudulent click traffic for mobile app ads. TalkingData provided the training data set and test data set. The train data set has the following features:

· ip: ip address of click.

· app: app id for marketing.

· device: device type id of user mobile phone (e.g., iphone 6 plus, iphone 7, huawei mate 7, etc.)

· os: os version id of user mobile phone

· channel: channel id of mobile ad publisher

· click\_time: timestamp of click (UTC)

· attributed\_time: if user download the app for after clicking an ad, this is the time of the app download

· is\_attributed: the target that is to be predicted, indicating the app was downloaded

The test data has one more feature on “click\_id” which is reference for making predictions and the “is\_attributed” feature is not included and to be predicted. This target variable only has two values, 0 or 1. If its value equals to 1, it is a non-fraudulent click, otherwise, it is a fraudulent click. The data set is very large, with 185 million rows in training set and 19 million rows in the test set. Due to limited kernel memory to process the whole dataset, I only used first 1000000 rows in both datasets.

|  |
| --- |
| import pandas as pd import numpy as np import matplotlib.pyplot as plt %matplotlib inline import seaborn as sns dftrain=pd.read\_csv('train.csv', nrows=1000000) dftest=pd.read\_csv('test.csv', nrows=1000000) |

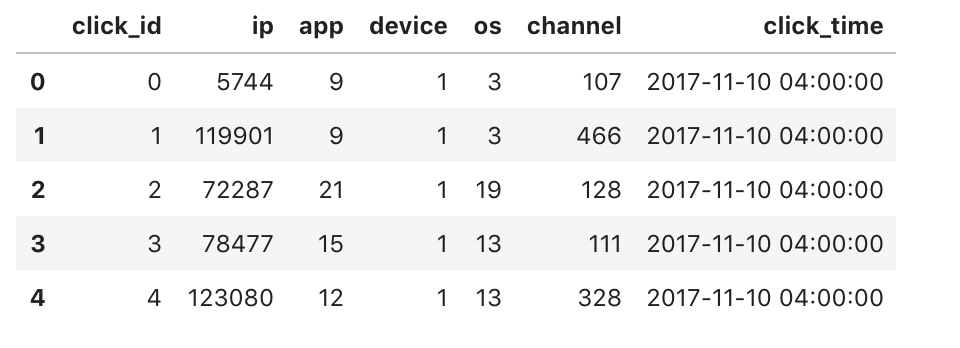
This is the first 5 rows of train data set:

|  |
| --- |
| dftrain.head() |



This is the first 5 rows of test data set:

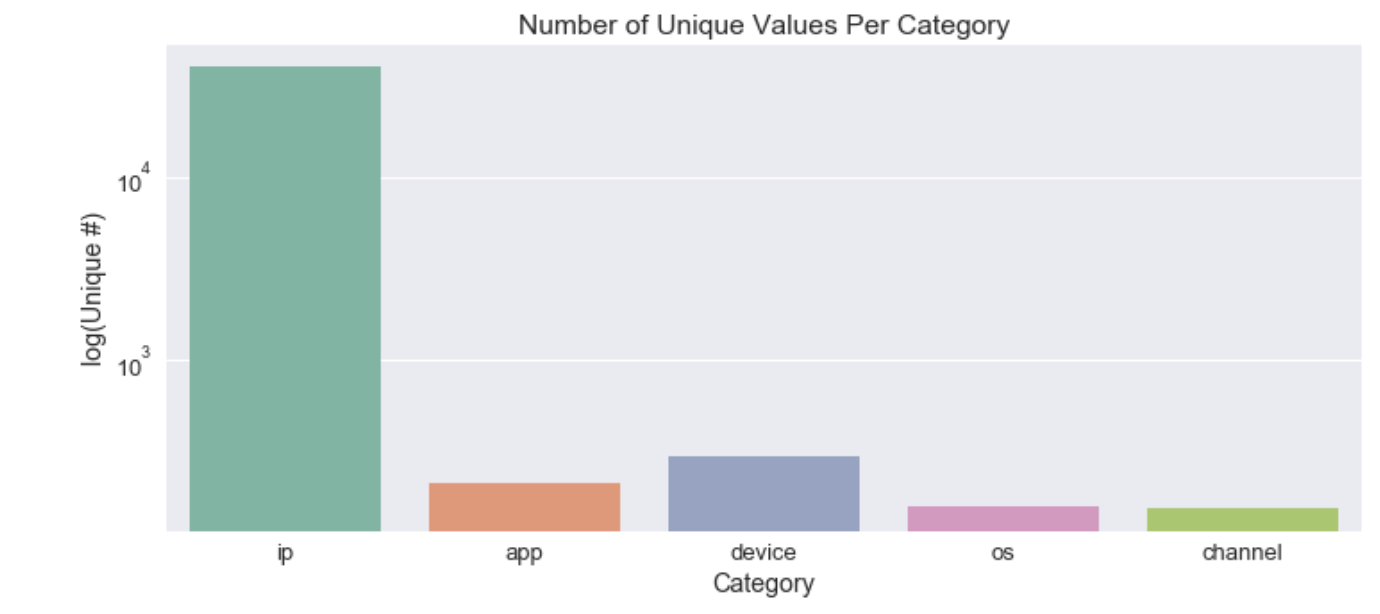
|  |
| --- |
| dftest.head() |



Descriptive Statistics:

This table shows the number of unique values in each category. There most unique ip address and least unique channels, which is the ad app.

|  |
| --- |
| plt.figure(figsize=(12,5)) cols=['ip','app','device','os','channel'] #len()returns the number of items in an object cate=[len(dftrain[col].unique()) for col in cols] sns.set(font\_scale=1.3) ax=sns.barplot(cols, cate, palette="Set2", log=True) ax.set(xlabel='Category',ylabel='log(Unique #)',title='Number of Unique Values Per Category') |



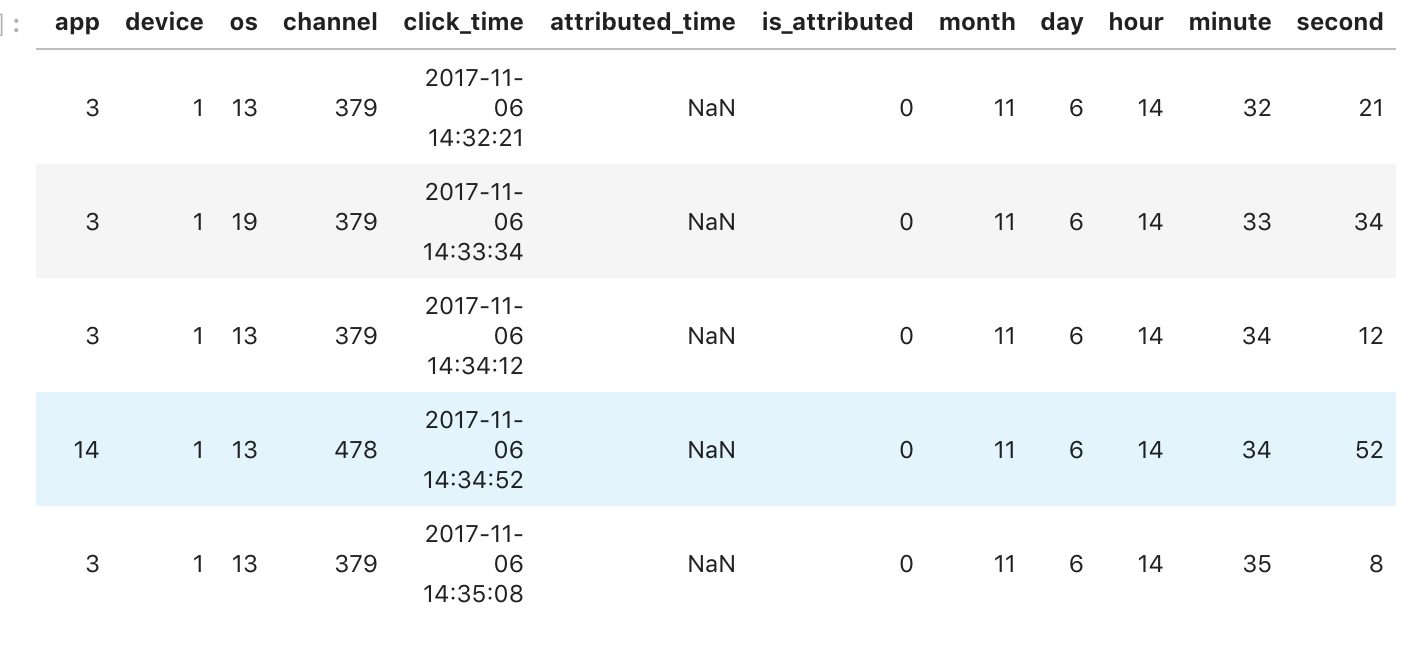
The following bar graph shows that in the train data most clicks are attributed as fraudulent.

|  |
| --- |
| #percentage of fraudulent clicks plt.figure(figsize=( 5 , 5 )) sns.set(font\_scale= 1 ) mean = (dftrain.is\_attributed.values == 1 ).mean() ax = sns.barplot([ 'Not Fraudulent (1)' , 'Fraudulent (0)' ], [mean, 1-mean]) ax.set(ylabel= 'Probability' , title= 'Target value distribution' ) |



I first changed “click\_time” to date time so that I can extract day, hour, minute and second from this feature and create separate columns for each time variable. This allowed me to explore relationships of clicking time and fraudulent clicks. It also made possible to compare other features within a certain time period.

|  |
| --- |
| dftrain['click\_time']=pd.to\_datetime(dftrain['click\_time']) dftrain['month'] = dftrain.click\_time.dt.month dftrain['day'] = dftrain.click\_time.dt.day dftrain['hour'] = dftrain.click\_time.dt.hour dftrain['minute'] = dftrain.click\_time.dt.minute dftrain['second'] = dftrain.click\_time.dt.second  dftrain.head() |



The first 1000000 rows in train set include data from November 6, 14:00 to 16:00.

|  |
| --- |
| hourcol=dftrain['hour'] hourcol.max()  16 hourcol.min()  14 |

To see how many clicks are attributed as fraudulent versus non fraudulent in each hour, I grouped by hour and is\_attributed and count the number of fraudulent (0) and non fraudulent (1) clicks. The result shows that most fraudulent clicks happened at 16:00 pm.

|  |
| --- |
| dftrain.groupby(['hour', 'is\_attributed'])['is\_attributed'].count() hour is\_attributed 14 0 48 15 0 434  1 1 16 0 997825  1 1692 Name: is\_attributed, dtype: int64 |

Creating Indicators for fraud clicks:

ip\_day\_hour\_channelcount:

I assumed that a click might be fraudulent if an ip address on the same day and same hour downloaded large amount of apps. I grouped by ip-day-hour and count the number of channels associated with a given IP address within each hour on the same day. The result shows some ip addresses downloaded over 6000 of apps within an hour and others only have 1 or 2 downloads. Those ip addresses with over 6000 downloads are probably fraudulent ones. And this difference indicates that the ip-day-hour combination is meaningful in detecting fraud clicks.

|  |
| --- |
| ga=dftrain[['ip','day','hour','channel']].groupby(by=['ip','day','hour'])[['channel']].count().reset\_index().rename(index=str, columns={'channel':'ip\_day\_hour\_channelcount'})  ga.sort\_values('ip\_day\_hour\_channelcount', ascending=False) |

Ip\_app\_channelcount:

I then grouped by ip and app and count channel. This reveals the number of downloads occured at same app per ip address.

|  |
| --- |
| gb=dftrain[['ip','app','channel']].groupby(by=['ip','app'])[['channel']].count().reset\_index().rename(index=str, columns={'channel':'ip\_app\_channelcount'}) gb.sort\_values('ip\_app\_channelcount', ascending=False) |

The results shows that some ip address clicked over 1000 app ads from the same app and other only clicked 1 or two times. The ip addresses associated with large numbers are suspect of fraud clicks.

Ip\_channel\_appcount:

I’ve created a 3rd combination by grouping ip and channel and count app. This would give us the number of same app that an ip address downloaded from various channels.

|  |
| --- |
| gc=dftrain[['ip','channel','app']].groupby(['ip','channel'])['app'].count().reset\_index().rename(index=str, columns={'app':'ip\_channel\_appcount'}) gc.sort\_values('ip\_channel\_appcount', ascending=False).head() |

The result shows that some ip address clicked the same app for downloads over 800 times. To see if those are fraudulent, I’ve located some ip addresses with big numbers in the whole train dataset. And I found out that those are attributed as fraud clicks. This can verify that the new variable that I created is a valid indicator to discern frauds.

|  |
| --- |
| dftrain.loc[170043] dftrain.loc[225080:225090] |

To further prove that the three combined variables are meaningful for prediction, I tested them on a new data set that included all non fraudulent clicks from the train dataset. If the numbers for the new variables are small and consistent, it would validate the previous work on variable grouping.

|  |
| --- |
| all\_real=dftrain[dftrain['is\_attributed']==1] na=all\_real[['ip','day','hour','channel']].groupby(by=['ip','day','hour'])[['channel']].count().reset\_index().rename(index=str, columns={'channel':'count\_channel'}) na.sort\_values('count\_channel', ascending=False).head() |

|  |
| --- |
| nb=all\_real[['ip','app','channel']].groupby(by=['ip','app'])[['channel']].count().reset\_index().rename(index=str, columns={'channel':'ip\_app\_channelcount'}) nb.sort\_values('ip\_app\_channelcount', ascending=False).head() |

|  |
| --- |
| nc=all\_real[['ip','channel','day','hour','app']].groupby(['ip','channel','day','hour'])['app'].count().reset\_index().rename(index=str, columns={'app':'ip\_time\_channel\_appcount'}) nc.sort\_values('ip\_time\_channel\_appcount', ascending=False).head() |

I sorted value from largest to smallest and found out that the largest would be single digits that’s below 5. This means that there is a huge value difference of the three variables between non fraudulent clicks and fraudulent ones.

After validation, I merged the three new columns into the train data set. The merged dataset will then be used as input to build models.

|  |
| --- |
| ga=dftrain[['ip','day','hour','channel']].groupby(by=['ip','day','hour'])[['channel']].count().reset\_index().rename(index=str, columns={'channel':'ip\_day\_hour\_channelcount'}) dftrainM = pd.merge(dftrainM,ga, on=['ip','day','hour'], how='left')  gb=dftrain[['ip','app','channel']].groupby(by=['ip','app'])[['channel']].count().reset\_index().rename(index=str, columns={'channel':'ip\_app\_channelcount'}) dftrainM=pd.merge(dftrainM,gb, on=['ip','app'], how='left')  gc=dftrain[['ip','channel','app']].groupby(['ip','channel'])['app'].count().reset\_index().rename(index=str, columns={'app':'ip\_channel\_appcount'}) dftrainM= pd.merge(dftrain,gc, on=['ip','channel'], how='left') |

Training model:

The first model I choose is logistic regression. It has a sigmoidal shape which is best fit for analyzing a dataset with binary dependent variable. Since the variable that we’re predicting “is\_atributted” has only two possible outcomes 1 or 0, this model might be valid to use. I splitted the merged dataset (dftrainM) into four sub groups: train\_x, train\_y, test\_x, and test\_y. I extracted relevant columns (‘ip', 'day', 'hour', 'channel', 'ip\_channel\_appcount', 'ip\_day\_hour\_channelcount', 'ip\_app\_channelcount') from train\_x along with train\_y to feed into the model. Then I used test\_x as input to predict “is\_attributed” feature. By comparing the result of this prediction to test\_y, I can decide how accurate is this predictive model.

|  |
| --- |
| from sklearn.linear\_model import LogisticRegression clf=LogisticRegression() x=dftrainM y=dftrain['is\_attributed'] from sklearn.model\_selection import train\_test\_split from sklearn.metrics import classification\_report x\_train, x\_test, y\_train, y\_test=train\_test\_split(x,y) |

This model cannot process date times so I dropped “click\_time” and “attributed\_time”. It will not affect the prediction because I already extracted and appended ‘day,’ ‘hour,’ ‘minute,’ ‘second’ columns to the data frame, and I didn’t use “attributed\_time” as an indicator:

|  |
| --- |
| x\_train = x\_train.drop('click\_time', axis=1) x\_test = x\_test.drop('click\_time', axis=1) x\_train = x\_train.drop('attributed\_time', axis=1) x\_test=x\_test.drop('attributed\_time', axis=1) x\_train=x\_train[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']] |

|  |
| --- |
| clf.fit(x\_train,y\_train)  LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,  intercept\_scaling=1, max\_iter=100, multi\_class='ovr', n\_jobs=1,  penalty='l2', random\_state=None, solver='liblinear', tol=0.0001,  verbose=0, warm\_start=False)  predict=clf.predict(x\_test[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']]) |

To look at the first 10 prediction:

|  |
| --- |
| predict[:10]  array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0]) |

To make sure there are 1 in the prediction:

|  |
| --- |
| y\_test.nlargest()  44456 1 202118 1 470919 1 763475 1 432767 1 Name: is\_attributed, dtype: int64 |

Print the classification report:

|  |
| --- |
| print(classification\_report(y\_test, predict))  precision recall f1-score support   0 1.00 1.00 1.00 249551  1 0.00 0.00 0.00 449  avg / total 1.00 1.00 1.00 250000 |

Getting the score for this prediction:

|  |
| --- |
| clf.score(x\_test[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']],y\_test)  0.998204 |

Model 2:

I also made predictions using KNeighborsClassifier. This is a non-parametric algorithm and classify based on similarity. It looks at how closely the sample’s features resemble the training data set and make decision accordingly.

|  |
| --- |
| from sklearn.neighbors import KNeighborsClassifier knn=KNeighborsClassifier() knn.fit(x\_train,y\_train)  KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',  metric\_params=None, n\_jobs=1, n\_neighbors=5, p=2,  weights='uniform') |

|  |
| --- |
| kpredict=knn.predict(x\_test[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']]) |

|  |
| --- |
| print(classification\_report(y\_test, predict))  precision recall f1-score support   0 1.00 1.00 1.00 249551  1 0.00 0.00 0.00 449  avg / total 1.00 1.00 1.00 250000 |

|  |
| --- |
| knn.score(x\_test[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']],y\_test)  0.9982 |

The classification report tells that this model didn’t predict very well for negative outcomes: the precision and recall when “is\_attritbute” is 1 are both 0. To optimize the model, I tested how would number of neighbors impact precision of this model.

|  |
| --- |
| params = {'n\_neighbors': [2, 3, 4, 7, 10, 20]}  grid = GridSearchCV(knn, param\_grid=params) grid.fit(x\_train, y\_train) Grid.best\_estimator\_ |

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',  
 metric\_params=None, n\_jobs=1, n\_neighbors=4, p=2,  
 weights='uniform')

This output tells that the model is most accurate among the 6 numbers, when number of neighbors equal 4.

|  |
| --- |
| print(classification\_report(y\_test, predict)) |

precision recall f1-score support  
  
 0 1.00 1.00 1.00 249519  
 1 0.00 0.00 0.00 481  
  
avg / total 1.00 1.00 1.00 250000

|  |
| --- |
| knn.score(x\_test[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']],y\_test) |

0.998076

Model 3:

The last classifier used is the decision tree. It uses a tree-like model of decisions and their possible consequences to make predictions on certain input.

|  |
| --- |
| from sklearn.tree import DecisionTreeClassifier tree=DecisionTreeClassifier() tree.fit(x\_train,y\_train)  DecisionTreeClassifier(class\_weight=None, criterion='gini', max\_depth=None,  max\_features=None, max\_leaf\_nodes=None,  min\_impurity\_decrease=0.0, min\_impurity\_split=None,  min\_samples\_leaf=1, min\_samples\_split=2,  min\_weight\_fraction\_leaf=0.0, presort=False, random\_state=None,  splitter='best') |

|  |
| --- |
| tree\_predict=tree.predict(x\_test[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']]) |

|  |
| --- |
| print(classification\_report(y\_test, tree\_predict))  precision recall f1-score support   0 1.00 1.00 1.00 249551  1 0.30 0.25 0.27 449  avg / total 1.00 1.00 1.00 250000 |

|  |
| --- |
| tree.score(x\_test[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']],y\_test)  0.997608 |

The decision tree classifier has the highest score comparing to the others. So I used the real test data on this model.

|  |
| --- |
| x=x[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']] tree.fit(x,y)  DecisionTreeClassifier(class\_weight=None, criterion='gini', max\_depth=None,  max\_features=None, max\_leaf\_nodes=None,  min\_impurity\_decrease=0.0, min\_impurity\_split=None,  min\_samples\_leaf=1, min\_samples\_split=2,  min\_weight\_fraction\_leaf=0.0, presort=False, random\_state=None,  splitter='best') |

Change the click time to date time in test data set and extract each time element out from this column to form separate and independent columns:

|  |
| --- |
| dftest['click\_time']=pd.to\_datetime(dftest['click\_time']) # adding a month column dftest['month'] = dftest.click\_time.dt.month #adding a day column dftest['day'] = dftest.click\_time.dt.day #adding a hour column dftest['hour'] = dftest.click\_time.dt.hour dftest\_drop=dftest.drop('click\_time',axis=1) |

Merge the 3 new variables into the test data frame:

|  |
| --- |
| ta=dftest[['ip','day','hour','channel']].groupby(by=['ip','day','hour'])[['channel']].count().reset\_index().rename(index=str, columns={'channel':'ip\_day\_hour\_channelcount'}) dftestM = pd.merge(dftest\_drop,ta, on=['ip','day','hour'], how='left')  tb=dftest[['ip','app','channel']].groupby(by=['ip','app'])[['channel']].count().reset\_index().rename(index=str, columns={'channel':'ip\_app\_channelcount'}) dftestM=pd.merge(dftestM,tb, on=['ip','app'], how='left')  tc=dftest[['ip','channel','app']].groupby(['ip','channel'])['app'].count().reset\_index().rename(index=str, columns={'app':'ip\_channel\_appcount'}) dftestM= pd.merge(dftestM,tc, on=['ip','channel'], how='left') |

Using test data set to make prediction on the click through decision tree:

|  |
| --- |
| real\_treepredict=tree.predict(dftestM[['ip','day','hour','channel','ip\_channel\_appcount','ip\_day\_hour\_channelcount','ip\_app\_channelcount']]) |

|  |
| --- |
| print(real\_treepredict[(len(real\_treepredict)-500):])  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] |