Customer Prediction

August 9, 2020

1 Customer Chun Prediction

** using Gradient Boosted Trees to predict Mobile Customer Departure**

1.0.1 Contents

1.1 Step 1 — Background

A Large number of loyal customer are a key to scucess of a business, however. Losing customers is costly for any business. Identifying unhappy customers early on gives you a chance to offer them incentives to stay. This notebook describes using machine learning (ML) for the automated identification of unhappy customers, also known as customer churn prediction. ML models rarely give perfect predictions though, so this notebook is also about how to incorporate the relative costs of prediction mistakes when determining the financial outcome of using ML.It's important for a mobile phone operator to retain their customer and prevent churn.

Can a mobile phone operator known beforehand that a customer is going to leave?. Seems like I can always find fault with my provider du jour! And if my provider knows that I'm thinking of leaving, it can offer timely incentives–I can always use a phone upgrade or perhaps have a new feature activated–and I might just stick around. Incentives are often much more cost effective than losing and reacquiring a customer.

---

[2]:
```
#pip install xgboost
```

1.1.1 Step 2 — Importing Scikit-learn

[4]:
```
# Import the python libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import auc,confusion_matrix,precision_score,␣
↪accuracy_score,recall_score,f1_score
from xgboost import XGBClassifier
```

1.1.2 Step 3 — Data Source

The dataset we use is publicly available and was mentioned in the book Discovering Knowledge in Data by Daniel T.Larose. It is attributed by the author to the university of California Irvine Repository of Machine Learning Datasets. Let's download and read the datset

[ ]:
```
!wget http://dataminingconsultant.com/DKD2e_data_sets.zip
!unzip -o DKD2e_data_sets.zip
```

[5]:
```
churn = pd.read_csv('./Data sets/churn.txt')
pd.set_option('display.max_columns', 500)
churn.head()
```

[5]:
```
  State  Account Length  Area Code     Phone Int'l Plan VMail Plan  \
0    KS             128        415  382-4657         no        yes
1    OH             107        415  371-7191         no        yes
2    NJ             137        415  358-1921         no         no
3    OH              84        408  375-9999        yes         no
4    OK              75        415  330-6626        yes         no

   VMail Message  Day Mins  Day Calls  Day Charge  Eve Mins  Eve Calls  \
0             25     265.1        110       45.07     197.4         99
1             26     161.6        123       27.47     195.5        103
2              0     243.4        114       41.38     121.2        110
3              0     299.4         71       50.90      61.9         88
4              0     166.7        113       28.34     148.3        122

   Eve Charge  Night Mins  Night Calls  Night Charge  Intl Mins  Intl Calls  \
0       16.78       244.7           91         11.01       10.0           3
1       16.62       254.4          103         11.45       13.7           3
2       10.30       162.6          104          7.32       12.2           5
3        5.26       196.9           89          8.86        6.6           7
4       12.61       186.9          121          8.41       10.1           3
```

```
   Intl Charge  CustServ Calls  Churn?
0         2.70               1  False.
1         3.70               1  False.
2         3.29               0  False.
3         1.78               2  False.
4         2.73               3  False.
```

By modern standards, it's a relatively small dataset, with only 3,333 records, where each record uses 21 attributes to describe the profile of a customer of an unknown US mobile operator. The attributes are:

- `State`: the US state in which the customer resides, indicated by a two-letter abbreviation; for example, OH or NJ
- `Account Length`: the number of days that this account has been active
- `Area Code`: the three-digit area code of the corresponding customer's phone number
- `Phone`: the remaining seven-digit phone number
- `Intl Plan`: whether the customer has an international calling plan: yes/no
- `VMail Plan`: whether the customer has a voice mail feature: yes/no
- `VMail Message`: presumably the average number of voice mail messages per month
- `Day Mins`: the total number of calling minutes used during the day
- `Day Calls`: the total number of calls placed during the day
- `Day Charge`: the billed cost of daytime calls
- `Eve Mins, Eve Calls, Eve Charge`: the billed cost for calls placed during the evening
- `Night Mins`, `Night Calls`, `Night Charge`: the billed cost for calls placed during nighttime
- `Intl Mins`, `Intl Calls`, `Intl Charge`: the billed cost for international calls
- `CustServ Calls`: the number of calls placed to Customer Service
- `Churn?`: whether the customer left the service: true/false

The last attribute, `Churn?`, is known as the target attribute–the attribute that we want the ML model to predict. Because the target attribute is binary, our model will be performing binary prediction, also known as binary classification.

Let's begin exploring the data:

1.1.3 Step 4 — Exploratory Data Analysis

[6]:
```
# To see all the columns and data types
churn.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 21 columns):
State             3333 non-null object
Account Length    3333 non-null int64
Area Code         3333 non-null int64
Phone             3333 non-null object
Int'l Plan        3333 non-null object
VMail Plan        3333 non-null object
```

```
VMail Message     3333 non-null int64
Day Mins          3333 non-null float64
Day Calls         3333 non-null int64
Day Charge        3333 non-null float64
Eve Mins          3333 non-null float64
Eve Calls         3333 non-null int64
Eve Charge        3333 non-null float64
Night Mins        3333 non-null float64
Night Calls       3333 non-null int64
Night Charge      3333 non-null float64
Intl Mins         3333 non-null float64
Intl Calls        3333 non-null int64
Intl Charge       3333 non-null float64
CustServ Calls    3333 non-null int64
Churn?            3333 non-null object
dtypes: float64(8), int64(8), object(5)
memory usage: 546.9+ KB
```

[7]:
```
churn.isna().sum()
```

[7]:
```
State             0
Account Length    0
Area Code         0
Phone             0
Int'l Plan        0
VMail Plan        0
VMail Message     0
Day Mins          0
Day Calls         0
Day Charge        0
Eve Mins          0
Eve Calls         0
Eve Charge        0
Night Mins        0
Night Calls       0
Night Charge      0
Intl Mins         0
Intl Calls        0
Intl Charge       0
CustServ Calls    0
Churn?            0
dtype: int64
```

[8]:
```
# Frequency table for each categorical feature
for column in churn.select_dtypes(include=['object']).columns:
    display(pd.crosstab(index=churn[column], columns='%observations', normalize␣
 ↪='columns' ))
```

```python
#Histograms for each numerics features
display(churn.describe())
%matplotlib inline
hist = churn.hist(bins=30, sharey= True , figsize=(10,10))
```

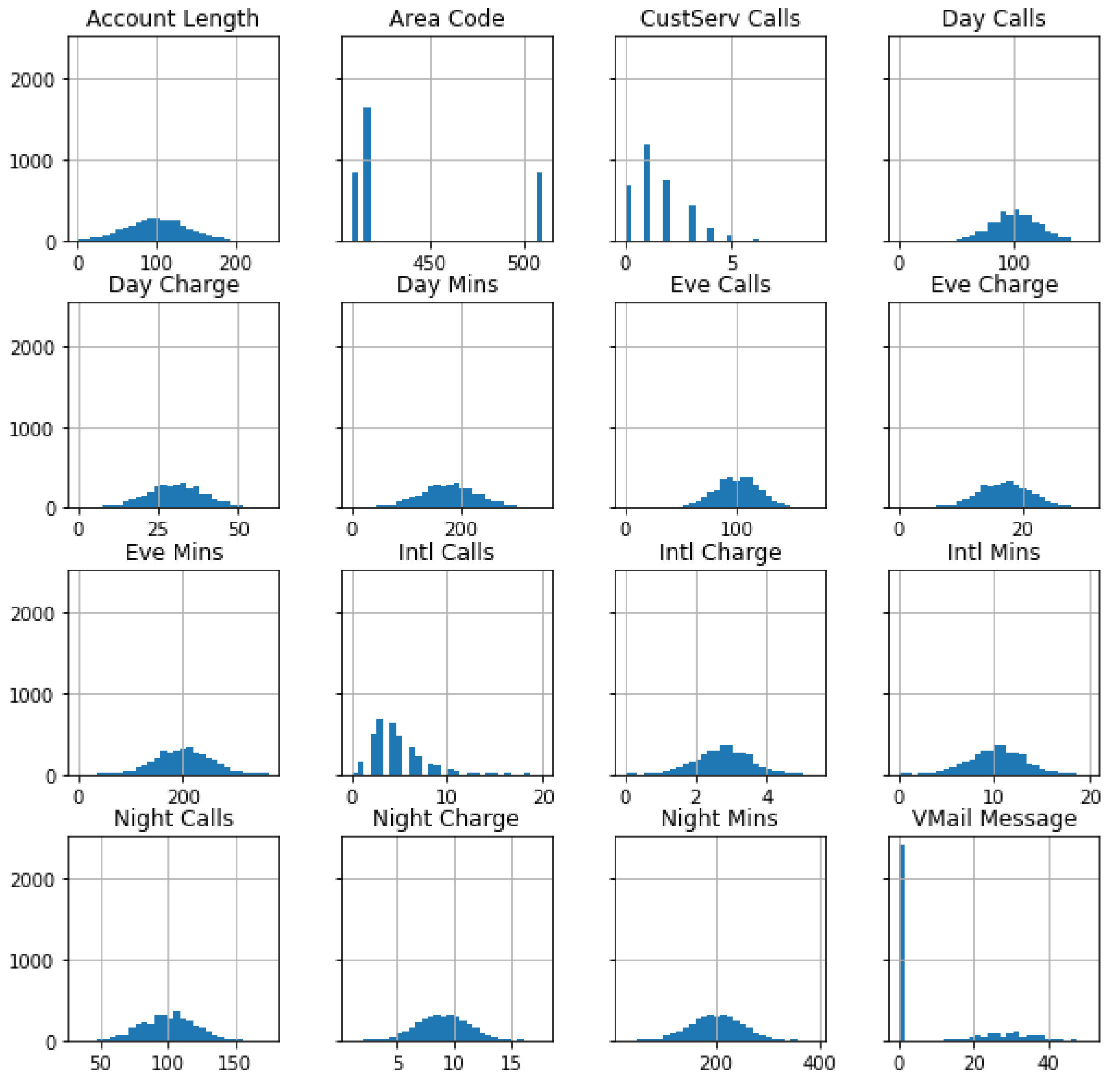
```
col_0  %observations
State
AK          0.015602
AL          0.024002
AR          0.016502
AZ          0.019202
CA          0.010201
CO          0.019802
CT          0.022202
DC          0.016202
DE          0.018302
FL          0.018902
GA          0.016202
HI          0.015902
IA          0.013201
ID          0.021902
IL          0.017402
IN          0.021302
KS          0.021002
KY          0.017702
LA          0.015302
MA          0.019502
MD          0.021002
ME          0.018602
MI          0.021902
MN          0.025203
MO          0.018902
MS          0.019502
MT          0.020402
NC          0.020402
ND          0.018602
NE          0.018302
NH          0.016802
NJ          0.020402
NM          0.018602
NV          0.019802
NY          0.024902
OH          0.023402
OK          0.018302
OR          0.023402
PA          0.013501
RI          0.019502
```

SC	0.018002
SD	0.018002
TN	0.015902
TX	0.021602
UT	0.021602
VA	0.023102
VT	0.021902
WA	0.019802
WI	0.023402
WV	0.031803
WY	0.023102

col_0	%observations
Phone	
327-1058	0.0003
327-1319	0.0003
327-3053	0.0003
327-3587	0.0003
327-3850	0.0003
327-3954	0.0003
327-4795	0.0003
327-5525	0.0003
327-5817	0.0003
327-6087	0.0003
327-6179	0.0003
327-6194	0.0003
327-6764	0.0003
327-6989	0.0003
327-8495	0.0003
327-8732	0.0003
327-9289	0.0003
327-9341	0.0003
327-9957	0.0003
328-1206	0.0003
328-1222	0.0003
328-1373	0.0003
328-1522	0.0003
328-1768	0.0003
328-2110	0.0003
328-2236	0.0003
328-2478	0.0003
328-2647	0.0003
328-2982	0.0003
328-3266	0.0003
...	...
421-7205	0.0003
421-7214	0.0003

```
421-7270         0.0003
421-8141         0.0003
421-8535         0.0003
421-8537         0.0003
421-9034         0.0003
421-9144         0.0003
421-9401         0.0003
421-9752         0.0003
421-9846         0.0003
422-1471         0.0003
422-1799         0.0003
422-2571         0.0003
422-3052         0.0003
422-3454         0.0003
422-4241         0.0003
422-4394         0.0003
422-4956         0.0003
422-5264         0.0003
422-5350         0.0003
422-5865         0.0003
422-5874         0.0003
422-6685         0.0003
422-6690         0.0003
422-7728         0.0003
422-8268         0.0003
422-8333         0.0003
422-8344         0.0003
422-9964         0.0003

[3333 rows x 1 columns]


col_0       %observations
Int'l Plan
no                0.90309
yes               0.09691


col_0       %observations
VMail Plan
no               0.723372
yes              0.276628


col_0   %observations
Churn?
False.       0.855086
True.        0.144914
```

```
       Account Length   Area Code  VMail Message     Day Mins    Day Calls  \
count     3333.000000  3333.000000    3333.000000  3333.000000  3333.000000
mean       101.064806   437.182418       8.099010   179.775098   100.435644
std         39.822106    42.371290      13.688365    54.467389    20.069084
min          1.000000   408.000000       0.000000     0.000000     0.000000
25%         74.000000   408.000000       0.000000   143.700000    87.000000
50%        101.000000   415.000000       0.000000   179.400000   101.000000
75%        127.000000   510.000000      20.000000   216.400000   114.000000
max        243.000000   510.000000      51.000000   350.800000   165.000000

        Day Charge     Eve Mins    Eve Calls   Eve Charge   Night Mins  \
count  3333.000000  3333.000000  3333.000000  3333.000000  3333.000000
mean     30.562307   200.980348   100.114311    17.083540   200.872037
std       9.259435    50.713844    19.922625     4.310668    50.573847
min       0.000000     0.000000     0.000000     0.000000    23.200000
25%      24.430000   166.600000    87.000000    14.160000   167.000000
50%      30.500000   201.400000   100.000000    17.120000   201.200000
75%      36.790000   235.300000   114.000000    20.000000   235.300000
max      59.640000   363.700000   170.000000    30.910000   395.000000

       Night Calls  Night Charge    Intl Mins   Intl Calls  Intl Charge  \
count  3333.000000   3333.000000  3333.000000  3333.000000  3333.000000
mean    100.107711      9.039325    10.237294     4.479448     2.764581
std      19.568609      2.275873     2.791840     2.461214     0.753773
min      33.000000      1.040000     0.000000     0.000000     0.000000
25%      87.000000      7.520000     8.500000     3.000000     2.300000
50%     100.000000      9.050000    10.300000     4.000000     2.780000
75%     113.000000     10.590000    12.100000     6.000000     3.270000
max     175.000000     17.770000    20.000000    20.000000     5.400000

       CustServ Calls
count     3333.000000
mean         1.562856
std          1.315491
min          0.000000
25%          1.000000
50%          1.000000
75%          2.000000
max          9.000000
```



We can see immediately that: - `State` appears to be quite evenly distributed - `Phone` takes on too many unique values to be of any practical use. It's possible parsing out the prefix could have some value, but without more context on how these are allocated, we should avoid using it. - Only 14% of customers churned, so there is some class imabalance, but nothing extreme. - Most of the numeric features are surprisingly nicely distributed, with many showing bell-like gaussianity. `VMail Message` being a notable exception (and `Area Code` showing up as a feature we should convert to non-numeric).

[8]:
```
#Let to drop phone column
churn.drop(['Phone'], axis=1, inplace=True)
churn['Area Code'] = churn["Area Code"].astype(object)
```

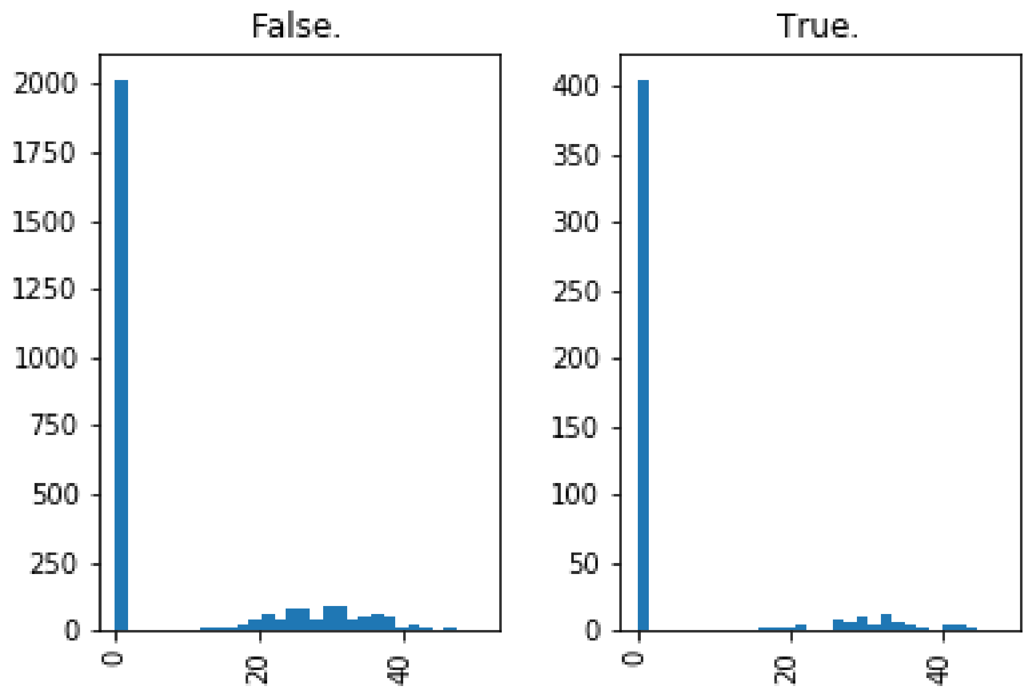
[10]:
```
# Next let's look at the relationship between each feature and our target␣
↪variable
```

```python
for column in churn.select_dtypes(include =['object']).columns:
    if column != 'Churn?':
        display(pd.crosstab(index= churn[column], columns = churn['Churn?'],␣
 ↪normalize = 'columns'))

for column in churn.select_dtypes(exclude=['object']).columns:
    print(column)
    hist = churn[[column, 'Churn?']].hist(by= 'Churn?', bins=30)
    plt.show()
```

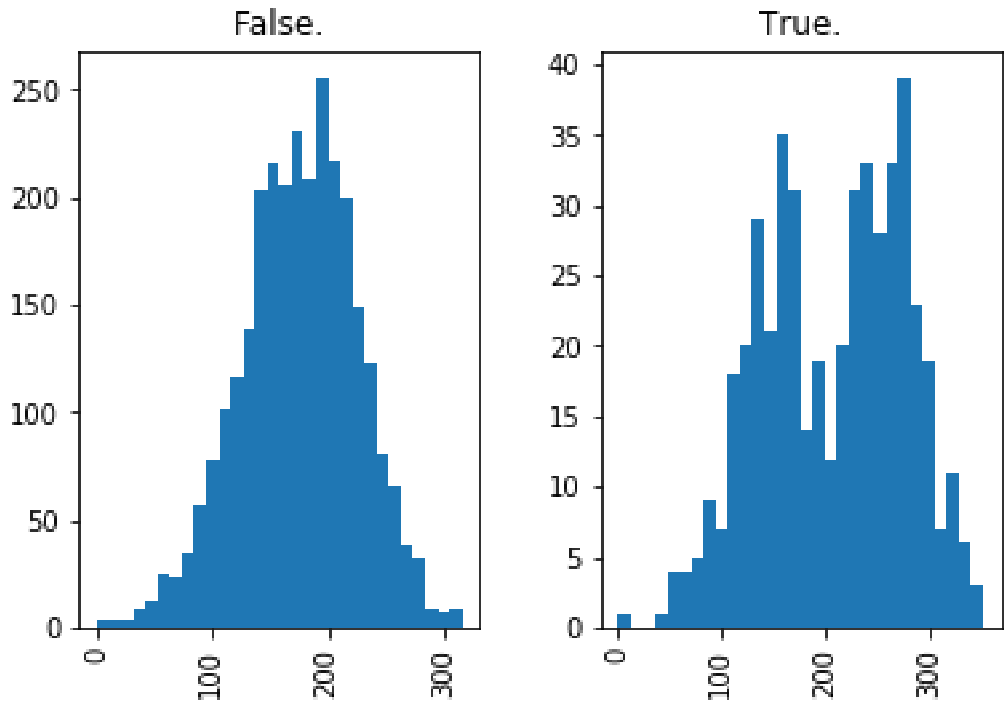
```
Churn?    False.     True.
State
AK      0.017193  0.006211
AL      0.025263  0.016563
AR      0.015439  0.022774
AZ      0.021053  0.008282
CA      0.008772  0.018634
CO      0.020000  0.018634
CT      0.021754  0.024845
DC      0.017193  0.010352
DE      0.018246  0.018634
FL      0.019298  0.016563
GA      0.016140  0.016563
HI      0.017544  0.006211
IA      0.014386  0.006211
ID      0.022456  0.018634
IL      0.018596  0.010352
IN      0.021754  0.018634
KS      0.020000  0.026915
KY      0.017895  0.016563
LA      0.016491  0.008282
MA      0.018947  0.022774
MD      0.018596  0.035197
ME      0.017193  0.026915
MI      0.020000  0.033126
MN      0.024211  0.031056
MO      0.019649  0.014493
MS      0.017895  0.028986
MT      0.018947  0.028986
NC      0.020000  0.022774
ND      0.019649  0.012422
NE      0.019649  0.010352
NH      0.016491  0.018634
NJ      0.017544  0.037267
NM      0.019649  0.012422
NV      0.018246  0.028986
NY      0.023860  0.031056
```

```
OH      0.023860  0.020704
OK      0.018246  0.018634
OR      0.023509  0.022774
PA      0.012982  0.016563
RI      0.020702  0.012422
SC      0.016140  0.028986
SD      0.018246  0.016563
TN      0.016842  0.010352
TX      0.018947  0.037267
UT      0.021754  0.020704
VA      0.025263  0.010352
VT      0.022807  0.016563
WA      0.018246  0.028986
WI      0.024912  0.014493
WV      0.033684  0.020704
WY      0.023860  0.018634


Churn?       False.     True.
Area Code
408        0.251228  0.252588
415        0.497895  0.488613
510        0.250877  0.258799


Churn?        False.     True.
Int'l Plan
no          0.934737  0.716356
yes         0.065263  0.283644


Churn?        False.     True.
VMail Plan
no          0.704561  0.834369
yes         0.295439  0.165631


Account Length
```

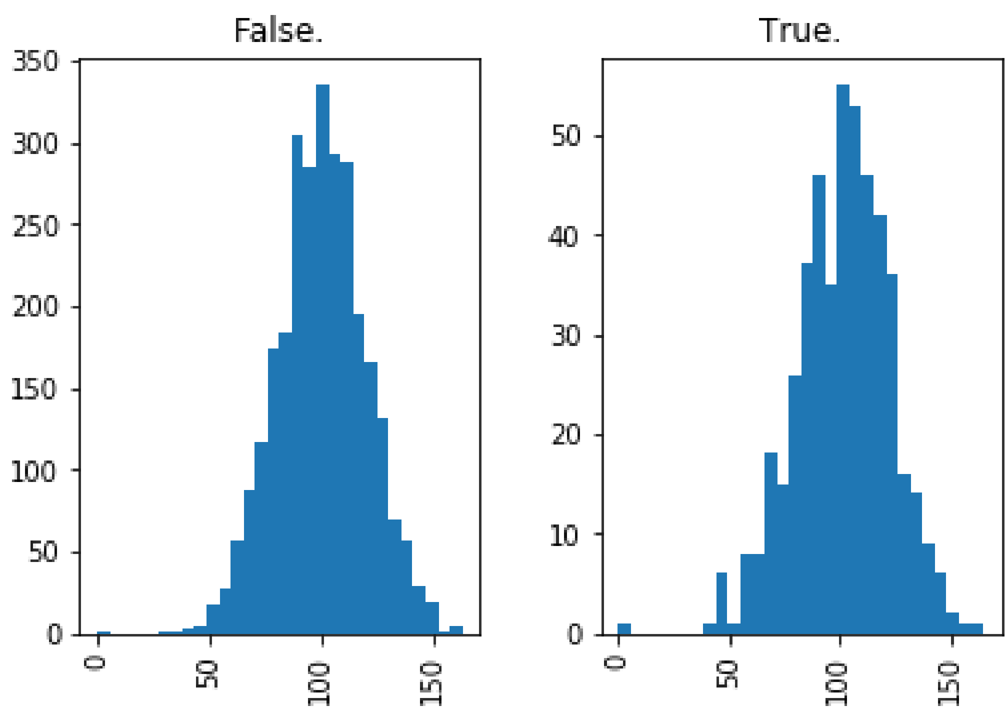
VMail Message



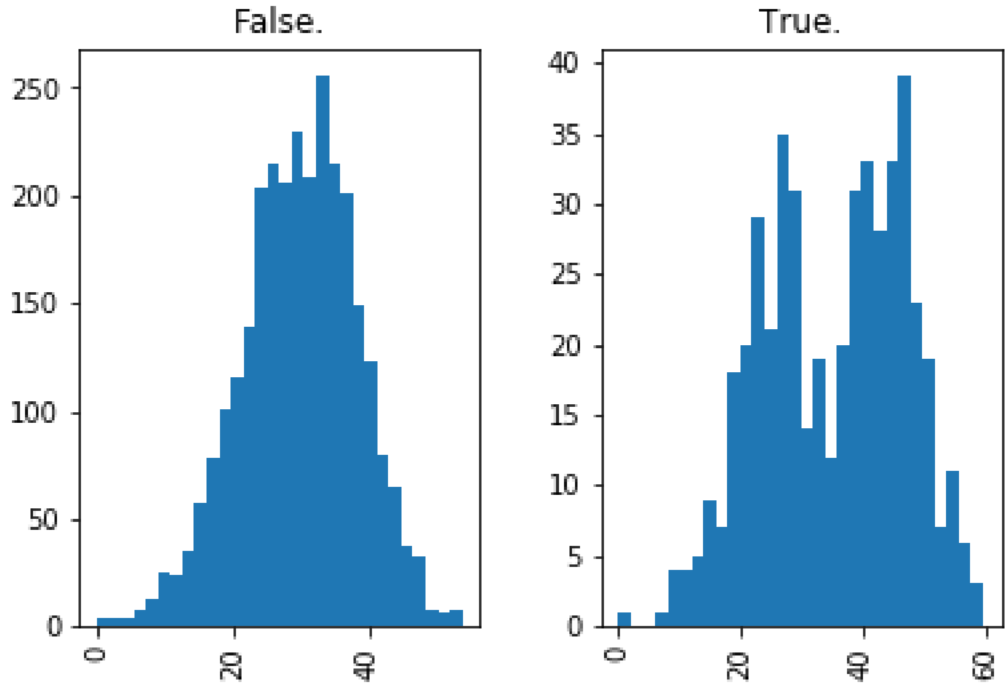
```
Day Mins
```



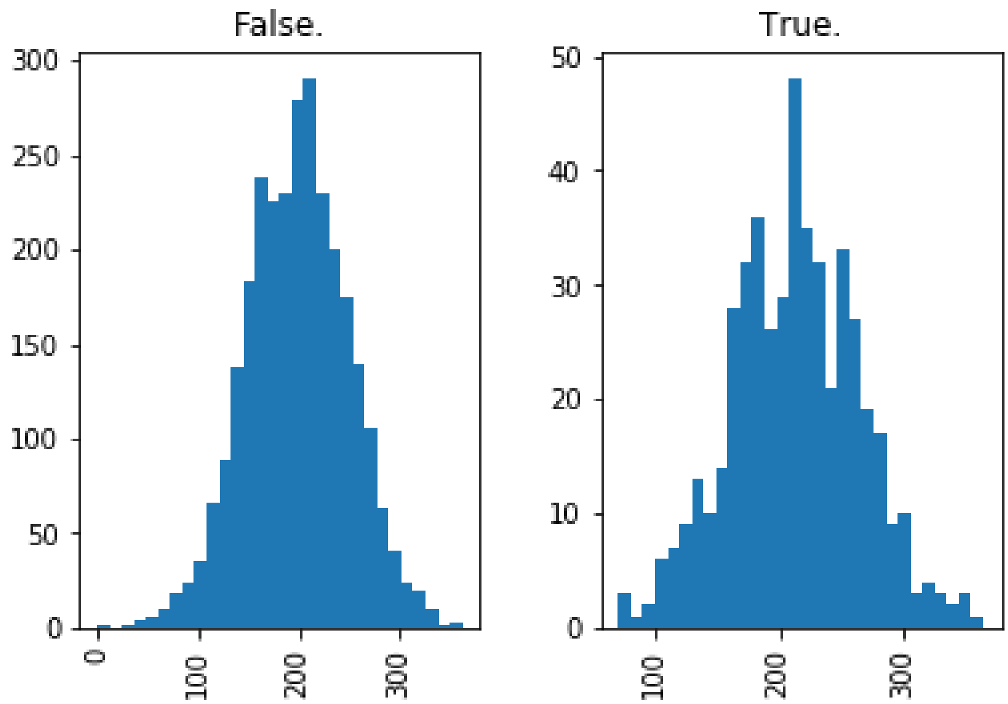

```
Day Calls
```



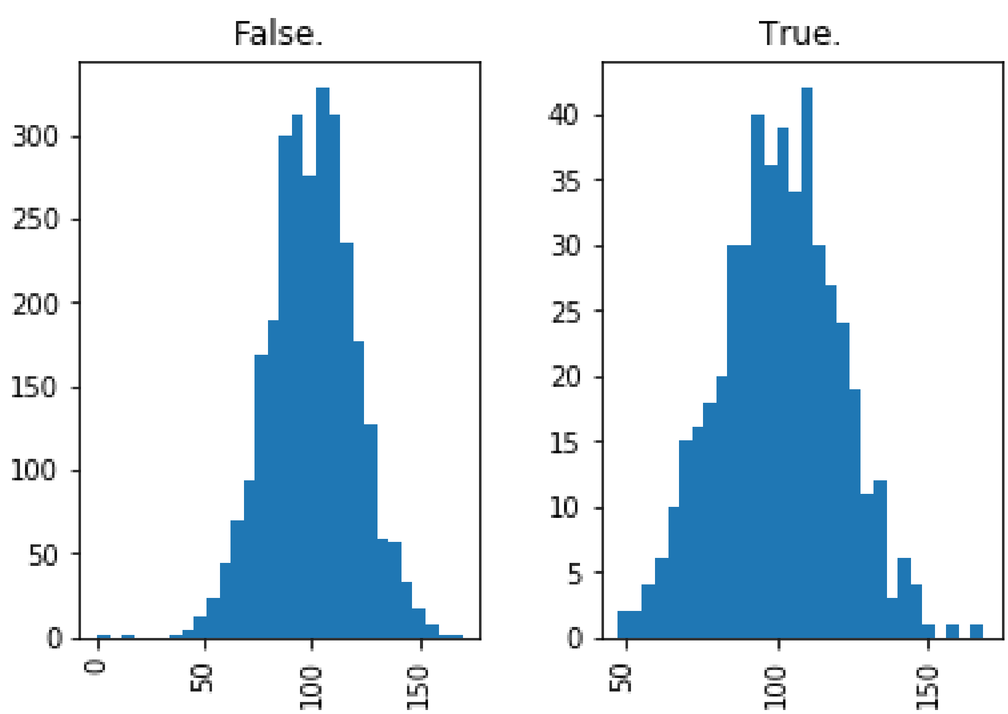
Day Charge



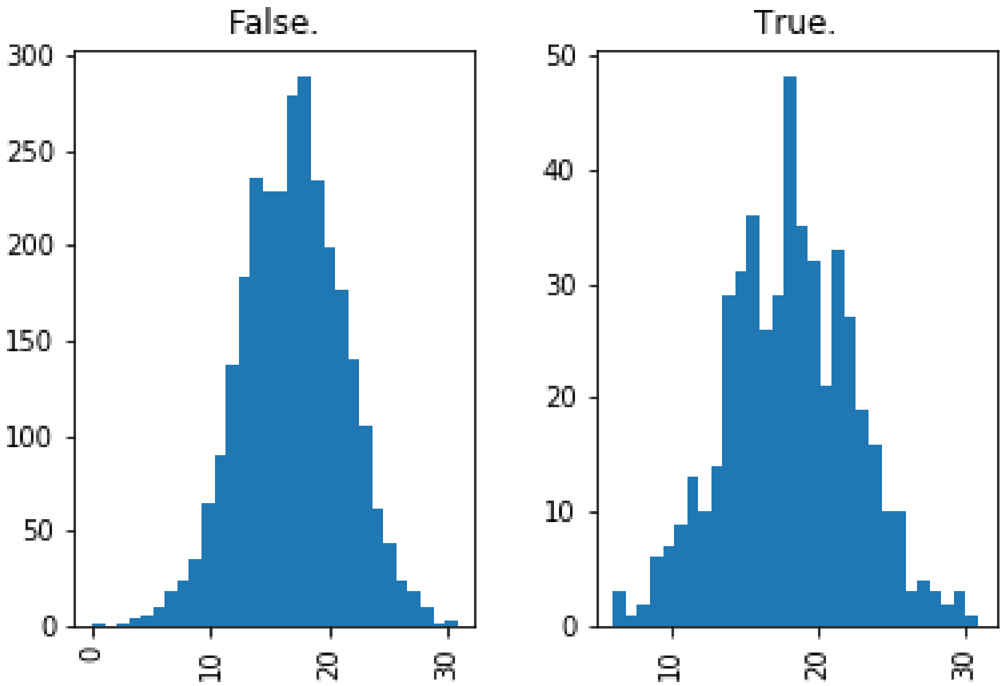
Eve Mins



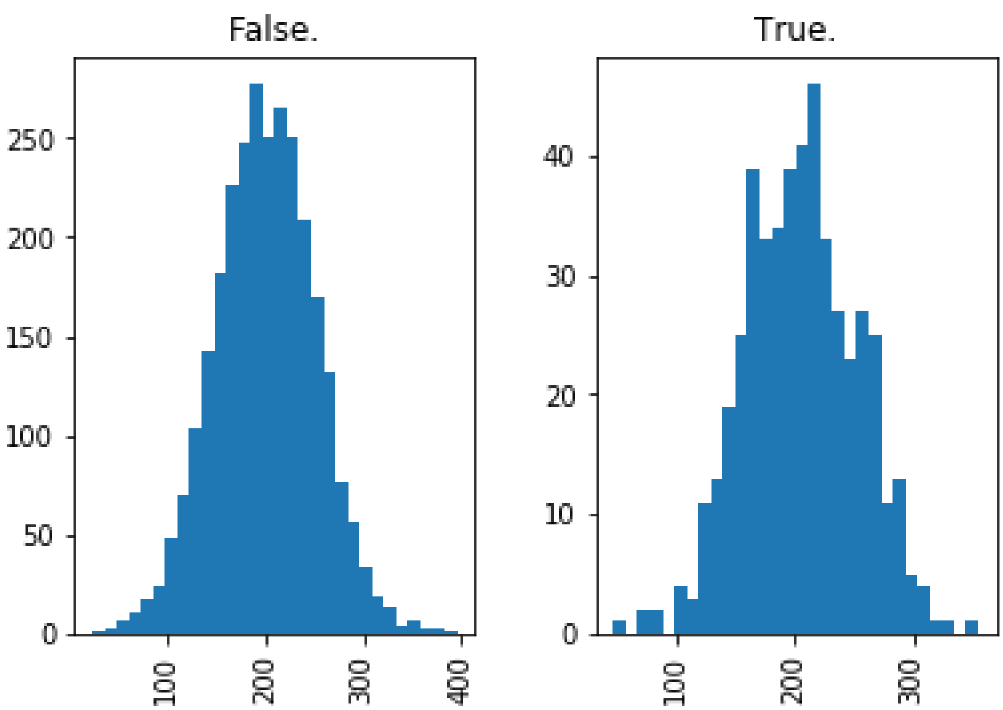
Eve Calls



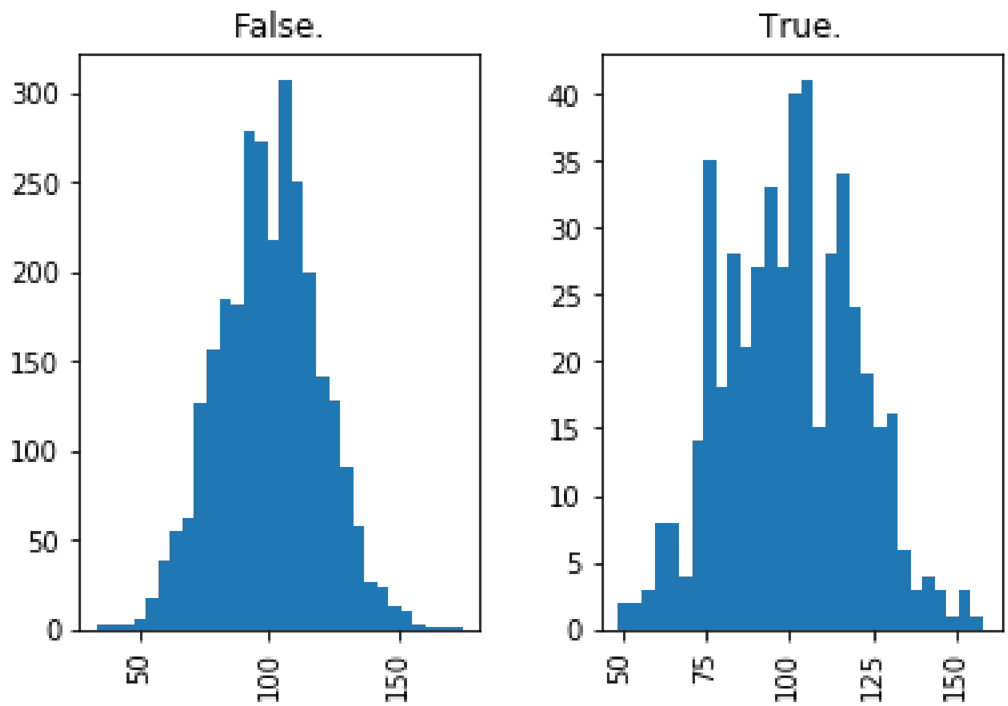
Eve Charge



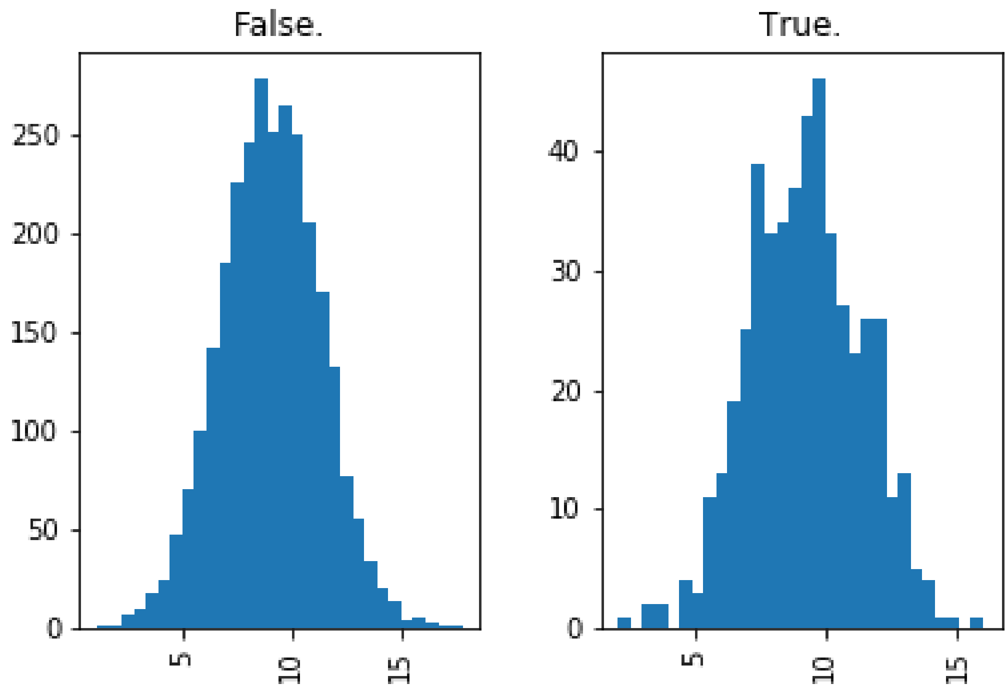
Night Mins



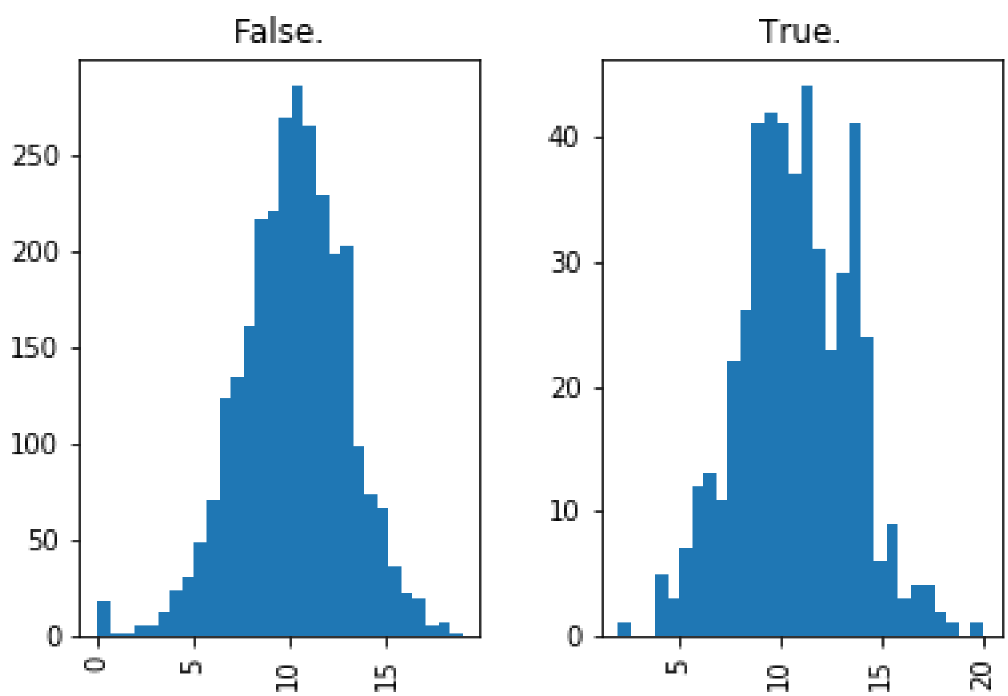
Night Calls



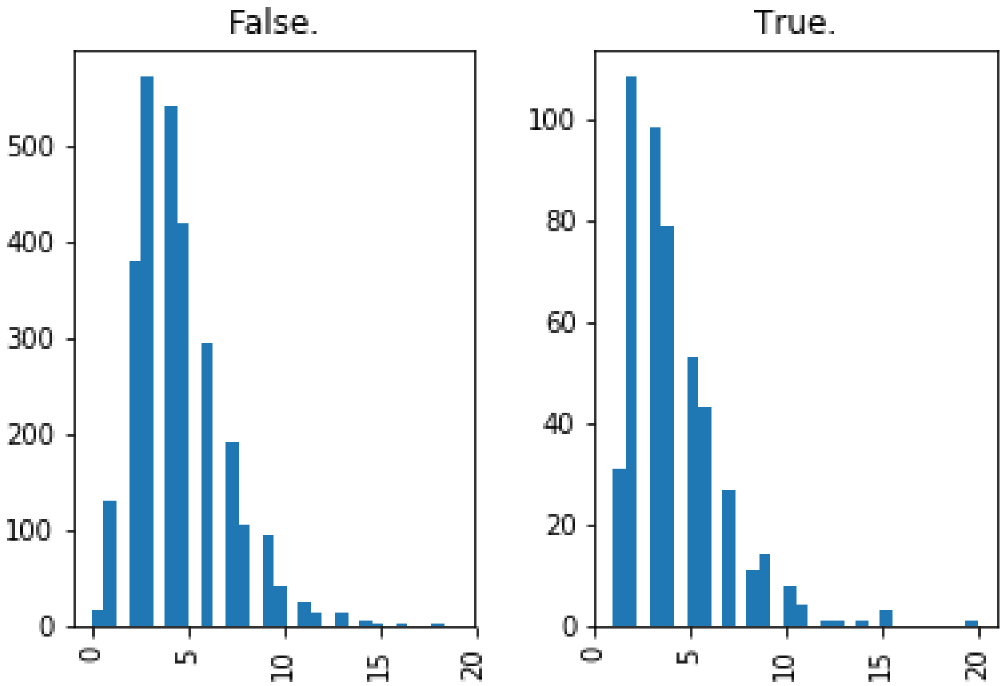
Night Charge


False.
True.
250
200
150
100
50
0
5
10
15
40
30
20
10
0
5
10
15

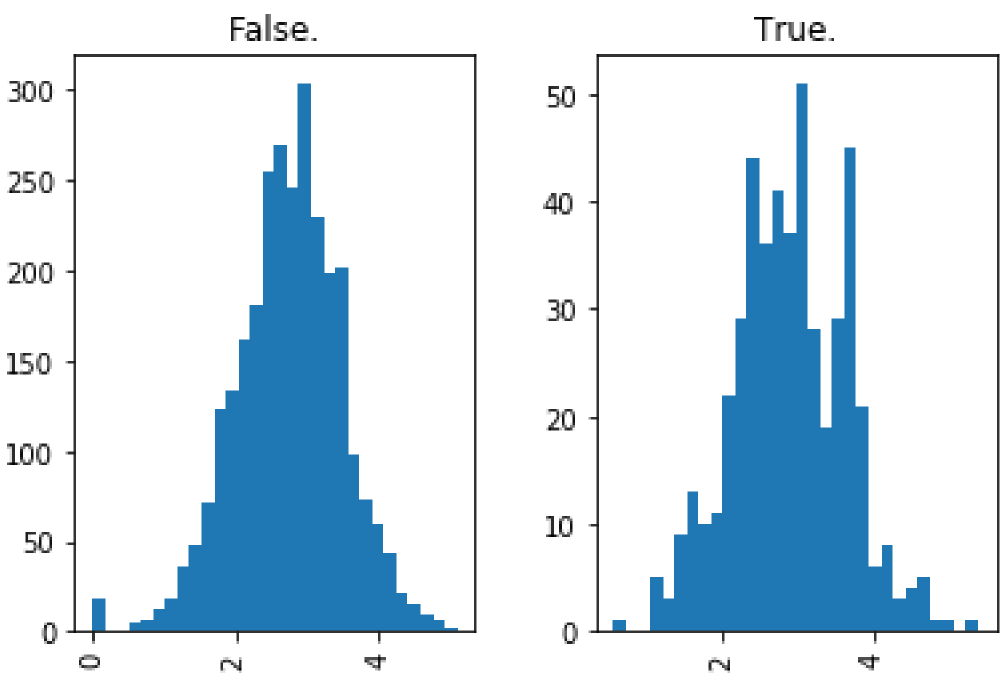
Intl Mins


False.
True.
250
200
150
100
50
0
0
5
10
15
40
30
20
10
0
5
10
15
20

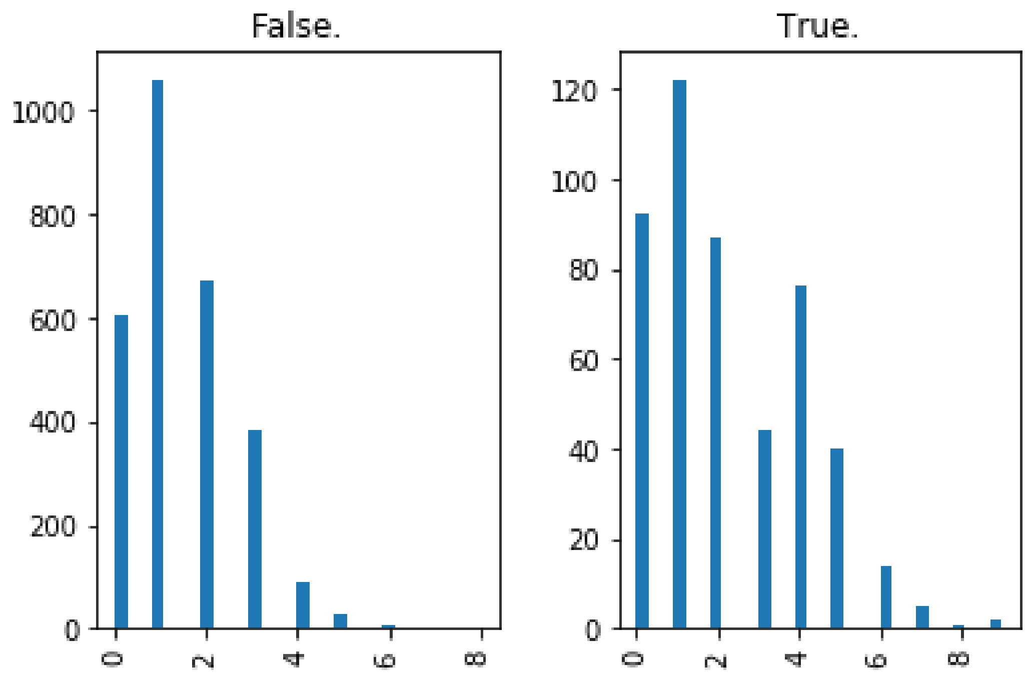
Intl Calls



Intl Charge



```
CustServ Calls
```



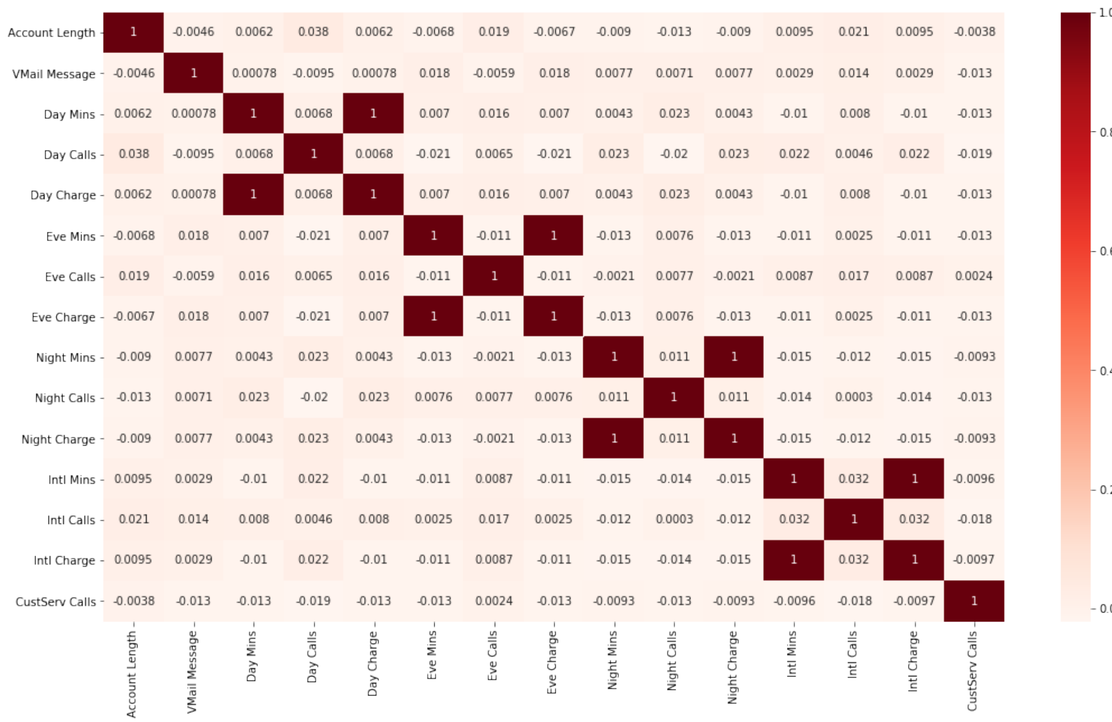
Interesting we see that churner appear: - Fairly evently distributed geopraphically - More likely to have an international plan - Less likely to the voicemail plan - To exhibit some bimodality in daily minutes

In addition, we see that churners take on very smiliar disributions for features like 'Day' , 'Mins' and 'Day Charge'. That's not surprising as we'd expect minutes spent talking to correlate with charges. Let's dig deeper into the relationships between our features

1.1.4 Lets draw heatmap to check multi collinearity

[9]:
```
# Correlation using Pearson correlation
plt.figure(figsize=(18,10))
cor = churn.corr()
sns.heatmap(cor, annot = True , cmap = plt.cm.Reds)
```

[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f25d2682358>
```



[9]:
```
#  Removal high correlation feature variables
churn = churn.drop(['Day Charge', 'Eve Charge' , 'Night Charge', 'Intl Charge'],␣
↪axis =1)
```

1.1.5 Step 5 — Organizing Data into Sets

[46]:
```
df_churn = churn.copy()
model_data = pd.get_dummies(df_churn)
model_data = pd.concat([model_data.drop(['Churn?_False.', 'Churn?_True.
↪'],axis=1), model_data['Churn?_True.']],axis=1)
```

2 Try to split feature and label

feature,label = model_data.iloc[:,:-1], model_data.iloc[:,[-1]]

[47]:
```
# split data into X and y
X = model_data.iloc[:,:-1]
y = model_data.iloc[:,-1]
```

Create arrays for the features and the target: X, y X, y = model_data.iloc[:,:-1], model_data.iloc[:,-1]

Now let's split the data into training, validation, and test sets. This will help prevent us from overfitting the model, and allow us to test the models accuracy on data it hasn't already seen.

[48]:
```
# Create train and test set

train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.25,␣
↪random_state= 42)

#eval_set = [(test_X, test_y)]
eval_set = [(train_X, train_y), (test_X, test_y)]
```

2.0.1 Step 6 — Model Building and Evaluating

It's now possible to identify the potential inactive customers that likely to churn and take measurable steps to retain then qucikly.

Before any modefication or tuning is made to the XGBoost algorithms, it's importnat to test default XGBoost model establish a baseline in performance.

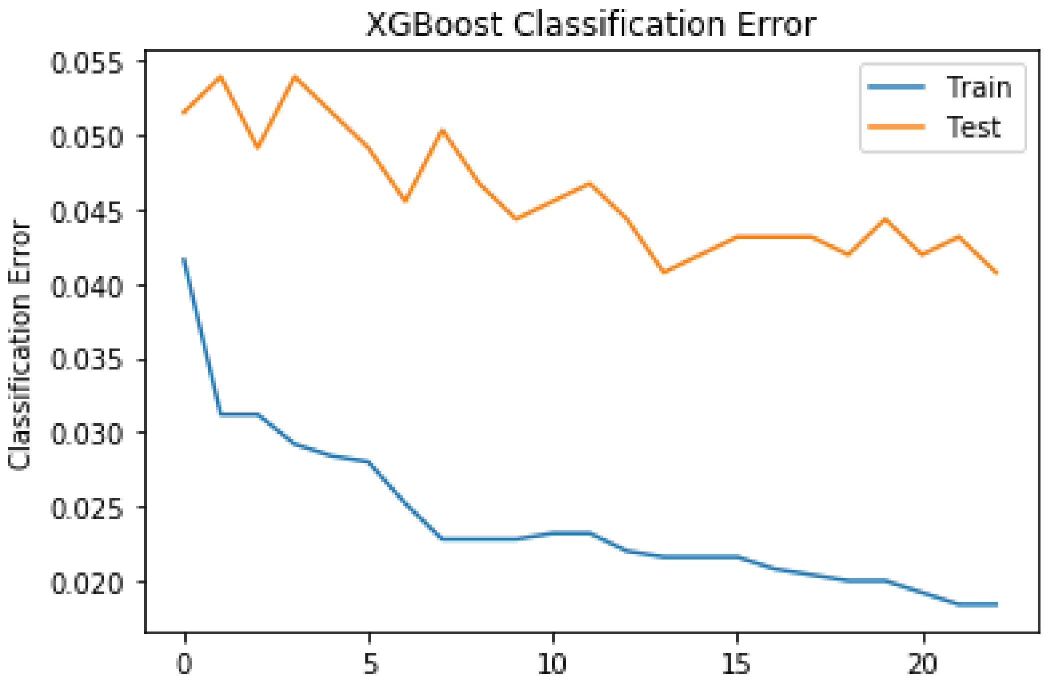
[13]:
```
# fit model no training data(default model)
model = XGBClassifier()
#eval_set = [(test_X, test_y)]
eval_set = [(train_X, train_y), (test_X, test_y)]
model.fit(train_X, train_y, eval_metric="error", early_stopping_rounds=10,␣
↪eval_set=eval_set, verbose=True)
```

```
[0]     validation_0-error:0.04162      validation_1-error:0.05156
Multiple eval metrics have been passed: 'validation_1-error' will be used for
early stopping.

Will train until validation_1-error hasn't improved in 10 rounds.
[1]     validation_0-error:0.03121      validation_1-error:0.05396
[2]     validation_0-error:0.03121      validation_1-error:0.04916
[3]     validation_0-error:0.02921      validation_1-error:0.05396
[4]     validation_0-error:0.02841      validation_1-error:0.05156
[5]     validation_0-error:0.02801      validation_1-error:0.04916
[6]     validation_0-error:0.02521      validation_1-error:0.04556
[7]     validation_0-error:0.02281      validation_1-error:0.05036
[8]     validation_0-error:0.02281      validation_1-error:0.04676
[9]     validation_0-error:0.02281      validation_1-error:0.04437
[10]    validation_0-error:0.02321      validation_1-error:0.04556
[11]    validation_0-error:0.02321      validation_1-error:0.04676
[12]    validation_0-error:0.02201      validation_1-error:0.04437
[13]    validation_0-error:0.02161      validation_1-error:0.04077
[14]    validation_0-error:0.02161      validation_1-error:0.04197
[15]    validation_0-error:0.02161      validation_1-error:0.04317
[16]    validation_0-error:0.02081      validation_1-error:0.04317
[17]    validation_0-error:0.02041      validation_1-error:0.04317
[18]    validation_0-error:0.02001      validation_1-error:0.04197
[19]    validation_0-error:0.02001      validation_1-error:0.04437
[20]    validation_0-error:0.01921      validation_1-error:0.04197
[21]    validation_0-error:0.01841      validation_1-error:0.04317
```

```
[22]	validation_0-error:0.01841	validation_1-error:0.04077
[23]	validation_0-error:0.01801	validation_1-error:0.04197
Stopping. Best iteration:
[13]	validation_0-error:0.02161	validation_1-error:0.04077
```

[13]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', 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=0, num_parallel_tree=1,
              objective='binary:logistic', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
              validate_parameters=1, verbosity=None)
```

[14]:
```python
from matplotlib import pyplot
# retrieve performance metrics
results = model.evals_result()
epochs = len(results['validation_0']['error'])
x_axis = range(0, epochs)
# plot classification error
fig, ax = pyplot.subplots()
ax.plot(x_axis, results['validation_0']['error'], label='Train')
ax.plot(x_axis, results['validation_1']['error'], label='Test')
ax.legend()
pyplot.ylabel('Classification Error')
pyplot.title('XGBoost Classification Error')
pyplot.show()
```



[84]:
```
# Make prediction
y_pred = model.predict(test_X)
```

2.0.2 Step 7 — Evaluating the Model's

Precision is a metric that quantifies the number of correct positive predictions made. Precision, therefore, calculate the accuracy for the minority class

[16]:
```
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
# Calculate prediction
precision = precision_score(test_y, y_pred, average = 'binary')
print('Precision: %.2f' % precision)

recall = recall_score(test_y, y_pred, average = 'binary')
print('Recall: %.2f' % recall)
```

```
Precision: 0.95
Recall: 0.77
```

[85]:
```
predictions = [round(value) for value in y_pred]
labels = np.unique(test_y)
# evaluate predictions
accuracy = accuracy_score(test_y, predictions)
print("Accuracy: %.0f%%" % (accuracy * 100.0))
```

```
pd.crosstab(test_y,y_pred , rownames=['Actual'], colnames=['Predicted'],␣
↪margins=True)
```

```
Accuracy: 96%
```

[85]:

```
Predicted    0    1  All
Actual
0          704    5  709
1           29   96  125
All        733  101  834
```

Of the 125 churners, we've correctly predicted 96 of them(true positives). And we incorrectly predicted 29 customers would churn who then end up not doing so (false negative). There are also 5 customers who end up churning, that we predicted would not(false postive).

Our model was pretty good! It was able to yield an accuracy score of almost 96%.

For the confusion matrix, the first element of the of the first row of the confusion matrix denotes the true negatives meaning the number of negative instances (False churn) predicted by the model correctly. And the last element of the second row of the confusion matrix denotes the true positives meaning the number of positive instances (True Churn) predicted by the model correctly.

2.0.3 So we will move forward with hyper paramenter tuning

Let's see if we can do better. We can perform a grid search of the model parameters to improve the model's ability to predict churn truly.

scikit-learn's implementation of xgboost classifier consists of different hyperparameters.

2.0.4 PreTune with default parameters

[19]:

```
xgb_pre_tune = XGBClassifier(max_depth= 3 , n_estimators= 100, min_child_weight=␣
↪1, reg_lambda= 1) # Default Paramters

# Ignore error
xgb_pre_tune.fit(train_X, train_y)
Y_pred = xgb_pre_tune.predict(test_X)

print ('F1 Accuracy : {}'.format(f1_score(test_y,Y_pred)))
print ('Precision Score: {}'.format(precision_score(test_y,Y_pred)))
print ('Recall Score: {}'.format(recall_score(test_y,Y_pred)))
```

```
F1 Accuracy : 0.8110599078341013
Precision Score: 0.9565217391304348
Recall Score: 0.704
```

2.0.5 Tuning

[20]:
```
clf_tune = XGBClassifier(n_estimators= 300, n_jobs= -1)
parameters = {'max_depth': range(5),
              'min_child_weight':[1,2,3,4,5],
              'reg_lambda': [0.50,0.75,1,1.25,1.5]}
```

[21]:
```
grid = GridSearchCV(clf_tune, param_grid = parameters, n_jobs= -1, cv = 5 )
```

[22]:
```
grid.fit(train_X, train_y)
```

[22]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=XGBClassifier(base_score=None, booster=None,
colsample_bylevel=None,
       colsample_bynode=None, colsample_bytree=None, gamma=None,
       gpu_id=None, importance_type='gain', interaction_constraints=None,
       learning_rate=None, max_delta_step=None, max_depth=None,
       min_child_w..._pos_weight=None, subsample=None,
       tree_method=None, validate_parameters=None, verbosity=None),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'max_depth': range(0, 5), 'min_child_weight': [1, 2, 3, 4,
5], 'reg_lambda': [0.5, 0.75, 1, 1.25, 1.5]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

[23]:
```
print('Best parameters: {}'.format(grid.best_params_))
```

```
Best parameters: {'max_depth': 4, 'min_child_weight': 1, 'reg_lambda': 1.5}
```

[24]:
```
# Ignore error
Y_pred_grid = grid.predict(test_X)
```

[25]:
```
f1_score(test_y, Y_pred_grid)
```

[25]:
```
0.8161434977578476
```

2.1 PostTune

[26]:
```
# Optimal Parameters
MAX_DEPTH = grid.best_params_['max_depth']
MIN_CHILD_WEIGHT = grid.best_params_['min_child_weight']
REG_LAMDBA = grid.best_params_['reg_lambda']
```

[27]:
```
clf_post_tune = XGBClassifier(max_depth= MAX_DEPTH , n_estimators= 500,
                              min_child_weight= MIN_CHILD_WEIGHT, reg_lambda=␣
  ↪REG_LAMDBA)
```

[29]:
```
# Ignore error
clf_post_tune.fit(train_X, train_y)
Y_pred = clf_post_tune.predict(test_X)
```

```
print ('Post_tune F1 Accuracy : {}'.format(f1_score(test_y,Y_pred)))
print ('Post_tune Precision Score: {}'.format(precision_score(test_y,Y_pred)))
print ('Post_tune Recall Score: {}'.format(recall_score(test_y,Y_pred)))
```

```
Post_tune F1 Accuracy : 0.8198198198198198
Post_tune Precision Score: 0.9381443298969072
Post_tune Recall Score: 0.728
```

2.1.1 Finding the best performing model

We have defined the grid of hyperparameter values and converted them into a single dictionary format which GridSearchCV() expects as one of its parameters. Now, we will begin the grid search to see which values perform best.

We will instantiate GridSearchCV() with our earlier xgboost model with all the data we have. Instead of passing train and test sets separately, we will supply X (scaled version) and y. We will also instruct GridSearchCV() to perform a cross-validation of five folds.

We'll end the notebook by storing the best-achieved score and the respective best parameters.

While building this customer churn predictor, we tackled some of the most widely-known preprocessing steps such as scaling, label encoding, and missing value imputation. We finished with some machine learning to predict if a person's application for a customer churn would get retain or not given some information about that person.

[87]:
```
pd.crosstab(test_y,Y_pred , rownames=['Actual'], colnames=['Predicted'],␣
↪margins=True)
```

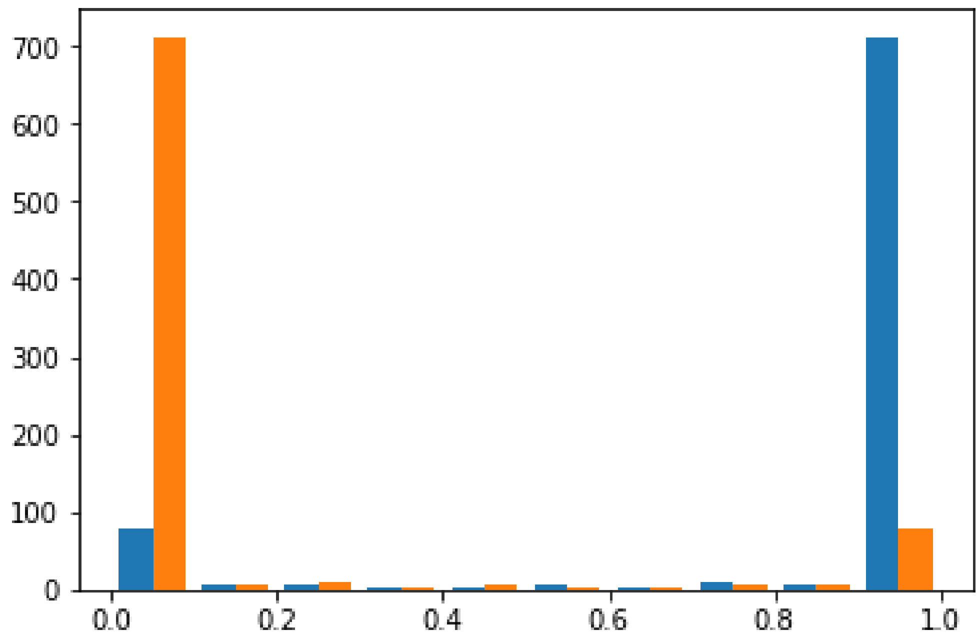
[87]:
```
Predicted    0   1  All
Actual
0          703   6  709
1           34  91  125
All        737  97  834
```

The continuous valued predictions coming from our best models tend to skew toward 0 or 1, but there is suffient mass between 0.1 and 0.9 that adjusting the cuttoff should indeed shift a number of customers' predictions.

For instances,

[53]:
```
# predict probabilities
yhat = clf_post_tune.predict_proba(test_X)
plt.hist(yhat)
plt.show
```

[53]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



[54]:
```
# keep probabilities for the positive outcome only
probs = yhat[:, 1]
```

[60]:
```
# define thresholds
thresholds = np.arange(0, 1, 0.001)
```

[70]:
```
# apply threshold to positive probabilities to create labels
def to_labels(pos_probs, threshold):
    return (pos_probs > threshold).astype('int')
```

[71]:
```
# evaluate each threshold
scores = [f1_score(test_y, to_labels(probs, t)) for t in thresholds]
```

[72]:
```
# get best threshold
ix = np.argmax(scores)
print('Threshold=%.3f, F-Score=%.3f' % (thresholds[ix], scores[ix]))
```

```
Threshold=0.254, F-Score=0.860
```

[86]:
```
# Print the confusion matrix
pd.crosstab(test_y,to_labels(probs,0.3) , rownames=['Actual'],␣
 ↪colnames=['Predicted'], margins=True)
```

[86]:

Predicted	0	1	All
Actual			
0	700	9	709
1	28	97	125
All	728	106	834

2.1.2 Step 8 — Conclusion

We can see that changing the cutoff from 0.5 to 0.25 results in 1 more true positives, 3 more false positive , and 6 fewer false negatives. The numbers are small overall, that's 6 -10% of customers overall that are shifting because of a change to the cuttoff.

Determing optimal cutoffs is a key step in properly applying machine learning in a real-word setting and also apply a specific problem like relative cost of error for our current situation for example False negative are the most problematic because they incorrectly predict that a churning customer will stay, we lose the customer and will have to pay all the cost of aquiring a replacement customer including adversting cost, adminstrivative cost.And then assign the cost of false negative. Let's assume $500

Finally for customers that our model identifies as churning, let's assume a retention incentive amount of dollar. Let's assume $100 . If the customer is happy but the model mistskenly predicted churn(False Positive) , we will waste the $100. We increased the loyality of aready loyal customer, so that's not so bad.

we will continue to work with feature selection algorthims or dimenstion reduction method for make to improve performance and efficency.

2.1.3 References:

- Jason Brownlee (2016, August 16).How to Evaluate Gradient Boosting Models with XGBoost in Python retrieve from https://machinelearningmastery.com/evaluate-gradient-boosting-models-xgboost-python/
- Jason Brownlee (2020, Feb10) A Gentle Introduction to Threshold-Moving for Imbalanced Classification retrieve from https://machinelearningmastery.com/threshold-moving-for-imbalanced-classification/