TELECOM CHURN - WHAT'S THE HANG UP?

1. Business Case

Our client, Syriatel, is concerned with their "churn" or customer loss. They work hard to provide great service at a reasonable price, and while losing customers is inevitabe, it's still painful.

But how painful? Syriatel is looking for a deep dive into their data to determine the scope of the problem and some steps to resolve it. Are there any patterns? Can we predict which customers could be leaving "soon"?

Syriatel's churn case is a binomial classification problem. This means, among all the data points we have, which determine whether a results will be true/false or 1/0. We can build a model to determine that.

We will create this model using an iterative process. First, we review and inspect the data, analyze it using 3 different modeling techniques, and then evaluate the results. From here, we re-evalute the data itself, and determine if there's any additional feature engineering or data cleaning. Then we would re-examine the results using our model of choice.

Instructions

The various sections of the code are outline below. Each Section is provided with a short summary and a link to more detailed description and source code below.

Road Map (to Perdition)

2. Data Understanding

For this project, we are utilizing data from SyriaTel, made available on Kaggle (https://www.kaggle.com/datasets/becksddf/churn-in-telecoms-dataset/), a popular data science hub. We are import the csv file into a DataFrame to do our work.

This is a binary classification problem. Fortunately our target column is called `churn` and already described to address this Churn, as a boolean target. In addition to that, we have 20 feature columns, 4 of which are objects. The rest are either int or floats.

This dataset appears to contain information about individual cellular accounts, with much of the information pertaining to day, night, evening, and international service usage and cost. There's also information on voicemail and the length of the account. There's only a state, area code, and phone number that identifies the account. There's a colun for account length, but we don't know if this account data is a total, a monthly, or yearly information.

Our Target variable distribution shows 483 churns out of 3333 entries. Our churn rate is approximately 15%, meaning we have some class imbalance we may need to address, and a relatively high accuracy bar to climb.

3. Data Preparation

In order to prep this data, we can first look at the quality of the data and realize a few things. First, we only have four categorical columns, the rest our numeric. Second, we have no null values. Amazing.

But we do have to convert these categorical columns. To do this, we're going to apply a little domain knowledge here. For starters, we know that phone numbers are assigned at random, and these should have no bearings on churn. So we've dropped the phone numbers from our dataset.

Next, we know that geography could have an effect on churn, so we'll go ahead and one hot encode the state and the area code. We also have voice mail and international plan as yes/no column, so we'll convert those columns to (1/0).

Finally, we will convert the target `churn` column from boolean to an integer (0/1).

4. Modeling

Our approach to modeling is to try 3 different modeling techniques to see which one yields the best early results. We utilized KNN, Logistic Regression, and Decision Tree. For each model, we got

As part of the modeling process, we perform a train-test split on the data. We'll set a random seed and the standard sample size of .25 for the test. We're going to do this prior to any scaling or other transformation to avoid data leakage.

We're also going to use three different modeling techniques to start and then build from the best performing one. We will tune the hyperparameters to ensure that we're getting optimized results for each model

We will also use the four categories of precision, recall, accuracy, and F1 score to test how well are model works.

Our approach to modeling is to try 3 different modeling techniques to see which one yields the best early results. We utilized KNN, Logistic Regression, and Decision Tree.

In each case, we trained the model on the same set of training data using default hyperparamters settings. The test results were evaluated, and the model rerun optimizing for the hyperparameters. The results were evaluated on train data with a cross-validation, to guard against overfit.

The model with the best results was the Decision Tree model, with the results of the test data shown below. This result included hypertuning and cross-validations sampling.

```
 Precision Score: 0.8969072164948454
Recall Score: 0.696
```

5. Evaluation

Now we have done some baseline modeling, we seemed to have settled on a Decision Tree approach which gives us some of best preliminary results. It's a good time to see what we have an tweak the model from there.

With decision tree, we can review a features importance plot. This tell us that the state and area code information has no importance. It tells us that the most important feature was the 'Total Day Charge'. The next most important was Customer Service Calls.

Thinking about these factors make sense intuitively, those people who talk on the phone a lot may have higher bills, and the most likely to quit. But... we don't have a feature for total charge. We have different categories of evening, daytime, and international charge, but no total. Let's verify that there is a flat charge for the different call rates.

Sampling - We're going to utilize the SMOTE technique on our data and train on models on that. This should increase our accuracy and limit and issues related to class imbalance.

6. Iteration

So, we removed the area code and state from our data and also confirmed the flat rates for our calls. We were able to consolidate all minute charges into one 'total charge category'.

From here we ran additional Decision Tree models with CV grid optimization and creaded our best pefroming model.

Our best performing model was a Decision Tree model with new, Feature Engineered Data.

DT Trial 1 - (train with hypertuning and cross-validation

```
 Precision Score: 0.990506329113924
Recall Score: 0.8505434782608695
Accuracy Score: 0.9767907162865146
F1 Score: 0.9152046783625729
Mean Cross Validation Score: 97.64%
```

Whoa! Precision of 0.99, and accuracy of 97.6, with cross-validation of 97.6. These are amazing. Let's try it on the test-data.

DT Trial 2 - (test with hypertuning)

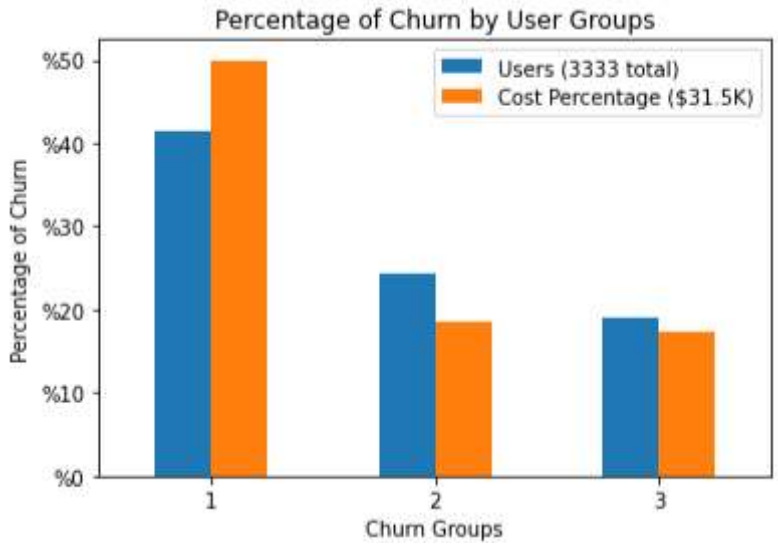
```
 Precision Score: 0.9805825242718447
Recall Score: 0.8782608695652174
Accuracy Score: 0.9808153477218226
F1 Score: 0.926605504587156
```

A precision and accuracy score of 98% on our test data is strong. And this is comparable to our train and cross-validation data.

We tried additional SMOTE resampling, and other boosting models but the above was our ebst result.

7. Conclusion

From our model we can conclude 3 major factors that drive Churn. We divided them into issue buckets.



- Bucket 1 - Price - churn users who pay more than 74 USD a month and have no voicemail.
- Bucket 2 - Customer Service - churn users who pay less than 60 USD and have more than 3 customer service calls.

- Bucket 3 - International Service - churn users less than 74 USD who have international

Data Understanding

For this project, we are utilizing data from SyriaTel, made available on Kaggle (https://www.kaggle.com/datasets/becksddf/churn-in-telecoms-dataset/), a popular data science hub.

To get started, we're going to import our standard collection of relevant items.

In [2]:
```
#import python libraries for data visualizations, dataframes, and statisti
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import itertools

#import scikit modules for machine learning techniques
from sklearn.model_selection import train_test_split, GridSearchCV, cross_
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifie
from sklearn.metrics import recall_score, precision_score, accuracy_score,
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import OneHotEncoder, StandardScaler, MinMaxSca

from imblearn.over_sampling import SMOTE, ADASYN
```

Obtaining Data

The data itself is a CSV file, so we can use `read_csv` to import it to Pandas.

In [3]:
```
df = pd.read_csv('data/bigml_59c28831336c6604c800002a.csv')
df.head()
```

Out[3]:

	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	...	tota ev call
0	KS	128	415	382-4657	no	yes	25	265.1	110	45.07	...	9
1	OH	107	415	371-7191	no	yes	26	161.6	123	27.47	...	10
2	NJ	137	415	358-1921	no	no	0	243.4	114	41.38	...	11
3	OH	84	408	375-9999	yes	no	0	299.4	71	50.90	...	8
4	OK	75	415	330-6626	yes	no	0	166.7	113	28.34	...	12

5 rows × 21 columns

Data Inspection

Let's do some high level inspection. We'll look at target variable distribution, feature types, and some of the overall statistical characteristics of our numeric data.

Target Variable Distribution

How many customers are "churning". True means they've left. False means they're still a customer.

In [4]:
```
#obtain total count of target variables
df['churn'].value_counts()
```

Out[4]:
```
False    2850
True      483
Name: churn, dtype: int64
```

In [5]:
```
#assign different
target = df['churn']
features = df.drop('churn', axis = 1)
```

Features Type

What are our features type? We'll need to convert these to numerical at some point

In [6]:
```
#find the type counts for your data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 21 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   state                   3333 non-null   object
 1   account length          3333 non-null   int64
 2   area code               3333 non-null   int64
 3   phone number            3333 non-null   object
 4   international plan      3333 non-null   object
 5   voice mail plan         3333 non-null   object
 6   number vmail messages   3333 non-null   int64
 7   total day minutes       3333 non-null   float64
 8   total day calls         3333 non-null   int64
 9   total day charge        3333 non-null   float64
 10  total eve minutes       3333 non-null   float64
 11  total eve calls         3333 non-null   int64
 12  total eve charge        3333 non-null   float64
 13  total night minutes     3333 non-null   float64
 14  total night calls       3333 non-null   int64
 15  total night charge      3333 non-null   float64
 16  total intl minutes      3333 non-null   float64
 17  total intl calls        3333 non-null   int64
 18  total intl charge       3333 non-null   float64
 19  customer service calls  3333 non-null   int64
 20  churn                   3333 non-null   bool
dtypes: bool(1), float64(8), int64(8), object(4)
memory usage: 524.2+ KB
```

Numeric Feature Statistics

In [11]:
```
#let's see some high level description of the first 10 columns
df.loc[:,'account length':'total night minutes'].describe()
```

Out[11]:

	account length	area code	number vmail messages	total day minutes	total day calls	total day charge	total ev minute
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.00000
mean	101.064806	437.182418	8.099010	179.775098	100.435644	30.562307	200.98034
std	39.822106	42.371290	13.688365	54.467389	20.069084	9.259435	50.71384
min	1.000000	408.000000	0.000000	0.000000	0.000000	0.000000	0.00000
25%	74.000000	408.000000	0.000000	143.700000	87.000000	24.430000	166.60000
50%	101.000000	415.000000	0.000000	179.400000	101.000000	30.500000	201.40000
75%	127.000000	510.000000	20.000000	216.400000	114.000000	36.790000	235.30000
max	243.000000	510.000000	51.000000	350.800000	165.000000	59.640000	363.70000

In [12]:
```
#let's see some high level description of the first 10 columns
df.loc[:,'total night minutes':'churn'].describe()
```

Out[12]:

	total night minutes	total night calls	total night charge	total intl minutes	total intl calls	total intl charge	custome servic call
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.00000
mean	200.872037	100.107711	9.039325	10.237294	4.479448	2.764581	1.56285
std	50.573847	19.568609	2.275873	2.791840	2.461214	0.753773	1.31549
min	23.200000	33.000000	1.040000	0.000000	0.000000	0.000000	0.00000
25%	167.000000	87.000000	7.520000	8.500000	3.000000	2.300000	1.00000
50%	201.200000	100.000000	9.050000	10.300000	4.000000	2.780000	1.00000
75%	235.300000	113.000000	10.590000	12.100000	6.000000	3.270000	2.00000
max	395.000000	175.000000	17.770000	20.000000	20.000000	5.400000	9.00000

Summary

A few high level summary items. We have: 1 - boolean target (binary classification) 20 - feature columns 3333 - total entries 4 -

There are 3333 total entries. Of our feature columns, 4 are objects. Of these 4 columns, they all feel relevant for our case, except for phone number. The other columns are floats or ints so we can process that appropriately. Additionally, we see that `'area code'` is an `int`. We know that area codes are really a description of a region, much like the `'state'` column, so we should probably treat that like a categorical. Our review of the states column shows that we have siginificant accounts from every state.

Our churn rate is shown below

In [8]:
```
def print_churn (target):

    print(f'There a total of {len(target)} entries.')
    class_imbalance = " "
    churn_percentage = len(target[target == True])/len(target)
    if churn_percentage > .2:
        class_imbalance = " not "
    print(f'The percentage of users who churn is {churn_percentage :.2%}.'
    print(f'We do{class_imbalance}have class imbalance concerns.')
    print(f'So... our model should have a higher accuracy than {(1 - churn_

print_churn (target)
```

```
There a total of 3333 entries.
The percentage of users who churn is 14.49%.
We do have class imbalance concerns.
So... our model should have a higher accuracy than 85.51% at predicting who w
ill stay.
```

In [9]:

```
#let's create a baseline evaluation metric.
from sklearn.metrics import precision_score, recall_score, accuracy_score,

labels = target.astype(int)
preds = np.zeros(len(labels),dtype = int)

def print_metrics(labels, preds):
    print("Precision Score: {}".format(precision_score(labels, preds)))
    print("Recall Score: {}".format(recall_score(labels, preds)))
    print("Accuracy Score: {}".format(accuracy_score(labels, preds)))
    print("F1 Score: {}".format(f1_score(labels, preds)))

print_metrics(labels, preds)
```

```
Precision Score: 0.0
Recall Score: 0.0
Accuracy Score: 0.8550855085508551
F1 Score: 0.0

C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\metrics\_cl
assification.py:1221: UndefinedMetricWarning: Precision is ill-defined and be
ing set to 0.0 due to no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Data Preparation

Before we begin the modeling, we have to make sure our data is cleaned and ready for processing. So, to do this, we're going to drop the phone numbers, one hot encode our categorical data, and convert our boolean category to integers (0,1). We previously observed that we have no null values, so we don't have to fill any in.

Drop Phone Numbers

From our domain knowledge, we know that the last seven digits of a phone number are unique and assigned randomly. We don't want these numbers considered so we're going to drop them from our data set.

In [10]:
```
X = features.drop('phone number', axis = 1)
X
```

Out[10]:

	state	account length	area code	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	tota ev call
0	KS	128	415	no	yes	25	265.1	110	45.07	197.4	9
1	OH	107	415	no	yes	26	161.6	123	27.47	195.5	10
2	NJ	137	415	no	no	0	243.4	114	41.38	121.2	11
3	OH	84	408	yes	no	0	299.4	71	50.90	61.9	8
4	OK	75	415	yes	no	0	166.7	113	28.34	148.3	12
...	...	...	...	...	...	...	...	...	...	...	.
3328	AZ	192	415	no	yes	36	156.2	77	26.55	215.5	12
3329	WV	68	415	no	no	0	231.1	57	39.29	153.4	5
3330	RI	28	510	no	no	0	180.8	109	30.74	288.8	5
3331	CT	184	510	yes	no	0	213.8	105	36.35	159.6	8
3332	TN	74	415	no	yes	25	234.4	113	39.85	265.9	8

3333 rows × 19 columns

One Hot Encoding

Now that we've dropped our phone number category, it's we can one hot encode our two remaning columns, which are `'state'`, `'area code'`, as well as `'international plan'` and `'voice mail plan'`. We can also convert our target column to 1/0 as well. To simplify, we'll do a binary converstion on `'international plan'` and `'voice mail plan'` to convert to 1/0 .

In [11]:
```
#we'll use a mapping function to convert the yes/nos to 1/0s
d = {'no': 0, 'yes': 1}
X['international plan'] = X['international plan'].map(d).fillna(X['voice m
X['voice mail plan'] = X['voice mail plan'].map(d).fillna(X['voice mail pl
```

In [12]:
```
#we'll use getdummies function to do one hot encoding for the 'state' and
X_ohe = pd.get_dummies(X)
X_ohe = pd.get_dummies(X_ohe, columns = ['area code'])
#we'll remerge these matrices.
#X_ohe = X.drop(['state','area code'],axis=1)
X_ohe
```

0	128	0	1	25	265.1	110	45.07	197.4	99	16.78
1	107	0	1	26	161.6	123	27.47	195.5	103	16.62
2	137	0	0	0	243.4	114	41.38	121.2	110	10.30
3	84	1	0	0	299.4	71	50.90	61.9	88	5.26
4	75	1	0	0	166.7	113	28.34	148.3	122	12.61
...	...	...	...	...	...	...	...	...	...	...
3328	192	0	1	36	156.2	77	26.55	215.5	126	18.32
3329	68	0	0	0	231.1	57	39.29	153.4	55	13.04
3330	28	0	0	0	180.8	109	30.74	288.8	58	24.55
3331	184	1	0	0	213.8	105	36.35	159.6	84	13.57
3332	74	0	1	25	234.4	113	39.85	265.9	82	22.60

3333 rows × 71 columns

Great, so we've expanded our now let's convert the target from yes/no to 1/0.

In [13]:
```
#we'll use the .astype function to convert boolean to the integer
target = target.astype(int)
```

In [14]:
```
target
```

Out[14]:
```
0       0
1       0
2       0
3       0
4       0
       ..
3328    0
3329    0
3330    0
3331    0
3332    0
Name: churn, Length: 3333, dtype: int32
```

Summary

We've now performed the one hot encoding for our categorical data, as well as removed the phone numbers from our data. We've also converted our target to 0,1s, so we should be ready to perform our train-test split, transform the data, and begin training our model.

Modeling

Now that we have cleaned our data, we can being to prepare it for preprocessing, transformation, and eventually modeling. We noted early that we have roughly 85% of our users who stay on our cellular plan, so we should account for the class imbalance here.

Train-Test Split and Scaling

To start, let's perform our train-test split and scaling.

In [15]:
```
#perform train-test split
X_train, X_test, y_train, y_test = train_test_split(X_ohe, target, random_

# Instantiate StandardScaler
scaler = StandardScaler()

# Transform the training and test sets
scaled_data_train = scaler.fit_transform(X_train)
scaled_data_test = scaler.transform(X_test)
```

Perfect, now, to inspect the split I'll convert into a dataframe.

In [16]:
```
# Convert into a DataFrame
scaled_df_train = pd.DataFrame(scaled_data_train, columns = X_ohe.columns)
scaled_df_train
```

Out[16]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	t
0	-1.404508	-0.327448	-0.611418	-0.584700	-1.883677	1.330852	-1.884170	1.037727	0
1	0.366388	-0.327448	-0.611418	-0.584700	0.294083	0.529165	0.293703	0.516178	0
2	0.518179	-0.327448	1.635543	1.685101	1.056392	-1.875896	1.056666	0.093407	0
3	2.010792	-0.327448	-0.611418	-0.584700	-0.679156	1.681590	-0.679320	-0.402459	0
4	0.290493	-0.327448	-0.611418	-0.584700	0.484660	1.080325	0.484172	-0.718549	-0
...	...	...	...	...	...	...	...	...	
2494	0.138701	-0.327448	-0.611418	-0.584700	1.746540	0.980114	1.746707	-0.044882	-0
2495	0.543478	-0.327448	-0.611418	-0.584700	-2.681141	-1.926002	-2.680873	-0.396533	-0
2496	-0.873239	-0.327448	-0.611418	-0.584700	-1.709753	-1.224526	-1.710027	1.207625	0
2497	1.732508	-0.327448	-0.611418	-0.584700	-0.014911	0.529165	-0.015400	-0.507164	1
2498	-1.632195	-0.327448	1.635543	2.563733	-2.777355	1.130430	-2.777740	-1.417899	0

2499 rows × 71 columns

Perfect, so, we have our scaled data set now. We can begin a various modeling techniques. We will compare various techniques, KNN, Logisitic Regression and Decision Tree and then make a decision from those base models on which to proceed.

KNN

We'll set up a baseline KNN classifier with no pyrameters tuning to see what we get.

In [17]:
```
# Import KNeighborsClassifier
from sklearn.neighbors import KNeighborsClassifier

# Instantiate KNeighborsClassifier
knn_clf = KNeighborsClassifier()

# Fit the classifier
knn_clf.fit(scaled_data_train, y_train)

# Predict on the test set
train_preds = knn_clf.predict(scaled_data_train)
```

Now we have our model, let's see the results. We're going to use a variety of metrics here.

In [18]:
```
# import the evaluation metrics
from sklearn.metrics import precision_score, recall_score, accuracy_score,

# create a function to evaluate the results
def print_metrics(labels, preds):
    print("Precision Score: {}".format(precision_score(labels, preds)))
    print("Recall Score: {}".format(recall_score(labels, preds)))
    print("Accuracy Score: {}".format(accuracy_score(labels, preds)))
    print("F1 Score: {}".format(f1_score(labels, preds)))

print_metrics(y_train, train_preds)
```

```
Precision Score: 0.9
Recall Score: 0.17597765363128492
Accuracy Score: 0.8791516606642658
F1 Score: 0.29439252336448596
```

Precision score of 0.9 isn't bad, and the accuracy score has improved. We used the default K in this case of 5. Let's see how it looks on the test data

In [19]:
```
#predict on test data
test_preds = knn_clf.predict(scaled_data_test)

#evaluate the results
print_metrics(y_test, test_preds)
```

```
Precision Score: 0.5333333333333333
Recall Score: 0.064
Accuracy Score: 0.8513189448441247
F1 Score: 0.11428571428571431
```

Okay, so not as good. It looks like we got very lucky on our train data and overfit quite a bit. Let's see if we can find an optimal K-value that might improve things. We can run a loop to find it.

In [20]:
```
#define a function to find the optimal K-value
def find_k(X_train, y_train, X_test, y_test, min_k=1, max_k=25):
    best_k = 0
    best_score = 0.0
    #iterate through the K-range previously specified and run a model each
    for k in range(min_k, max_k+1, 2):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        preds = knn.predict(X_test)
        #retrieve accuracy score and assign to best_score if the highest
        acc = accuracy_score(y_test, preds)
        if acc > best_score:
            best_k = k
            best_score = acc

    print("Best Value for k: {}".format(best_k))
    print("Accuracy-Score: {}".format(best_score))

#let's call the function
find_k(scaled_data_train, y_train, scaled_data_train, y_train)
```

```
Best Value for k: 1
Accuracy-Score: 1.0
```

Whoa! These results seem... off. It saying our best value for K is to test 1 neighbor and our Accuracy score is perfect.

Maybe we got lucky with our split. Let's rerun this optimzation loop but this time we will score with Cross validation.

In [21]:
```
#define a function to find the optimal K-value with cross_validation
def find_k_cross(X_train, y_train, X_test, y_test, min_k=1, max_k=25):
    best_k = 0
    best_score = 0.0
    for k in range(min_k, max_k+1, 2):
        knn_clf_cross = KNeighborsClassifier(n_neighbors=k)
        dt_cv_score = cross_val_score(knn_clf_cross, X_train, y_train, cv=
        acc = np.mean(dt_cv_score)
        if acc > best_score:
            best_k = k
            best_score = acc

    print("Best Value for k: {}".format(best_k))
    print("Accuracy-Score: {}".format(best_score))

find_k_cross(scaled_data_train, y_train, scaled_data_train, y_train)
```

```
Best Value for k: 7
Accuracy-Score: 0.8591430861723446
```

Okay... this look more reasonable. Our optimal K value was 7 and our Accuracy was 85.9%. So, let's see how this performs on our test.

In [22]:
```
knn_clf = KNeighborsClassifier(n_neighbors=7)
knn_clf.fit(X_train, y_train)
preds = knn_clf.predict(X_test)

print_metrics(y_test, preds)
```

```
Precision Score: 0.7857142857142857
Recall Score: 0.264
Accuracy Score: 0.8788968824940048
F1 Score: 0.3952095808383233
```

This is an improvement from our baseline with a 78% precision score. However our F1 and Recall our quite low. It seems as though we still have trouble with False Negatives, meaning we need to identify more churn than we have. Let's move on to a different model and see if anything changes.

Logistic Regression

Now let's try a logistic regression. To perform this, we're going to scale our model differently. Previously, we used a standard scaler with KNN. However, now with the Logistic Regression, we must used the MinMaxScaler to avoid negative numbers.

Let's go ahead and transform the data and then run our model.

In [23]:
```
# Instantiate StandardScaler
mmscaler = MinMaxScaler()

# Transform the training and test sets
mmscaled_data_train = mmscaler.fit_transform(X_train)
mmscaled_data_test = mmscaler.transform(X_test)
```

In [24]:
```
# Instantiate the Logistic Regression Model
from sklearn.linear_model import LogisticRegression

logireg = LogisticRegression(fit_intercept=False, C=1e12, solver='liblinea
model_log = logireg.fit(mmscaled_data_train, y_train)
model_log
```

Out[24]:
```
LogisticRegression(C=1000000000000.0, fit_intercept=False, solver='liblinea
r')
```

In [25]:
```
train_preds = model_log.predict(mmscaled_data_train)
print_metrics(y_train, train_preds)
```

```
Precision Score: 0.6333333333333333
Recall Score: 0.26536312849162014
Accuracy Score: 0.8727490996398559
F1 Score: 0.37401574803149606
```

Okay, so... at quick glance we did improve slightly on the Accuracy Score. Again, it's nothing... amazing. Perhaps we can iterate over a few options and find a good C, as well as a solver.

This time, instead of a custom optimizer, I'm going to use the GridSearchCV option.

In [26]:
```
logi_grid_clf = LogisticRegression(fit_intercept=False)

param_grid = {
    'C': [1e1, 1e5, 1e7, 1e10, 1e12, 1e15, 1e20],
    'solver': ['lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga'],
    'max_iter': [10, 50, 100, 250, 500]
}

gs_logi = GridSearchCV(logi_grid_clf, param_grid, cv=5)
gs_logi.fit(mmscaled_data_train, y_train)

gs_logi.best_params_
```

```
C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\linear_mo
del\_logistic.py:762: ConvergenceWarning: lbfgs failed to converge (status=
1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://sci
kit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regr
ession (https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression)
  n_iter_i = _check_optimize_result(
C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\linear_mo
del\_logistic.py:762: ConvergenceWarning: lbfgs failed to converge (status=
1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://sci
```

Okay, so we hypyertuned some parameters. I'm going to run the logistic regression, again with the new numbers.

In [27]:
```
#instantiate an updated logistic regression with tuned hyperparameters
logireg = LogisticRegression(fit_intercept=False, C=1e10, solver='saga', m
model_log = logireg.fit(mmscaled_data_train, y_train)

#see how model performs on training data
train_preds = model_log.predict(mmscaled_data_train)
print_metrics(y_train, train_preds)
```

```
Precision Score: 0.6546762589928058
Recall Score: 0.2541899441340782
Accuracy Score: 0.8739495798319328
F1 Score: 0.36619718309859156

C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\linear_mode
l\_sag.py:329: ConvergenceWarning: The max_iter was reached which means the c
oef_ did not converge
  warnings.warn("The max_iter was reached which means "
```

Right, so only a marginal increase in accuracy and F1 score. Our precision score actually dropped, so this model doesn't feel appropriate for this data. Let's move to a decision tree.

Decision Tree

With decision tree, we're going to try without scaling and see what happens. We have our original training data and we're going to add a cross validation score at the end to make sure our accuracy is okay, and to account for any overfitting that would naturally occur with a Decision tree.

In [28]:
```
#instantiate the tree
dt_clf = DecisionTreeClassifier()
dt_clf.fit(X_train, y_train)
train_preds = dt_clf.predict(X_train)

dt_cv_score = cross_val_score(dt_clf, X_train, y_train, cv=50)
mean_dt_cv_score = np.mean(dt_cv_score)

print_metrics(y_train, train_preds)

print(f"Mean Cross Validation Score: {mean_dt_cv_score :.2%}")
```

```
Precision Score: 1.0
Recall Score: 1.0
Accuracy Score: 1.0
F1 Score: 1.0
Mean Cross Validation Score: 91.60%
```

Whoa! We got a perfect score on our decision tree, with a mean cross validation accuracy of 91.48%. So, we're overfit with this particular model, but we have our highest accuract so far. This looks promising. Let's continue down this path and run a "quick" Grid CV and see what we can do with the current model and dataset. We can update a few of the hyper parameters and see if this improves. We know that decision trees tend to overfit.

In [29]:
```
dt_grid_clf = DecisionTreeClassifier()

dt_param_grid = {"criterion": ("gini", "entropy"),
                 "max_depth": ["None", 2,3,4,5,6],
                 "min_samples_split": [2,5,10],
                 "min_samples_leaf": [1,2,3,4,5,6]}

gs_dt_grid = GridSearchCV(dt_grid_clf, dt_param_grid, cv=5)
gs_dt_grid.fit(X_train, y_train)

gs_dt_grid.best_params_
```

```
C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\model_sel
ection\_validation.py:548: FitFailedWarning: Estimator fit failed. The scor
e on this train-test partition for these parameters will be set to nan. Det
ails:
Traceback (most recent call last):
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\m
odel_selection\_validation.py", line 531, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\t
ree\_classes.py", line 890, in fit
    super().fit(
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\t
ree\_classes.py", line 276, in fit
    if max_depth <= 0:
TypeError: '<=' not supported between instances of 'str' and 'int'

  warnings.warn("Estimator fit failed. The score on this train-test"
C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\model_sel
ection\_validation.py:548: FitFailedWarning: Estimator fit failed. The scor
```

Okay, so we've run into "a few" warnings. Let's take these results and run a new Decision Tree to see what we've got.

In [30]:
```
dt_clf = DecisionTreeClassifier(criterion = 'gini', max_depth = 6, min_sam
dt_clf.fit(X_train, y_train)
train_preds = dt_clf.predict(X_train)

dt_cv_score = cross_val_score(dt_clf, X_train, y_train, cv=50)
mean_dt_cv_score = np.mean(dt_cv_score)

print_metrics(y_train, train_preds)

print(f"Mean Cross Validation Score: {mean_dt_cv_score :.2%}")
```

```
Precision Score: 0.9743589743589743
Recall Score: 0.7430167597765364
Accuracy Score: 0.9603841536614646
F1 Score: 0.8431061806656102
Mean Cross Validation Score: 93.88%
```

Okay, this is promising. We have really strong scores here. Precision, Recall, and F1 are up across the board. The Decision tree appears to be the best "base" model, before any boosting or further feature engineering. Let's take a look at what features are prominent and re-examine the

data from here. We have a mean cross validation score across our training data as 94.12%. This is almost a 9% increase over our 85%. Let's evaluate this in more depth.

Let's see the test data.

In [31]:
```
train_preds = dt_clf.predict(X_test)

print_metrics(y_test, train_preds)

```

```
Precision Score: 0.8627450980392157
Recall Score: 0.704
Accuracy Score: 0.9388489208633094
F1 Score: 0.7753303964757708
```

Okay, this is mixed. The precision number on our test data is 86%. This means, when our model predicts a churn, it still misses almost 13%, which isn't bad. Our recall shows that we have more false negatives than we do false positives, but our overall accuracy is still at 94%, but let's go back to our original goal here. To predict and identify churn and make some recommendations to stop it. Let's evaluate this model and determine next steps.

Evaluation

Now we have done some baseline modeling, we seemed to have settled on a Decision Tree approach which gives us some of best preliminary results. It's a good time to see what we have an tweak the model from there.

Remember, our goal here is to determine why the churn is happening. A 91% precision rate means we've probably found something, however, we're still missing almost 10% of the true positives. It appears that we still have considerable amount of false negatives, but our overall accuracy to almost 95%, which is higher (an increase of 10%) than our original baseline accuracy.

It's important to consider the problem - churn. Churn means people want to leave their phone provider. We know that there will be some cases we can't predict, and it's likely we'll be battling a higher number of false negatives than false positives. Just think, people quit their phone service for a variety of reasons related to moving, employment, our life considerations that may not show up on the data. However, it might be nice to get more than a 10% increase in prediction here.

Sampling

In our final Decision Tree model, we had a cross validation accuracy of 94%. When we went to our test data, it went to roughly the same. This suggests that our model underfit, maybe, but not by much.

Let's compare the churn rate for our original data.

In [32]:
```
#original data
print_churn (df['churn'])
```

```
There a total of 3333 entries.
The percentage of users who churn is 14.49%.
We do have class imbalance concerns.
So... our model should have a higher accuracy than 85.51% at predicting who w
ill stay.
```

In [33]:
```
def print_churn_int (target):

    print(f'There a total of {len(target)} entries.')
    class_imbalance = " "
    churn_percentage = len(target[target == 1])/len(target)
    if churn_percentage > .2:
        class_imbalance = " not "
    print(f'The percentage of users who churn is {churn_percentage :.2%}.'
    print(f'We do{class_imbalance}have class imbalance concerns.')
    print(f'So... our model should have a higher accuracy than {(1 - churn_

print_churn_int (y_train)
print_churn_int (y_test)

```

```
There a total of 2499 entries.
The percentage of users who churn is 14.33%.
We do have class imbalance concerns.
So... our model should have a higher accuracy than 85.67% at predicting who w
ill stay.
There a total of 834 entries.
The percentage of users who churn is 14.99%.
We do have class imbalance concerns.
So... our model should have a higher accuracy than 85.01% at predicting who w
ill stay.
```

These splits are reasonable - they each have a similar churn rate so there isn't much difference between them.

Sample and Data

As we noted in our original assessment, there are class imbalance concerns because we only have a churn rate of about 15%. We created our original train/test split and have been careful since then to use cross-validation to mitigate an overfitting, we could be missing something with a relatively small sample size to work with. One way to mitigate that would be using a SMOTE technique to create a new population for a train-test split, and then rerun our decision tree from there.

Modeling Techniques

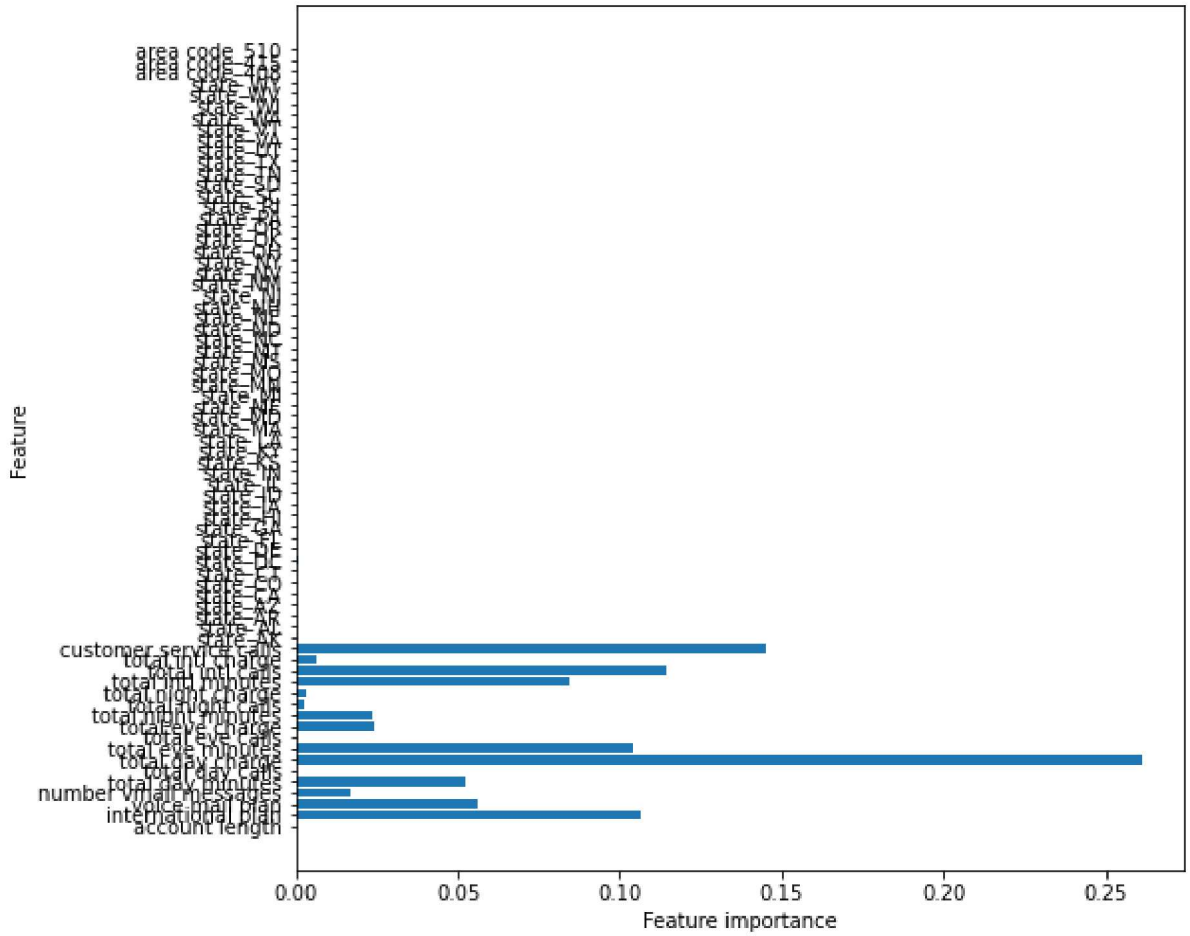
We used three different modeling techniques to train on our data: KNN, Logistic Regression, and Decision Tree. With KNN and Logistic Regression, we only achieved small bumps in accuracy and F1. The Decision Tree with some hyperparamater tuning yielded more favorable results.

Going forward, we'll use a Decision Tree on this data. The advantage with the Decision tree is we don't have to utilize much scaling and we can optimize later with Random Forests or other boosting.

Features

So, we like our model, but how do we explain it. What features really drive the success of the

In [34]:
```
def plot_feature_importances(model):
    n_features = X_train.shape[1]
    plt.figure(figsize=(8,8))
    plt.barh(range(n_features), model.feature_importances_, align='center'
    plt.yticks(np.arange(n_features), X_train.columns.values)
    plt.xlabel('Feature importance')
    plt.ylabel('Feature')

plot_feature_importances(dt_clf)
```



Perfect. This graph tells us our important features. From here we can see that Total Day charge features most prominently but not far in front of the next feature, which is customer service calls. Thinking about these factors make sense intuitively, those people who talk on the phone a lot may have higher bills, and the most likely to quit. But... we don't have a feature for total charge. We have different categories of evening, daytime, and international charge, but no total.

Also, those who most call customer service a lot are probably disappointed with their service.

Also, of note, there's is little if any importance attached to the state, as well as the area code. It could be useful to delete these variables going forward in order to speed up processing time. And refocus the training of the models.

Summary

We reviewed the modeling techniques and well, we're moving on. We had the best preliminary results with the Decision Tree, so we will continue to utilize that model, and perhaps boost it later.

We're going to remove the state and Area Code features all together. We're going to create new feature which shows us the total charge, and perhaps convert the international call to a charge per minute number. We're going to remove the state and area code features all together, as they seem to have little effect.

Sampling - We're going to utilize the SMOTE technique on our data and train on models on that. This should increase our accuracy and limit and issues related to class imbalance.

So... let's give it a shot.

Iteration

Now that we've taken a first crack at the modeling, we need to adjust some things. We're going to do some feature engineering, some resampling, and then we will rerun our Decision Tree model, with some Random Forest modeling if needed.

Features

Features. As we mentioned previously, we're going to do a little feature engineering. We may even violate some collinearity guidelines in the process. Who knows!

So, we're going to rexamine our original dataset, and remove state and area code altogether. Additionally, we're going to create a column called total charge. While we're at it, let's modify the evening charge to be a cost per to determine the average cost/minute to see if we can see anything there.

In [35]:
```
#pull up df again
df
```

Out[35]:

	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	...
0	KS	128	415	382-4657	no	yes	25	265.1	110	45.07	...
1	OH	107	415	371-7191	no	yes	26	161.6	123	27.47	...
2	NJ	137	415	358-1921	no	no	0	243.4	114	41.38	...
3	OH	84	408	375-9999	yes	no	0	299.4	71	50.90	...
4	OK	75	415	330-6626	yes	no	0	166.7	113	28.34	...
...	...	...	...	...	...	...	...	...	...	...	...
3328	AZ	192	415	414-4276	no	yes	36	156.2	77	26.55	...
3329	WV	68	415	370-3271	no	no	0	231.1	57	39.29	...
3330	RI	28	510	328-8230	no	no	0	180.8	109	30.74	...
3331	CT	184	510	364-6381	yes	no	0	213.8	105	36.35	...
3332	TN	74	415	400-4344	no	yes	25	234.4	113	39.85	...

3333 rows × 21 columns

In [36]:

```
#let's separate our target to avoid date leakage
y_iter = df['churn']

#let's create our features set and drop out the phone number, like we did p
X_iter = df.drop(columns = ['area code', 'phone number', 'state', 'churn']
X_iter

#let's convert the yes/nos of the various plans to 1/0s, just as we did pre
d = {'no': 0, 'yes': 1}
X_iter['international plan'] = X_iter['international plan'].map(d).fillna(
X_iter['voice mail plan'] = X_iter['voice mail plan'].map(d).fillna(X_iter
X_iter.head()
```

Out[36]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	total eve calls	total eve charge	to nig minut
0	128	0	1	25	265.1	110	45.07	197.4	99	16.78	24
1	107	0	1	26	161.6	123	27.47	195.5	103	16.62	25
2	137	0	0	0	243.4	114	41.38	121.2	110	10.30	16
3	84	1	0	0	299.4	71	50.90	61.9	88	5.26	19
4	75	1	0	0	166.7	113	28.34	148.3	122	12.61	18

In [37]:

```
#we're going to create a new column, total charge, as well as three new co
X_iter['total charge'] = X_iter['total day charge'] + X_iter['total eve ch
X_iter
```

Out[37]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	total eve calls	total eve charge	m
0	128	0	1	25	265.1	110	45.07	197.4	99	16.78	
1	107	0	1	26	161.6	123	27.47	195.5	103	16.62	
2	137	0	0	0	243.4	114	41.38	121.2	110	10.30	
3	84	1	0	0	299.4	71	50.90	61.9	88	5.26	
4	75	1	0	0	166.7	113	28.34	148.3	122	12.61	
...	...	...	...	...	...	...	...	...	...	...	
3328	192	0	1	36	156.2	77	26.55	215.5	126	18.32	
3329	68	0	0	0	231.1	57	39.29	153.4	55	13.04	
3330	28	0	0	0	180.8	109	30.74	288.8	58	24.55	
3331	184	1	0	0	213.8	105	36.35	159.6	84	13.57	
3332	74	0	1	25	234.4	113	39.85	265.9	82	22.60	

3333 rows × 18 columns

That was successful. It looks like we were able to create the column. Before we delete these columns, let's just make sure that the minutes of each price tier is constant. I'm going to add a new column and then describe it.

In [38]:
```
X_iter['day rate'] = X_iter['total day charge'] / X_iter['total day minute
X_iter['intl rate'] = X_iter['total intl charge'] / X_iter['total intl min
X_iter['eve rate'] = X_iter['total eve charge'] / X_iter['total eve minute
X_iter['night rate'] = X_iter['total night charge'] / X_iter['total night
```

In [39]:
```
X_iter.describe()
```

Out[39]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total da charg
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.00000
mean	101.064806	0.096910	0.276628	8.099010	179.775098	100.435644	30.56230
std	39.822106	0.295879	0.447398	13.688365	54.467389	20.069084	9.25943
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
25%	74.000000	0.000000	0.000000	0.000000	143.700000	87.000000	24.43000
50%	101.000000	0.000000	0.000000	0.000000	179.400000	101.000000	30.50000
75%	127.000000	0.000000	1.000000	20.000000	216.400000	114.000000	36.79000
max	243.000000	1.000000	1.000000	51.000000	350.800000	165.000000	59.64000

8 rows × 22 columns

Okay, so the standard rates ARE indeed flat as dollars. Evening rate is .085/min. Day rate is .17/min, Night is .045/min, Intl rate is .27/min.

In [40]:
```
X_iter.drop(columns = ['total day charge', 'total eve charge', 'total nigh
X_iter.describe()
```

Out[40]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total ev minute
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.00000
mean	101.064806	0.096910	0.276628	8.099010	179.775098	100.435644	200.98034
std	39.822106	0.295879	0.447398	13.688365	54.467389	20.069084	50.71384
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
25%	74.000000	0.000000	0.000000	0.000000	143.700000	87.000000	166.60000
50%	101.000000	0.000000	0.000000	0.000000	179.400000	101.000000	201.40000
75%	127.000000	0.000000	1.000000	20.000000	216.400000	114.000000	235.30000
max	243.000000	1.000000	1.000000	51.000000	350.800000	165.000000	363.70000

In [41]:
```
y_iter = y_iter.astype(int)
```

Fantastic! It looks like we cleaned up this data... for now. A couple of takeaways, because the charges are a flat rate, it turns out the minutes and charges for each pricing tier are collinear. By combining the charges into one total charge, we've streamlined our data, solved potential collinearity issues, and have a more "customer-focused" column.

In [42]:
```
#create a new train-test split from our new and improved data
X_train_1, X_test_1, y_train_1, y_test_1 = train_test_split(X_iter, y_iter
```

In [43]:
```
# let's see how our Decision Tree looks
dt_grid_clf_1 = DecisionTreeClassifier()

dt_param_grid_1 = {"criterion": ("gini", "entropy"),
                   "max_depth": ["None", 2,3,4,5,6],
                   "min_samples_split": [2,5,10],
                   "min_samples_leaf": [1,2,3,4,5,6]}

gs_dt_grid_1 = GridSearchCV(dt_grid_clf_1, dt_param_grid_1, cv=5)
gs_dt_grid_1.fit(X_train_1, y_train_1)

gs_dt_grid_1.best_params_
```

```
ails:
Traceback (most recent call last):
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\m
odel_selection\_validation.py", line 531, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\t
ree\_classes.py", line 890, in fit
    super().fit(
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\t
ree\_classes.py", line 276, in fit
    if max_depth <= 0:
TypeError: '<=' not supported between instances of 'str' and 'int'

  warnings.warn("Estimator fit failed. The score on this train-test"
C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\model_sel
ection\_validation.py:548: FitFailedWarning: Estimator fit failed. The scor
e on this train-test partition for these parameters will be set to nan. Det
ails:
Traceback (most recent call last):
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\m
```

In [44]:
```
#dt_clf_2 = DecisionTreeClassifier(criterion = 'gini', max_depth = 6, min_s

dt_clf_2 = DecisionTreeClassifier(criterion = 'gini', max_depth = 5, min_s
dt_clf_2.fit(X_train_1, y_train_1)
train_preds_2 = dt_clf_2.predict(X_train_1)

dt_cv_score_2 = cross_val_score(dt_clf_2, X_train_1, y_train_1, cv=50)
mean_dt_cv_score_2_resamp = np.mean(dt_cv_score_2)

print_metrics(y_train_1, train_preds_2)

print(f"Mean Cross Validation Score: {mean_dt_cv_score_2_resamp :.2%}")
```

```
Precision Score: 0.990506329113924
Recall Score: 0.8505434782608695
Accuracy Score: 0.9767907162865146
F1 Score: 0.9152046783625729
Mean Cross Validation Score: 97.64%
```

Whoa! Precision of 0.99, and accuracy of 97.6, with cross-validation of 97.6. These are amazing. Let's try it on the test-data.

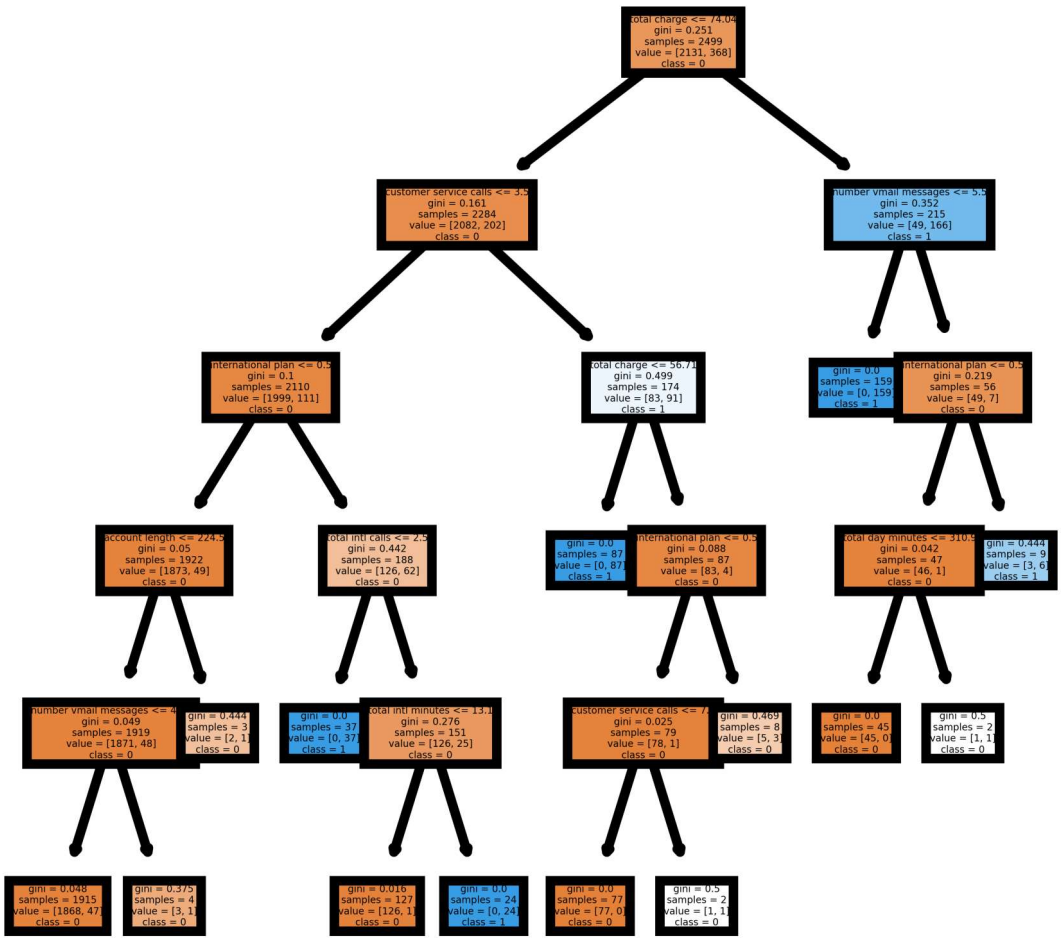
In [45]:
```
train_preds_2 = dt_clf_2.predict(X_test_1)

print_metrics(y_test_1, train_preds_2)
```

```
Precision Score: 0.9805825242718447
Recall Score: 0.8782608695652174
Accuracy Score: 0.9808153477218226
F1 Score: 0.926605504587156
```

Wow... These numbers are strong. 98% precision and 98% accuracy. F1 score of 92.7%. We have a recall score of 88% but remember, we're okay with a lower recal score because we accept they'll be a few stray churns. Let's take a look at the features.

In [46]:

```
#plot the feature
from sklearn import tree

fig, axes = plt.subplots(nrows = 1,ncols = 1, figsize = (2,2), dpi=2000)
tree.plot_tree(dt_clf_2,
               feature_names = X_test_1.columns,
               class_names=np.unique(target).astype('str'),
               filled = True)
plt.show()
```



Interesting that we see that total charge is the most defining feature. This is one we had to engineer, but it's clearly very indicative.

SMOTE

We talked about imbalance before, and ways to address it. We know our churn rate is only 15%. So let's create a more even sample size using a synthetic minority oversampling technique. You want this SMOTE? To do this, I'm create a brand new train-test set, prior to the synthetic

creation. We will verufy that the split has a similar churn rate at 15%

In [47]:
```
#now let's create a synthetic oversampling of our training data
X_resampled, y_resampled = SMOTE().fit_sample(X_train_1, y_train_1)

# Split resampled data into training and test sets
X_train_resamp, X_test_resamp, y_train_resamp, y_test_resamp = train_test_

# observe split for churn values
y_train_resamp.value_counts()
```

Out[47]:
```
1    1603
0    1593
Name: churn, dtype: int64
```

Aha! This looks good. We have a 50-50 split, and approximately 3200 piece of data, but now with a 50-50 split.

Decision Tree with CVGrid.

Now that we have a more complete model, it's a good opportunity to rerun a decision tree with a CV grid and scope out the goods.

In [48]:
```
dt_iter_grid = DecisionTreeClassifier()

dt_param_grid = {"criterion": ("gini", "entropy"),
                 "max_depth": ["None", 2,3,4,5,6],
                 "min_samples_split": [2,5,10],
                 "min_samples_leaf": [1,2,3,4,5,6]}

gs_iter_grid = GridSearchCV(dt_iter_grid, dt_param_grid, cv=5)
gs_iter_grid.fit(X_train_resamp, y_train_resamp)

gs_iter_grid.best_params_
```

```
C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\model_sel
ection\_validation.py:548: FitFailedWarning: Estimator fit failed. The scor
e on this train-test partition for these parameters will be set to nan. Det
ails:
Traceback (most recent call last):
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\m
odel_selection\_validation.py", line 531, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\t
ree\_classes.py", line 890, in fit
    super().fit(
  File "C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\t
ree\_classes.py", line 276, in fit
    if max_depth <= 0:
TypeError: '<=' not supported between instances of 'str' and 'int'

  warnings.warn("Estimator fit failed. The score on this train-test"
C:\Users\benne\anaconda3\envs\learn-env\lib\site-packages\sklearn\model_sel
ection\_validation.py:548: FitFailedWarning: Estimator fit failed. The scor
```

Okay, these results are similar to the ones we had previously. Except we used 'gini' before.

In [49]:
```
dt_clf_resamp = DecisionTreeClassifier(criterion = 'entropy', max_depth = 
dt_clf_resamp.fit(X_train_resamp, y_train_resamp)
train_preds_resamp = dt_clf_resamp.predict(X_train_resamp)

dt_cv_score_resamp = cross_val_score(dt_clf_resamp, X_train_resamp, y_trai
mean_dt_cv_score_resamp = np.mean(dt_cv_score_resamp)

print_metrics(y_train_resamp, train_preds_resamp)

print(f"Mean Cross Validation Score: {mean_dt_cv_score_resamp :.2%}")
```

```
Precision Score: 1.0
Recall Score: 0.7199001871490954
Accuracy Score: 0.859511889862328
F1 Score: 0.8371418208197317
Mean Cross Validation Score: 85.82%
```

This is very strong. Recall that, with our new SMOTE data, we had a roughly 50-50 chance of predicting churn. We achieved a precision score of nearly 100%, with an accuracy score of 89%... this is 39% higher than 50%. This is a strong performance on 50-50 data. Let's try this new model out now on some of our original training data, still using the information from our SMOTE sample.

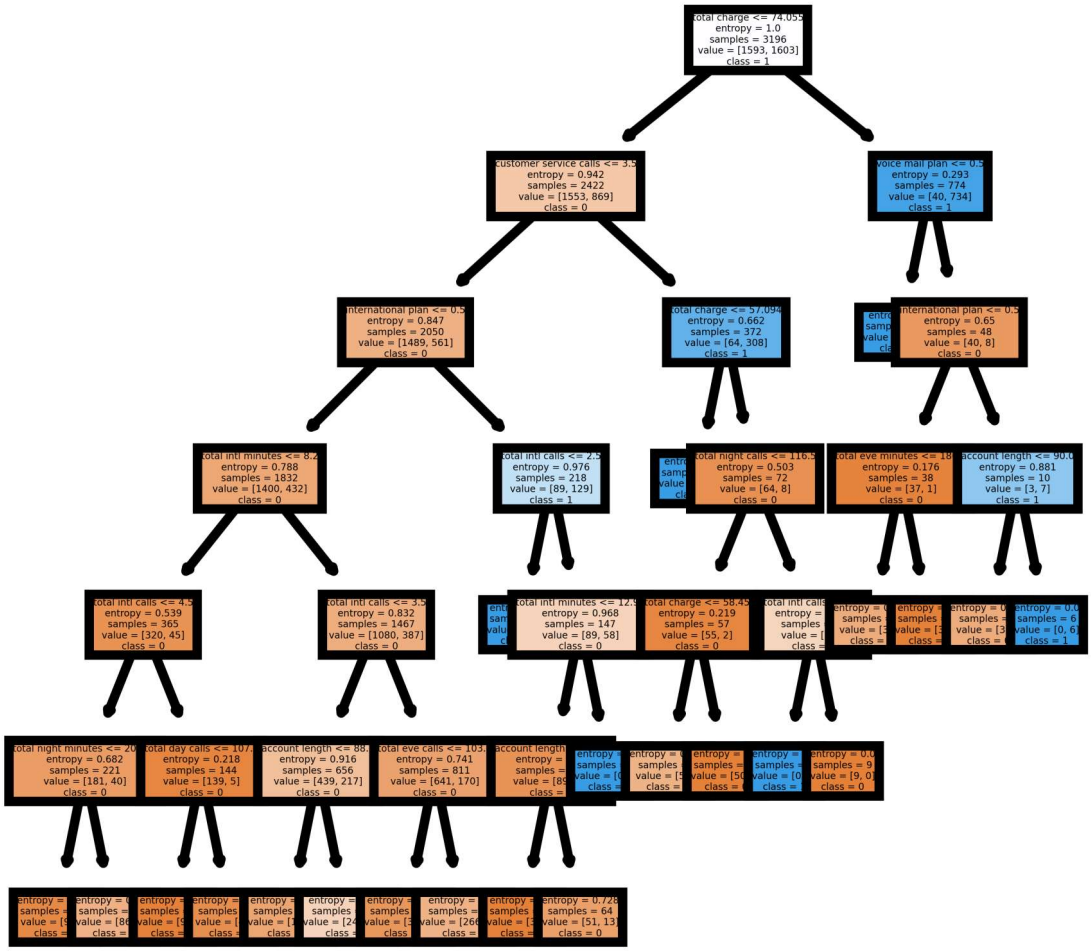
In [50]:
```
train_preds = dt_clf_resamp.predict(X_test_1)
print_metrics(y_test_1, train_preds)
```

```
Precision Score: 0.926605504587156
Recall Score: 0.8782608695652174
Accuracy Score: 0.973621103117506
F1 Score: 0.9017857142857143
```

This is good, but actually not as good as what we previously achieved. We have 97.1% accuracy with both an F1 score of 89%. Our recall improved, but remember, we probably care more about precision than we do recall. Precision dropped significantly. What's interesting here is that SMOTE, did not improve our score. Let's take a look at the tree it produced.

In [51]:

```
#plot the feature
from sklearn import tree

fig, axes = plt.subplots(nrows = 1,ncols = 1, figsize = (2,2), dpi=2000)
tree.plot_tree(dt_clf_resamp,
               feature_names = X_test_1.columns,
               class_names=np.unique(target).astype('str'),
               filled = True)
plt.show()
```



We still get some of the prominent features and similar leafs, but a few different leafs at the very bottom. We changed some of the parameters however, so that would explain it.

In [52]:
```
# Instantiate an AdaBoostClassifier
adaboost_clf = AdaBoostClassifier(random_state=42)

# Instantiate an GradientBoostingClassifier
gbt_clf = GradientBoostingClassifier(random_state=42)
```

In [53]:
```
# Fit AdaBoostingClassifier
adaboost_clf.fit(X_train, y_train)
```

Out[53]:
```
AdaBoostClassifier(random_state=42)
```

In [54]:
```
# Fit GradientBoostingClassifier
gbt_clf.fit(X_train, y_train)
```

Out[54]:
```
GradientBoostingClassifier(random_state=42)
```

In [55]:
```
# AdaBoost model predictions
adaboost_train_preds = adaboost_clf.predict(X_train)
adaboost_test_preds = adaboost_clf.predict(X_test)

# GradientBoosting model predictions
gbt_clf_train_preds = gbt_clf.predict(X_train)
gbt_clf_test_preds = gbt_clf.predict(X_test)
```

In [56]:
```
print_metrics(y_test_1, adaboost_test_preds)
```

```
Precision Score: 0.14285714285714285
Recall Score: 0.08695652173913043
Accuracy Score: 0.802158273381295
F1 Score: 0.1081081081081081
```

In [57]:
```
print_metrics(y_test_1, gbt_clf_test_preds)
```

```
Precision Score: 0.12244897959183673
Recall Score: 0.10434782608695652
Accuracy Score: 0.7733812949640287
F1 Score: 0.11267605633802817
```

Summary

Now that we've completed a fair bit of iteration modeling, we've stumbled upon a fairly precise and accurate model. Our `dt_clf_2` model on the newly updated data set. Let's recap.

We updated our features to get rid of the area codes and state features. We also combined all of the charges into one, total charge. This eliminated some collinearity concerns, and also produced a, strong, customer relevant feature.

We continued using our Decision Tree Model, and reran it on the new data. We got some of our best numbers, with precision and accuracy numbers at 98% for test data, and 100% and 98%, respectively for training data. Total customer charge turned out to be our primary feature on our decision tree, with the boudary of <=74.04.

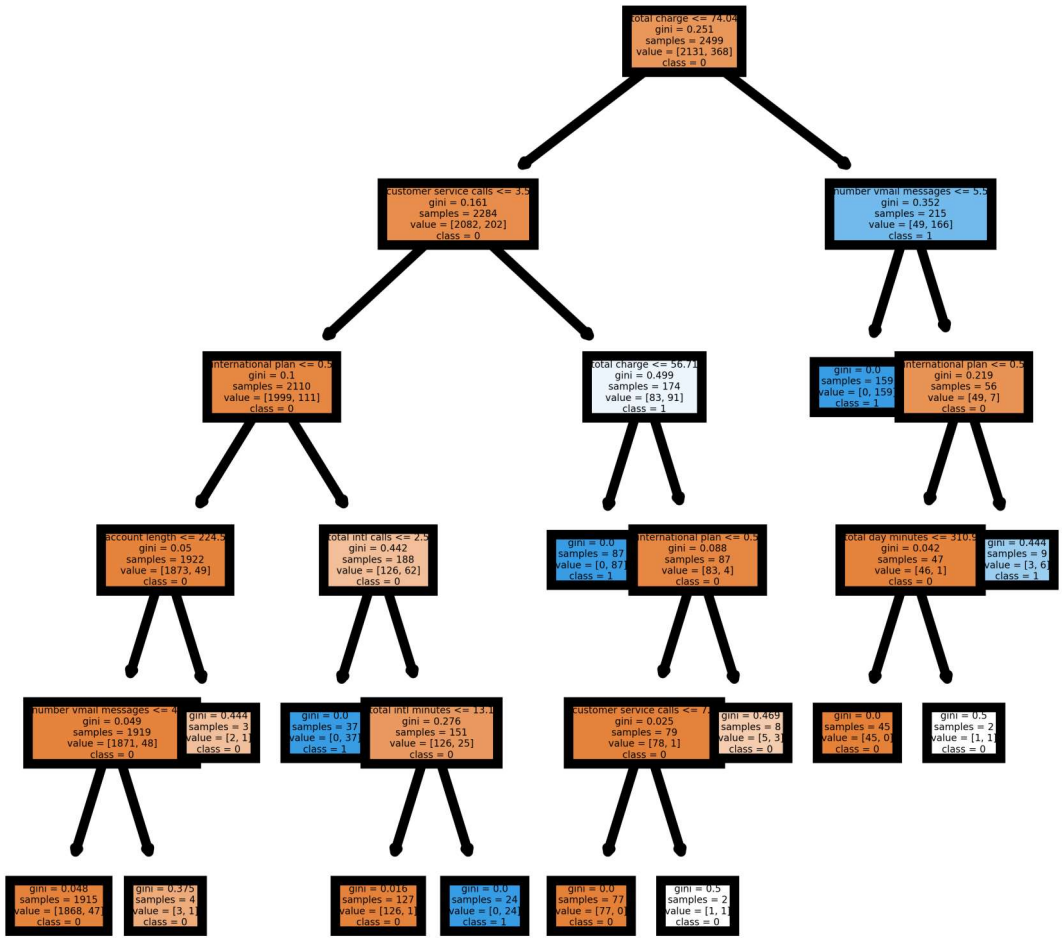
Our other models, did not produce results as impressive. We attempted to resample the data by creating synthetic data points to achieve a 50-50 churn rate. By running a decision tree model here, we thought we could produce better results, which didn't happen. Furthermore, other methods Adaboost and Gradient Boosting didn't produce results better than our straightforward

Conclusion

So, we've got some positive results with which to analyze this model. Remember, we did a little feature engineering to create a column called total charge, and that turned out to be our most important feature with regard to predicting outcome. This makes sense when we think of our business problem - why are we losing customers? Well, maybe the biggest predictor is how much they pay. Let's do a deeper dive into our model tree to see what's going on.

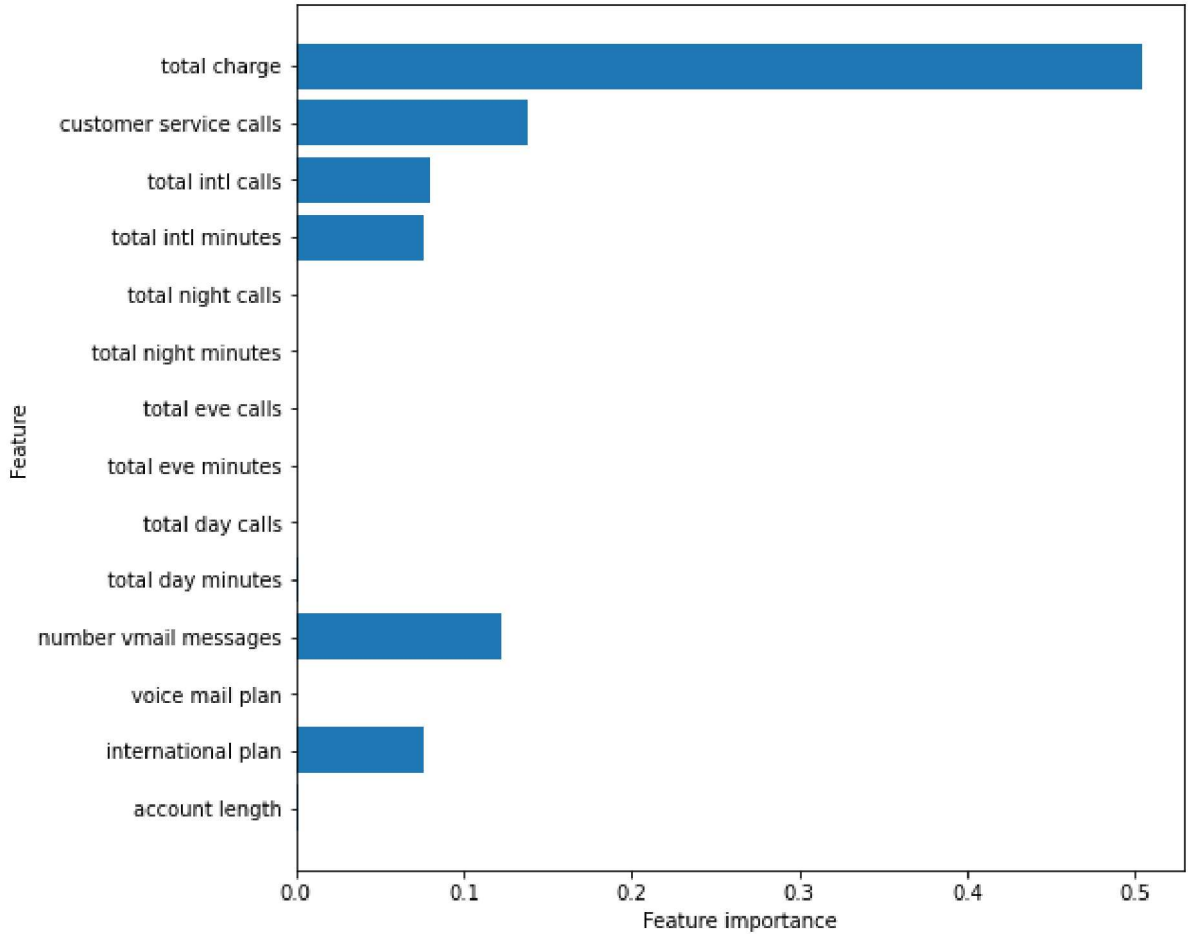
In [58]:

```
#replot the decision tree of our dt_clf_2 model
fig, axes = plt.subplots(nrows = 1,ncols = 1, figsize = (2,2), dpi=2000)
tree.plot_tree(dt_clf_2,
               feature_names = X_test_1.columns,
               class_names=np.unique(target).astype('str'),
               filled = True)
plt.show()
```



In [59]:

```
def plot_feature_importances_2(model):
    n_features = X_test_1.shape[1]
    plt.figure(figsize=(8,8))
    plt.barh(range(n_features), model.feature_importances_, align='center'
    plt.yticks(np.arange(n_features), X_test_1.columns.values)
    plt.xlabel('Feature importance')
    plt.ylabel('Feature')

plot_feature_importances_2(dt_clf_2)
```



What we can gather from here our really 5 positive churn leaves representing unique cases.

Analysis

We have one leaf representing total_charge > 74.04 and no voicemale plan. This has 159 occurences, or almost half of the churned customers in our training data. We don't know much about a voice male plan, but it's safe to say this user base has high engagement with the product, but it budget constrained for whatever reason. Love to talk but don't want to pay, and maybe will price shop. We need to figure out how to keep these high rollers happy!

Next we have 87 churned users who's total charge is less than 56.01, but has 4 or more customer service calls. That's about 25% of the churned base. These users are not happy with the servcice, and are not engaged. Perhaps we don't chase this group.

Next we have 37 churned who pay less than 74.04, have 3 or fewer customer service calls, have the internation plan but make fewer than 3 calls a month. So, this is an issue of low engagement - people with international plans who aren't using it. Hey! call your mom people!

Next we have 24 churned users who pay less than 74.04, have 3 or fewer customer service calls, and make more than 3 international calls with greater than 13 minutes. Safe to say this is the budget concious international crew. The have strong engagement and perhaps don't want to pay for the service they use. This could be a group we care about but represents

Then, in a very small bucket, we have approximately 5 users who pay more than 74.04 a month

Original Data

Now that we have all of this information, let's abandon the model altogether and just look at data that we can filter based on this model.

In [60]:
```
#let's concatenate our two dataframes back to one
X_final = pd.concat([X_iter, y_iter], axis = 1)

#create new column by multiplying two existing columns
X_final['churn_cost'] = X_final['churn'] * X_final['total charge']

#calculate the total monthly cost of churn
total_churn = X_final['churn_cost'].sum()
total_churn
```

Out[60]: 31566.93

In [61]:
```
#let's calculate our total monthly cost
total_cost = X_final['total charge'].sum()
total_cost
```

Out[61]: 198146.03

so... quickly we can see that the total churn cost is actually very similar to the churn rate... around 15% but... maybe we know that's not the story. Much of the churn, almost 50%, comes from spending a lot of money. The churn scenario

Let's use our buckets from our model to determine how much contribution of the 5 buckets we have both as a user and cost base.

In [62]:
```
#let's isolate the data that is churn
churn = X_final[X_final['churn'] == 1]
no_churn = X_final[X_final['churn'] == 0]
```

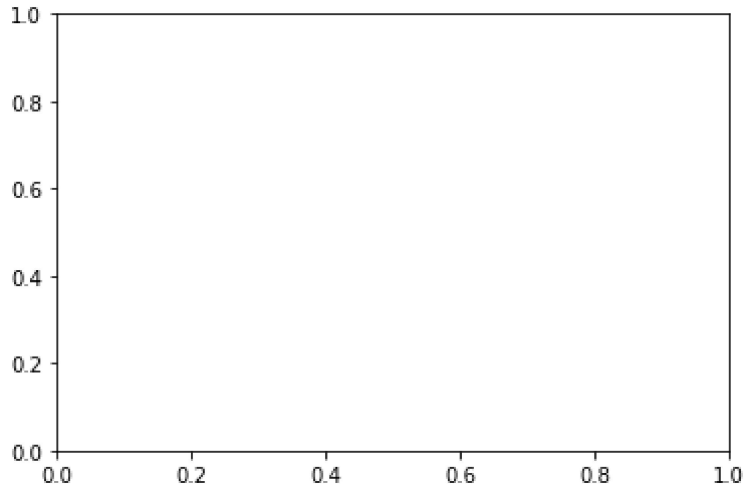
In [63]:

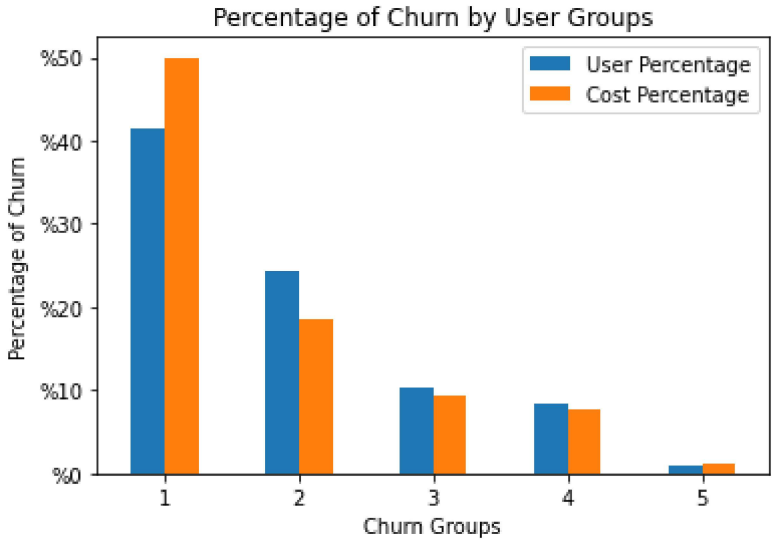
```
#now, let's create the various buckets
group_1 = churn[(churn['total charge'] > 74.04) & (churn['voice mail plan'
group_2 = churn[(churn['total charge'] <= 57.173) & (churn['customer servi
group_3 = churn[(churn['total charge'] <= 74.04) & (churn['customer servic
group_4 = churn[(churn['total charge'] <= 74.04) & (churn['customer servic
group_5 = churn[(churn['total charge'] > 74.04) & (churn['voice mail plan'

#let's create a dataframe from a dictionary where the buckets are represen
groups = [group_1, group_2, group_3, group_4, group_5]
percentages = {}
ctr = 1

for group in groups:
    percentages[ctr] = [100*(len(group)/len(churn)), 100*group['total char
    ctr+=1

group_d = pd.DataFrame(percentages)
group_df = group_d.transpose()
group_df.columns = ["User Percentage", "Cost Percentage"]

unaccounted_churn_users = 100 - group_df["User Percentage"].sum()
unaccounted_churn_charge =  100 - group_df["Cost Percentage"].sum()
unaccounted_churn_users
```

Out[63]: 14.492753623188406

In [64]:

```
fig, ax = plt.subplots()

ax = group_df.plot.bar(rot=0)

ax.set_xlabel("Churn Groups")
ax.set_ylabel("Percentage of Churn")
ax.set_title("Percentage of Churn by User Groups")
ax.legend(loc='upper right')
ax.yaxis.set_major_formatter('%{x:,.0f}')

```





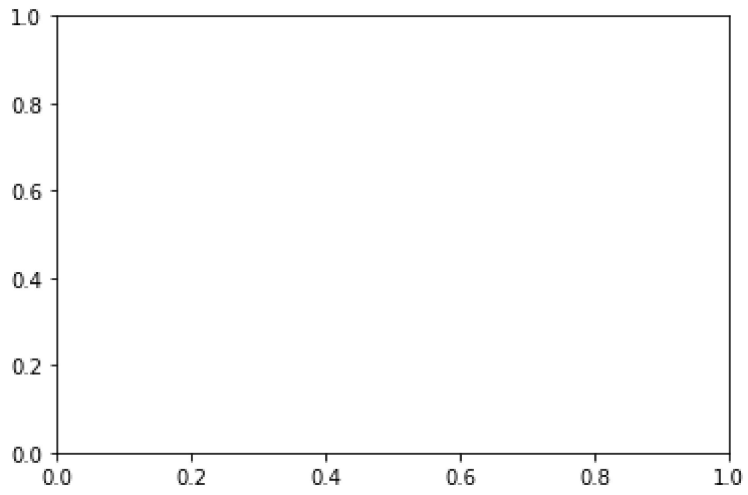
In [65]:

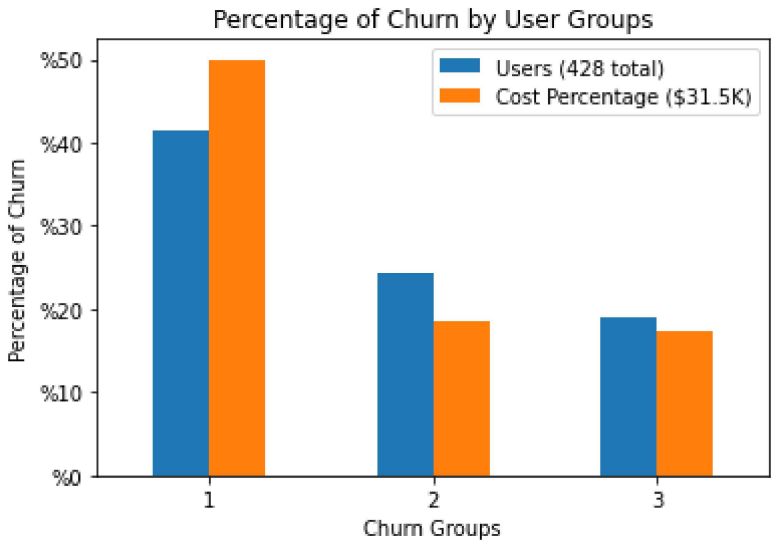
```
#now, let's create the various buckets
group_1 = churn[(churn['total charge'] > 74.04) & (churn['voice mail plan'
group_2 = churn[(churn['total charge'] <= 57.173) & (churn['customer servi
group_3 = churn[(churn['total charge'] <= 74.04) & (churn['customer servic
#group_4 = churn[(churn['total charge'] <= 74.04) & (churn['customer servi
#group_5 = churn[(churn['total charge'] > 74.04) & (churn['voice mail plan

#let's create a dataframe from a dictionary where the buckets are represen
groups = [group_1, group_2, group_3]
percentages = {}
ctr = 1

for group in groups:
    percentages[ctr] = [100*(len(group)/len(churn)), 100*group['total char
    ctr+=1

group_d = pd.DataFrame(percentages)
group_df = group_d.transpose()
group_df.columns = ["Users (428 total)", "Cost Percentage ($31.5K)"]

unaccounted_churn_users = 100 - group_df["Users (428 total)"].sum()
unaccounted_churn_charge =  100 - group_df["Cost Percentage ($31.5K)"].sum
len(group_1)

len(X_final[(X_final['total charge'] > 74.04) & (X_final['voice mail plan'
```

Out[65]: 200

In [66]:

```
fig, ax = plt.subplots()

ax = group_df.plot.bar(rot=0)

ax.set_xlabel("Churn Groups")
ax.set_ylabel("Percentage of Churn")
ax.set_title("Percentage of Churn by User Groups")
ax.legend(loc='upper right')
ax.yaxis.set_major_formatter('%{x:,.0f}')
```



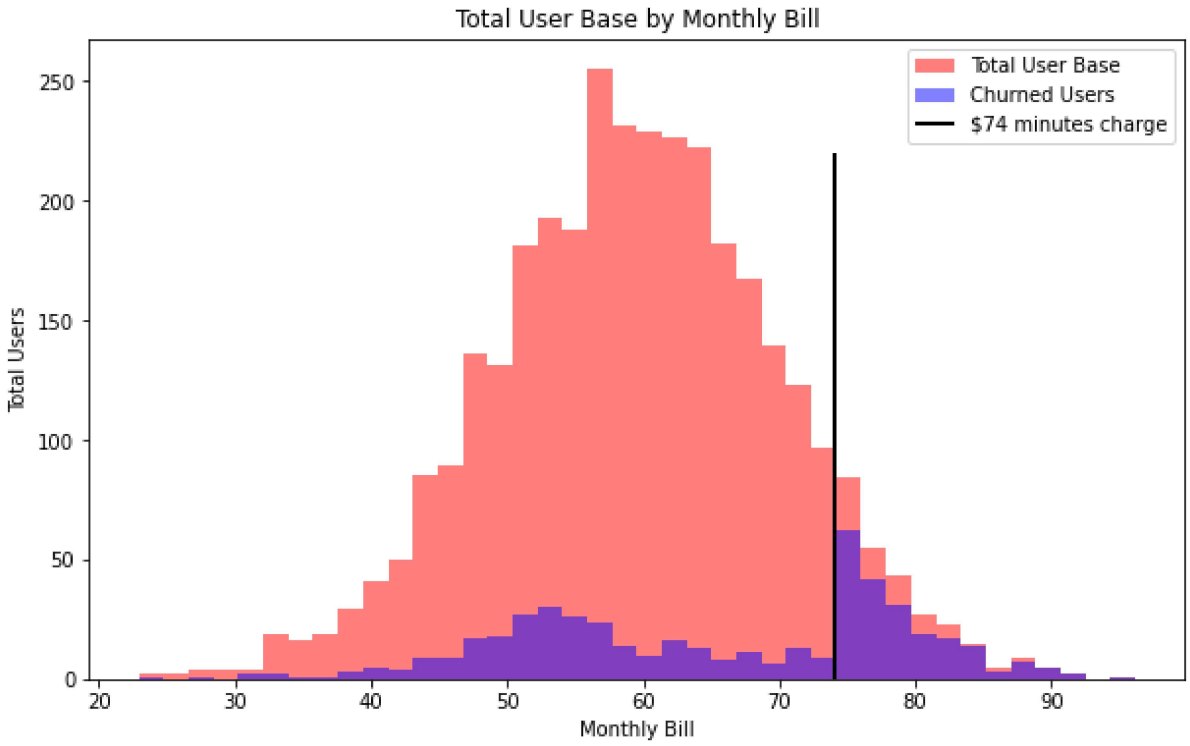


In [67]:

```
group_1 = churn[(churn['total charge'] > 74.04) | (churn['customer service

#let's create a dataframe from a dictionary where the buckets are represent
#groups = [group_1, group_2, group_3]
#percentages = {}
#ctr = 1

#for group in groups:
#    percentages[ctr] = [100*(len(group)/len(churn)), 100*group['total char
#    ctr+=1

#group_d = pd.DataFrame(percentages)
group_df = group_d.transpose()
group_df.columns = ["Users (3333 total)", "Cost Percentage ($31.5K)"]

unaccounted_churn_users = 100 - group_df["Users (3333 total)"].sum()
unaccounted_churn_charge =  100 - group_df["Cost Percentage ($31.5K)"].sum
unaccounted_churn_users
```

Out[67]: 15.320910973084878

In [68]:
```
churn = X_final[X_final['churn'] == 1]
no_churn = X_final[X_final['churn'] == 0]

fig, ax = plt.subplots(figsize = (10,6))

ax.hist(X_final['total charge'], bins = 40, color = 'red', alpha = .5, lab
ax.hist(churn['total charge'], bins = 40, color = 'blue', alpha = .5, labe
ax.vlines(x=74.04, ymin=.0, ymax=220, colors='black', ls='solid', lw=2, la

ax.set_xlabel("Monthly Bill")
ax.set_ylabel("Total Users")
ax.set_title("Total User Base by Monthly Bill")
legend = ax.legend(loc='upper right')

plt.show()
```



As we can see, there's a disturbing peak in the data as the monthly bill increases. Once we hit a certain threshold in monthly spend, we're at a high risk of losing users. Let's take a closer look at the buckets here.

In [69]:
```
#let's look at the two biggest scenarios
high_spend = X_final[(X_final['total charge'] > 74.04) & (X_final['voice m
#low_spend_customerservice = X_final[(X_final['total charge'] <= 57.173) &
#int_low_eng = X_final[(X_final['total charge'] <= 74.04) & (X_final['custo

high_spend_churn_perc = len(churn[churn['total charge'] > 74.04])/len(X_fi
high_spend_churn_perc
```

Out[69]: 0.7527272727272727

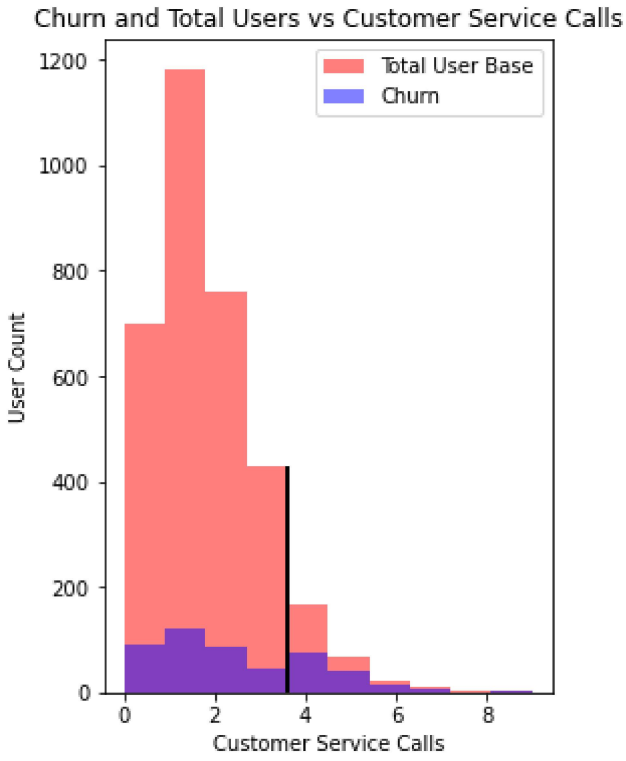
Okay, so we have 75% of our users, once they his this threshold, we'll leave. This is easily the largest predictor and accounts for about 50%, but cost, of our churn base.

In [70]:
```
high_spend_churn_vm_perc = len(churn[(churn['total charge'] > 74.04) & (ch
high_spend_churn_vm_perc
```

Out[70]: 0.966183574879227

Additionally, of those we lose at this higher spend, 97% of them do not have a voice mail plan. Let's take a step back, and see if voice mail really matters. Perhaps we can looked at a stacked histogram of voicemail users.

In [71]:
```
fig, ax = plt.subplots(figsize = (4,6))

ax.hist(X_final['customer service calls'], bins = 10, color = 'red', alpha
ax.hist(X_final[X_final['churn'] == 1]['customer service calls'], bins = 1

ax.set_xlabel("Customer Service Calls")
ax.set_ylabel("User Count")
ax.set_title("Churn and Total Users vs Customer Service Calls")
legend = ax.legend(loc='upper right')
ax.vlines(x=3.6, ymin=.0, ymax=430, colors='black', ls='solid', lw=2, labe

plt.show()
```

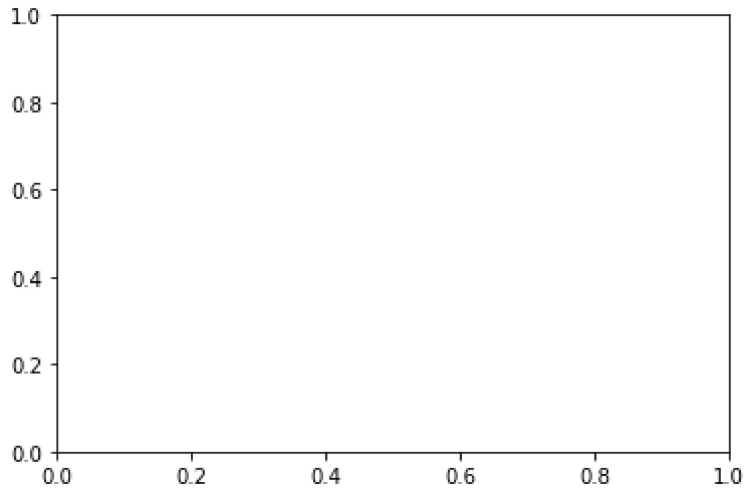


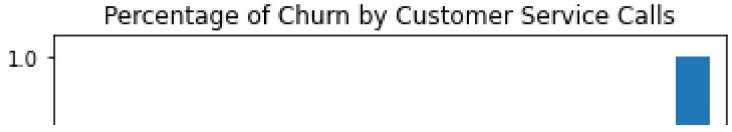
In [72]:

```
#now, let's create the various buckets
#cust_serv_1 = churn[(churn['customer service calls'] > 74.04) & (churn['v
#group_2 = churn[(churn['total charge'] <= 57.173) & (churn['customer serv
#group_3 = churn[(churn['total charge'] <= 74.04) & (churn['customer servi
#group_4 = churn[(churn['total charge'] <= 74.04) & (churn['customer servi
#group_5 = churn[(churn['total charge'] > 74.04) & (churn['voice mail plan

#let's create an array of the percentage of churn vs all users by number o

percentages = []
max_calls = 10 #from the describe file

for calls in range(0,max_calls):
    percentages.append(len(churn[(churn['customer service calls'] == calls

percentage_df = pd.DataFrame(percentages)
```

In [73]:

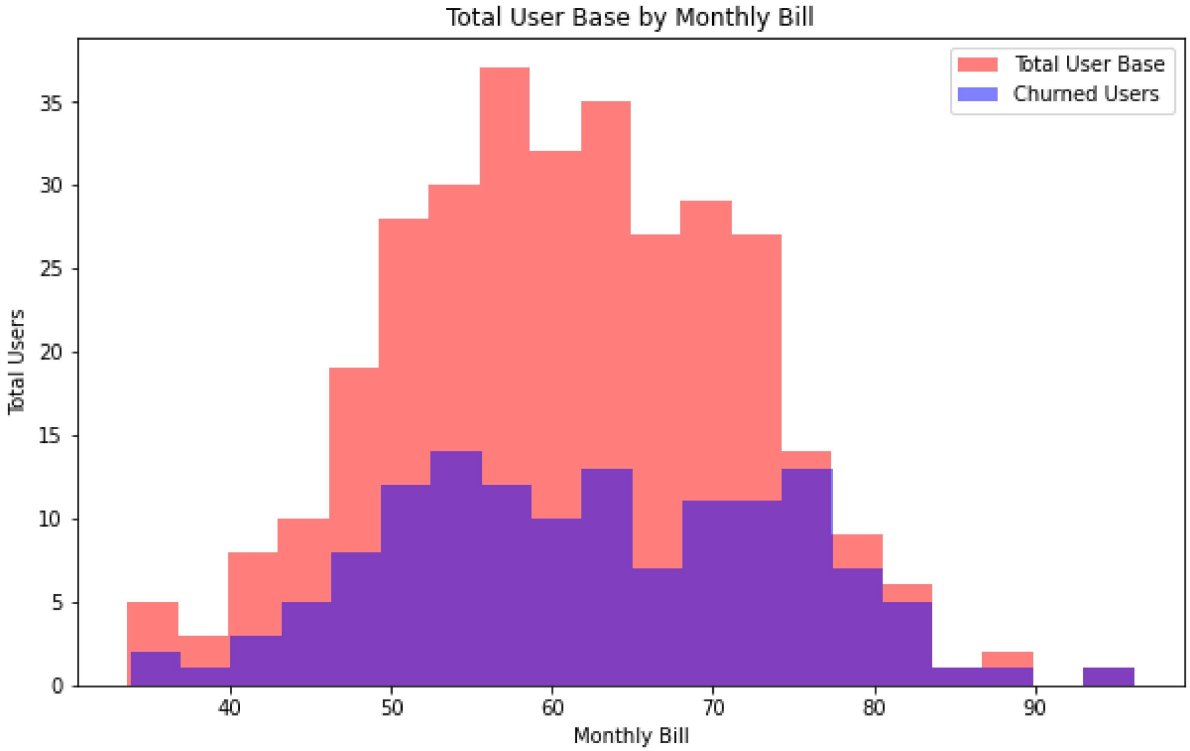
```
fig, ax = plt.subplots()

ax = percentage_df.plot.bar(rot=0)

ax.set_xlabel("Customer Service Calls")
ax.set_ylabel("Percentage Churn")
ax.set_title("Percentage of Churn by Customer Service Calls")
ax.legend().remove()

```





In [74]:

```
int_churn = X_final[(X_final['international plan'] == 1) & (X_final['churn
int_ = X_final[X_final['international plan'] == 1]

fig, ax = plt.subplots(figsize = (10,6))

ax.hist(int_['total charge'], bins = 20, color = 'red', alpha = .5, label
ax.hist(int_churn['total charge'], bins = 20, color = 'blue', alpha = .5,


ax.set_xlabel("Monthly Bill")
ax.set_ylabel("Total Users")
ax.set_title("Total User Base by Monthly Bill")
legend = ax.legend(loc='upper right')

plt.show()
```



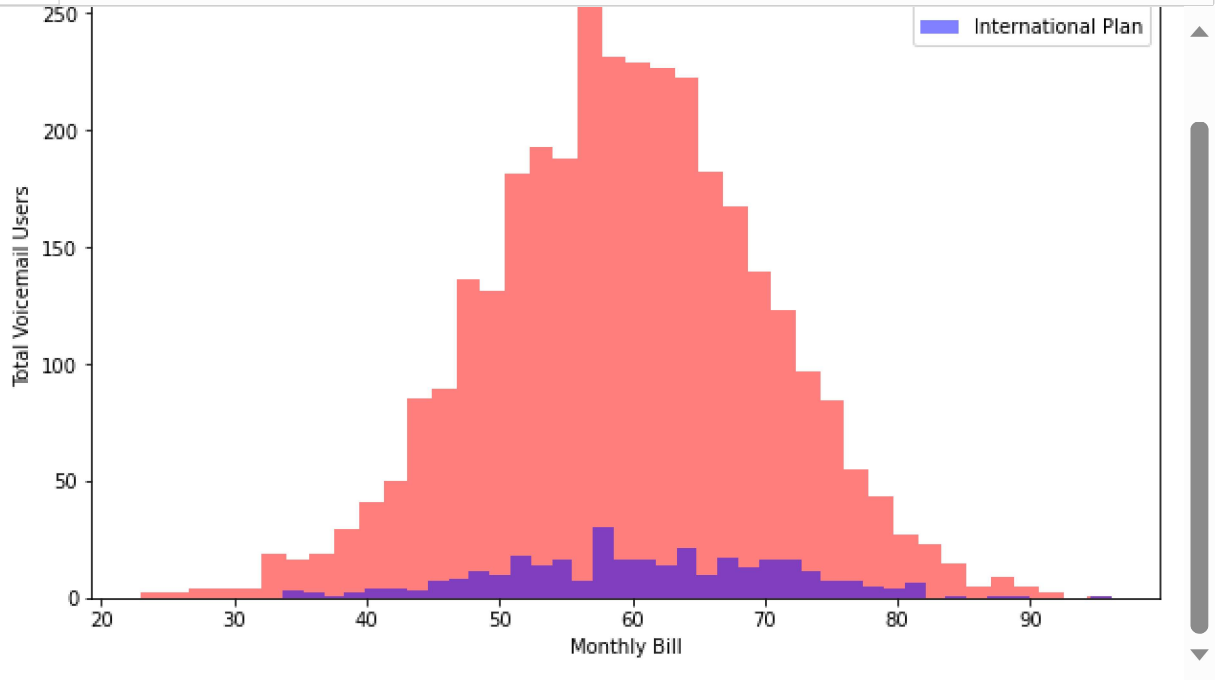
In [75]:

```
len(int_churn)/len(int_)
len(int_)/len(X_final)
len(X_final)


```

Out[75]: 3333

In [76]:

```
1  fig, ax = plt.subplots(figsize = (10,6))
2
3  ax.hist(X_final['total charge'], bins = 40, color = 'red', alpha = .5, lab
4  ax.hist(X_final[X_final['international plan'] == 1]['total charge'], bins
5
6  ax.set_xlabel("Monthly Bill")
7  ax.set_ylabel("Total Voicemail Users")
8  ax.set_title("Total User Base by Monthly Bill")
9  legend = ax.legend(loc='upper right')
10
11 plt.show()
```



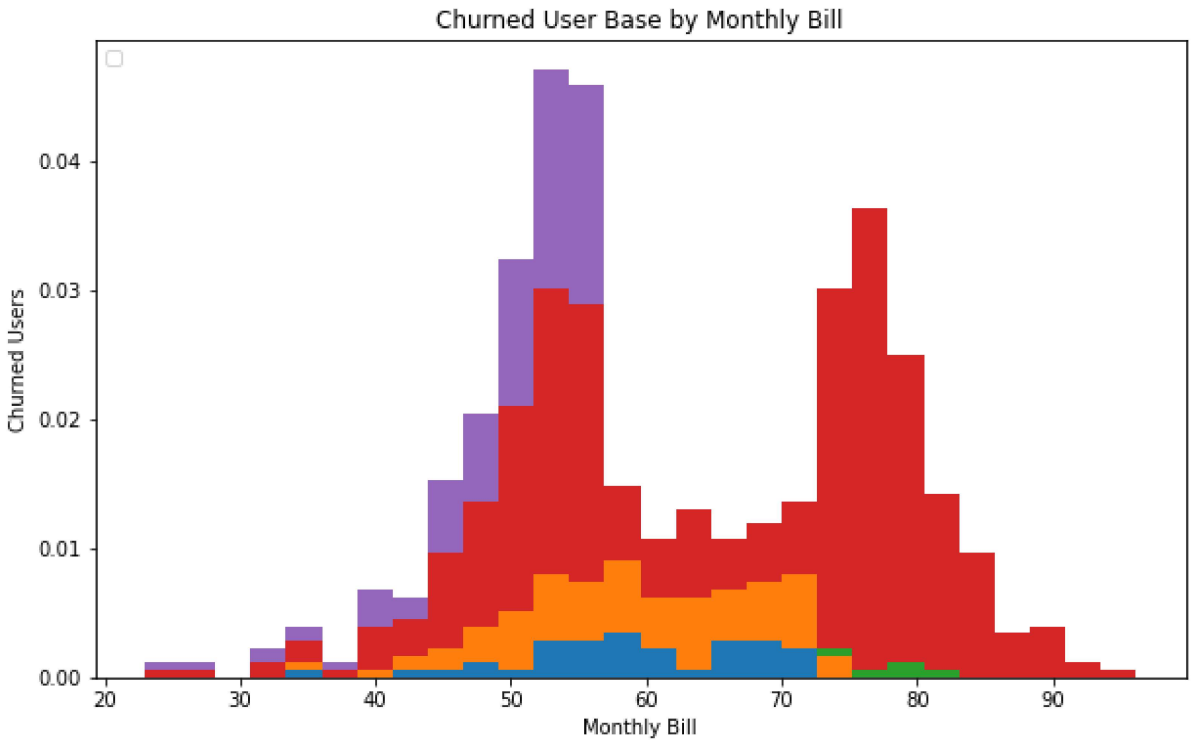
In [ ]:

```
1
```

In [77]:

```
fig, ax = plt.subplots(figsize = (10,6))

#ax.hist(churn['total charge'], bins = 20, color = 'blue', alpha = .5, labe
#ax.hist(high_spend['total charge'], bins = 8, color = 'red', alpha = .5,
#ax.hist(low_spend_customerservice['total charge'], color = 'yellow', alpha
#ax.hist(int_low_eng['total charge'], stacked = True, color = 'pink', alpha
#ax.vlines(x=74.04, ymin=.05, ymax=.95, colors='black', ls=':', lw=2, labe

ax.hist([group_4['total charge'], group_3['total charge'], group_5['total

ax.set_xlabel("Monthly Bill")
ax.set_ylabel("Churned Users")
ax.set_title("Churned User Base by Monthly Bill")
legend = ax.legend(loc='upper left')

plt.show()
```

```
No handles with labels found to put in legend.
```



In [ ]:

```
1
```