The Yhat Blog

machine learning, data science, engineering

Predicting customer churn with scikit-learn

by Eric Chiang | March 20, 2014

Customer Churn

"Churn Rate" is a business term describing the rate at which customers leave or cease paying for a product or service. It's a critical figure in many businesses, as it's often the case that acquiring *new* customers is a lot more costly than retaining *existing* ones (in some cases, 5 to 20 times more expensive).

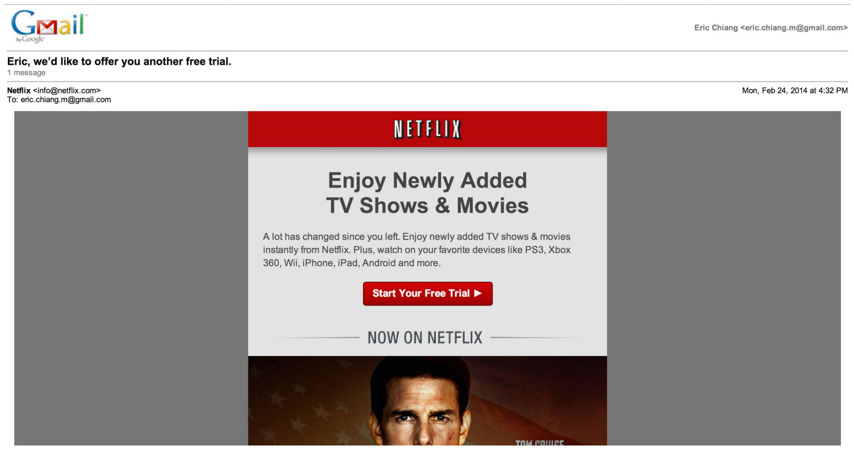
Understanding what keeps customers engaged, therefore, is incredibly valuable, as it is a logical foundation from which to develop retention strategies and roll out operational practices aimed to keep customers from walking out the door. Consequently, there's growing interest among companies to develop better churn-detection techniques, leading many to look to data mining and machine learning for new and creative approaches.

Predicting churn is particularly important for businesses w/ subscription models such as cell phone, cable, or merchant credit card processing plans. But modeling churn has wide reaching applications in many domains. For example, casinos have used predictive models to predict ideal room conditions for keeping patrons at the blackjack table and when to reward unlucky gamblers with front row seats to Celine Dion. Similarly, airlines may offer first class upgrades to complaining customers. The list goes on.

This is a post about modeling customer churn using Python.

Wait, don't go!

So what are some of ops strategies that companies e
reducing churn, it turns out, often requires non-trivia
retention teams are common in many industries anc
lists of at-risk customers to plead for their continued



Eric Chiang <eric.chiang.m@gmail.com>

Eric, we'd like to offer you another free trial.
1 message

Netflix <info@netflix.com>
To: eric.chiang.m@gmail.com

Mon, Feb 24, 2014 at 4:32 PM

NETFLIX

Enjoy Newly Added TV Shows & Movies

A lot has changed since you left. Enjoy newly added TV shows & movies instantly from Netflix. Plus, watch on your favorite devices like PS3, Xbox 360, Wii, iPhone, iPad, Android and more.

Organizing and running such teams is tough. From an ops perspective, cross-geographic teams must be well organized and trained to respond to a huge spectrum of customer complaints. Customers must be accurately targeted based on churn-risk, and retention treatments must be well-conceived and correspond reasonably to match expected customer value to ensure the economics make sense. Spending $1,000 on someone who wasn't about to leave can get expensive pretty quickly.

The good news is that we live in the data age and have some pretty great tools at our disposal to help answer these questions. John Forman of MailChimp summarizes this well:

> "I work in the e-mail marketing industry for a website called MailChimp.com. We help customers send e-mail newsletters to their audience, and every time someone uses the term 'e-mail blast,' a little part of me dies.
>
> "Why? Because e-mail addresses are no longer black boxes that you lob 'blasts' at like flash grenades. No, in e-mail marketing (as with many other forms of online engagement, including tweets, Facebook posts, and Pinterest campaigns), a business receives feedback on how their audience is engaging at the individual level through click tracking, online purchases, social sharing, and so on. This data is not noise. It characterizes your audience. But to the uninitiated, it might as well be Greek. Or Esperanto."
>
> \- John Foreman, *DataSmart*

Within this frame of mind, efficiently dealing with turnover is an exercise of distinguishing who is likely to churn from who is not using the data at our disposal. The remainder of this post will explore a simple case study to show how Python and its scientific libraries can be used to predict churn and how you might deploy such a solution within operations to guide a retention team.

The Dataset

The data set I'll be using is a longstanding telecom cu
download here.

The data is straightforward. Each row represents a s
customer. Each column contains customer attributes
minutes used during different times of day, charges
account duration, and whether or not the customer

In [2]:

```
from __future__ import division
import pandas as pd
import numpy as np

churn_df = pd.read_csv('churn.csv')
col_names = churn_df.columns.tolist()

print "Column names:"
print col_names

to_show = col_names[:6] + col_names[-6:]

print "\nSample data:"
churn_df[to_show].head(6)
```

```
        Column names:
        ['State', 'Account Length', 'Area Code', 'Phone', "Int'l Plan", 'VMail Plan',
'VMail Message', 'Day Mins', 'Day Calls', 'Day Charge', 'Eve Mins', 'Eve Calls', 'Eve Ch
arge', 'Night Mins', 'Night Calls', 'Night Charge', 'Intl Mins', 'Intl Calls', 'Intl Cha
rge', 'CustServ Calls', 'Churn?']

        Sample data:
```

Out[2]:

	State	Account Length	Area Code	Phone	Int'l Plan	VMail Plan	Night Charge	Intl Mins	Intl Calls	Intl Charge	CustServ Calls	Churn?
0	KS	128	415	382-4657	no	yes	11.01	10.0	3	2.70	1	False.
1	OH	107	415	371-7191	no	yes	11.45	13.7	3	3.70	1	False.
2	NJ	137	415	358-1921	no	no	7.32	12.2	5	3.29	0	False.
3	OH	84	408	375-9999	yes	no	8.86	6.6	7	1.78	2	False.
4	OK	75	415	330-6626	yes	no	8.41	10.1	3	2.73	3	False.
5	AL	118	510	391-8027	yes	no	9.18	6.3	6	1.70	0	False.

6 rows × 12 columns

I'll be keeping the statistical model pretty simple for this blog so the feature space is almost unchanged from what you see above. The following code simply drops irrelevant columns and converts strings to boolean values (since models don't handle "yes" and "no" very well). The rest of the numeric columns are left untouched.

In [3]:

```
# Isolate target data
churn_result = churn_df['Churn?']
y = np.where(churn_result == 'True.',1,0)

# We don't need these columns
to_drop = ['State','Area Code','Phone','Churn?']
churn_feat_space = churn_df.drop(to_drop,axis=1)

# 'yes'/'no' has to be converted to boolean values
# NumPy converts these from boolean to 1. and 0. late
yes_no_cols = ["Int'l Plan","VMail Plan"]
```

```
churn_feat_space[yes_no_cols] = churn_feat_space[yes_no_cols] == 'yes'

# Pull out features for future use
features = churn_feat_space.columns

X = churn_feat_space.as_matrix().astype(np.float)

# This is important
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(X)

print "Feature space holds %d observations and %d features" % X.shape
print "Unique target labels:", np.unique(y)
```

```
Feature space holds 3333 observations and 17 features
Unique target labels: [0 1]
```

One slight side note. Many predictors care about the relative size of different features even though those scales might be arbitrary. For instance: the number of points a basketball team scores per game will naturally be a couple orders of magnitude larger than their win percentage. But this doesn't mean that the latter is 100 times less significant. `StandardScaler` fixes this by normalizing each feature to a range of around 1.0 to -1.0 thereby preventing models from misbehaving. Well, at least for that reason.

Great, I now have a feature space '`X`' and a set of target values '`y`'. On to the predictions!

How good is your model?

Express, test, cycle. A machine learning pipeline should be anything but static. There are always new features to design, new data to use, new classifiers to consider each with unique parameters to tune. And for every change it's critical to be able to ask, *"Is the new version better than the last?"* So how do I do that?

As a good start, cross validation will be used throughout this blog. Cross validation attempts to avoid overfitting (training on and predicting the same datapoint) while still producing a prediction for each observation dataset. This is accomplished by systematically hiding different subsets of the data while training a set of models. After training, each model predicts on the subset that had been hidden to it, emulating multiple train-test splits. When done correctly, every observation will have a 'fair' corresponding prediction.

Here's what that looks like using `scikit-learn` libraries.

In [4]:

```
from sklearn.cross_validation import KFold

def run_cv(X,y,clf_class,**kwargs):
    # Construct a kfolds object
    kf = KFold(len(y),n_folds=5,shuffle=True)
    y_pred = y.copy()

    # Iterate through folds
    for train_index, test_index in kf:
        X_train, X_test = X[train_index], X[test_index]
        y_train = y[train_index]
        # Initialize a classifier with key word arguments
        clf = clf_class(**kwargs)
        clf.fit(X_train,y_train)
        y_pred[test_index] = clf.predict(X_test)
    return y_pred
```

I've decided to compare three fairly unique algorithms support vector machines, random forest, and k-nearest-neighbors. Nothing fancy here, just passing each to cross validation and determining how often the classifier predicted the correct class.

In [5]:

```
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RF
from sklearn.neighbors import KNeighborsClassifier as KNN

def accuracy(y_true,y_pred):
    # NumPy interprets True and False as 1. and 0.
    return np.mean(y_true == y_pred)

print "Support vector machines:"
print "%.3f" % accuracy(y, run_cv(X,y,SVC))
print "Random forest:"
print "%.3f" % accuracy(y, run_cv(X,y,RF))
print "K-nearest-neighbors:"
print "%.3f" % accuracy(y, run_cv(X,y,KNN))
```

```
Support vector machines:
0.918
Random forest:
0.943
K-nearest-neighbors:
0.896
```

Random forest won, right?

Precision and recall

Measurements aren't golden formulas which always spit out high numbers for good models and low numbers for bad ones. Inherently they convey something sentiment about a model's performance, and it's the job of the human designer to determine each number's validity. The problem with accuracy is that outcomes aren't necessarily equal. If my classifier predicted a customer would churn and they didn't, that's not the best but it's forgivable. However, if my classifier predicted a customer would return, I didn't act, and then they churned... that's really bad.

I'll be using another built in `scikit-learn` function to matrix. A confusion matrix is a way of visualizing pre and is just a table showing the distribution of predict axis indicates the true class of each observation (if a while the y-axis corresponds to the class predicted b said a customer would churned or not).

More great reads, straight to your inbox. Subscribe to the Yhat blog to get emailed when we publish a new post!

Email Address

SUBSCRIBE

In [6]:

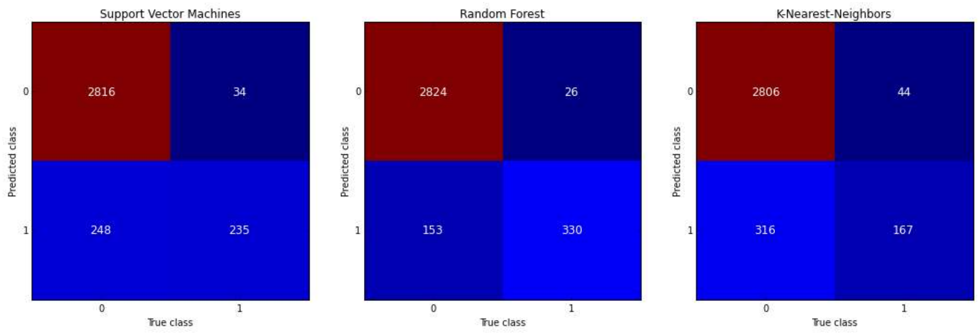
```
from sklearn.metrics import confusion_matrix

y = np.array(y)
class_names = np.unique(y)

confusion_matrices = [
    ( "Support Vector Machines", confusion_matrix(y,run_cv(X,y,SVC)) ),
    ( "Random Forest", confusion_matrix(y,run_cv(X,y,RF)) ),
    ( "K-Nearest-Neighbors", confusion_matrix(y,run_cv(X,y,KNN)) ),
]

# Pyplot code not included to reduce clutter
from churn_display import draw_confusion_matrices
%matplotlib inline

draw_confusion_matrices(confusion_matrices,class_names)
```



An important question to ask might be, *When an individual churns, how often does my classifier predict that correctly?* This measurement is called "recall" and a quick look at these diagrams can demonstrate that random forest is clearly best for this criteria. Out of all the churn cases (outcome " 1 ") random forest correctly retrieved 330 out of 482. This translates to a churn "recall" of about 68% ($330/482 \approx 2/3$), far better than support vector machines ($\approx 50\%$) or k-nearest-neighbors ($\approx 35\%$).

Another question of importance is "precision" or, *When a classifier predicts an individual will churn, how often does that individual actually churn?* The differences in semantic are small from the previous question, but it makes quite a different. Random forest again out preforms the other two at about 93% precision (330 out of 356) with support vector machines a little behind at about 87% (235 out of 269). K-nearest-neighbors lags at about 80%.

While, just like accuracy, precision and recall still rank random forest above SVC and KNN, this won't always be true. When different measurements do return a different pecking order, understanding the values and trade-offs of each rating should effect how you proceed.

Thinking in Probabilities

Decision making often favors probability over simple classifications. There's plainly more information in statements like *"there's a 20% chance of rain tomorrow"* and *"about 55% of test takers pass the California bar exa* *shouldn't rain tomorrow"* or *"you'll probably pass."* F churn also allow us to gauge a customers expected v Who do you want to reach out to first, the client with $20,000 annually, or the client who's worth $100,00C much should you spend on each client?

While I'm moving a bit away from my expertise, bein requires producing predictions a little differently. Ho moving to probabilities easy; my three models have `predict_proba()` built right into their class objects. This is the same cross validation code with only a few lines changed.

In [7]:

```
def run_prob_cv(X, y, clf_class, **kwargs):
    kf = KFold(len(y), n_folds=5, shuffle=True)
    y_prob = np.zeros((len(y),2))
    for train_index, test_index in kf:
        X_train, X_test = X[train_index], X[test_index]
        y_train = y[train_index]
        clf = clf_class(**kwargs)
        clf.fit(X_train,y_train)
        # Predict probabilities, not classes
        y_prob[test_index] = clf.predict_proba(X_test)
    return y_prob
```

How good is good?

Determining how good a predictor which gives *probabilities* rather than *classes* is a bit more difficult. If I predict there's a 20% likelihood of rain tomorrow I don't get to live out all the possible outcomes of the universe. It either rains or it doesn't.

What helps is that the predictors aren't making one prediction, they're making 3000+. So for every time I predict an event to occur 20% of the time I can see how often those events actually happen. Here's I use `pandas` to help me compare the predictions made by random forest against the actual outcomes.

In [8]:

```
import warnings
warnings.filterwarnings('ignore')

# Use 10 estimators so predictions are all multiples of 0.1
pred_prob = run_prob_cv(X, y, RF, n_estimators=10)
pred_churn = pred_prob[:,1]
is_churn = y == 1

# Number of times a predicted probability is assigned to an observation
counts = pd.value_counts(pred_churn)

# calculate true probabilities
true_prob = {}
for prob in counts.index:
    true_prob[prob] = np.mean(is_churn[pred_churn == prob])
    true_prob = pd.Series(true_prob)

# pandas-fu
counts = pd.concat([counts,true_prob], axis=1).reset_
counts.columns = ['pred_prob', 'count', 'true_prob']
counts
```

Out[8]:

	pred_prob	
0	0.0	1765
1	0.1	693
2	0.2	269
3	0.3	123
4	0.4	77

	pred_prob	count	true_prob
5	0.5	54	0.518519
6	0.6	73	0.835616
7	0.7	76	0.855263
8	0.8	70	0.957143
9	0.9	75	0.973333
10	1.0	58	1.000000

11 rows × 3 columns

We can see that random forests predicted that 75 individuals would have a 0.9 probability of churn and in actuality that group had a ~0.97 rate.

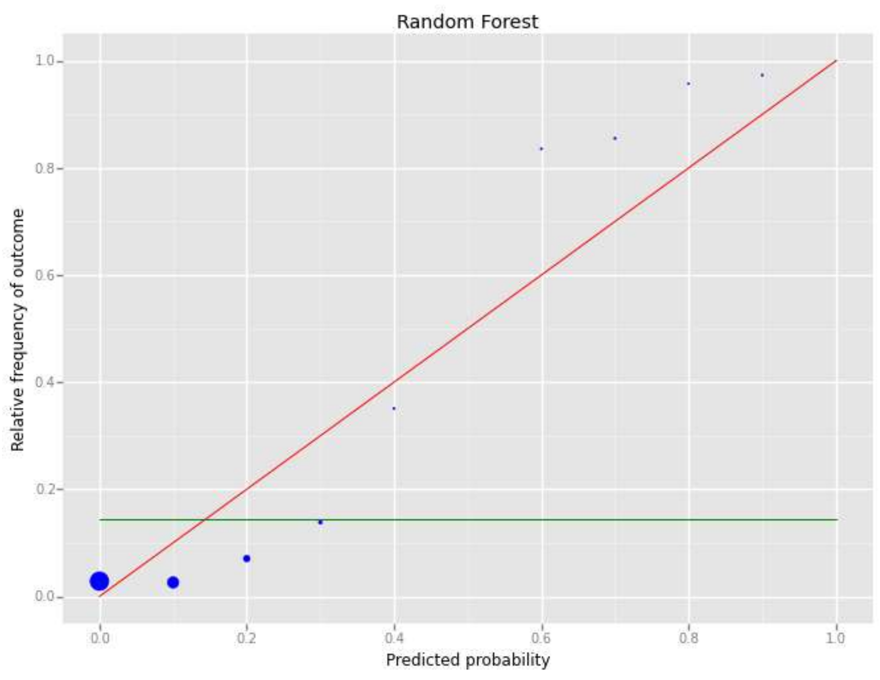
Calibration and Discrimination

Using the `DataFrame` above I can draw a pretty simple graph to help visualize probability measurements. The x axis represents the churn probabilities which random forest assigned to a group of individuals. The y axis is the actual rate of churn within that group, and each point is scaled relative to the size of the group.

In [9]:

```
from ggplot import *
%matplotlib inline

baseline = np.mean(is_churn)
ggplot(counts,aes(x='pred_prob',y='true_prob',size='count')) + \
    geom_point(color='blue') + \
    stat_function(fun = lambda x: x, color='red') + \
    stat_function(fun = lambda x: baseline, color='green') + \
    xlim(-0.05,  1.05) + ylim(-0.05,1.05) + \
    ggtitle("Random Forest") + \
    xlab("Predicted probability") + ylab("Relative frequency of outcome")
```



You may have also noticed the two lines I drew with

The red line represents a perfect prediction for a giv
probability forecasted equals the outcome frequenc
baseline probability of churn. For this dataset it's abo

Calibration is a relatively simple measurement and c
Events predicted to happen 60% of the time should
individuals I predict to have a churn risk of between

rate for that group should be about 35%. For the graph above think of it as, *How close are my predictions to the red line?*

Discrimination measures *How far are my predictions away from the green line?* Why is that important?

Well, if I assign a churn probability of 15% to every individual I'll have near perfect calibration due to averages, but I'll be lacking any real insight. Discrimination gives a model a better score if it's able to isolate groups which are further from the base set.

`Scikit-learn` doesn't come with these measurements, meaning I've had to implement them myself. For everyone's sake I've kept the math and source code out of this blog. Equations are replicated from Yang, Yates, and Smith (1991) and the code I wrote in the `churn_measurements` import below can be found on GitHub here.

In [10]:

```
from churn_measurements import calibration, discrimination
```

Let's see how my three models fair on these measurements.

In [11]:

```
def print_measurements(pred_prob):
    churn_prob, is_churn = pred_prob[:,1], y == 1
    print "  %-20s %.4f" % ("Calibration Error", calibration(churn_prob, is_churn))
    print "  %-20s %.4f" % ("Discrimination", discrimination(churn_prob,is_churn))

    print "Note -- Lower calibration is better, higher discrimination is better"

    print "Support vector machines:"
    print_measurements(run_prob_cv(X,y,SVC,probability=True))

    print "Random forests:"
    print_measurements(run_prob_cv(X,y,RF,n_estimators=18))

    print "K-nearest-neighbors:"

print_measurements(run_prob_cv(X,y,KNN))
```

```
Note -- Lower calibration is better, higher discrimination is better
Support vector machines:
Calibration Error    0.0017
Discrimination       0.0667
Random forests:
Calibration Error    0.0079
Discrimination       0.0830
K-nearest-neighbors:
Calibration Error    0.0022
Discrimination       0.0449
```

Unlike the classification comparisons earlier, randon
front-runner here. While it's good at differentiating b
probability churn events, it has trouble assigning an
to those events. For example the group which rando
30% churn rate actually had a true churn rate of 14%
be done, but I leave that to you as a challenge.

More great reads, straight to your inbox. Subscribe to the Yhat blog to get emailed when we publish a new post!

Email Address

SUBSCRIBE

Putting the model to use with Yhat

Time to upload a model to the cloud! In order to show some cool functionality, I'm going to go ahead and create a test set from the original churn data using `test_train_split()` from sklearn. From there, I fit a support vector classifier for my deployment.

In [12]:

```
from sklearn.cross_validation import train_test_split

train_index,test_index = train_test_split(churn_df.index)
clf = SVC(probability=True)
clf.fit(X[train_index],y[train_index])

test_churn_df = churn_df.ix[test_index]
test_churn_df.to_csv("test_churn.csv")
```

The model I'm going to deploy using `yhat` replicates the pipeline of this blog with a few modifications. Because I already defined variables within my global scope such as `yes_no_cols`, `features`, and `scaler` I can just use them without having to specify them further.

On the methodological side I've added a few calculations. First the customer worth has been added (the sum of the total charges to that individual). Combining this value with probability of churn creates a very important measurement: the expected loss of revenue from that customer. This is where an accurate prediction model plays an important role as it's impossible to produce these values with only classifications.

In [13]:

```
from yhat import Yhat,YhatModel,preprocess

class ChurnModel(YhatModel):
    # Type casts incoming data as a dataframe
    @preprocess(in_type=pd.DataFrame,out_type=pd.DataFrame)
    def execute(self,data):
        # Collect customer meta data
        response = data[['Area Code','Phone']]
        charges = ['Day Charge','Eve Charge','Night Charge','Intl Charge']
        response['customer_worth'] = data[charges].sum(axis=1)
        # Convert yes no columns to bool
        data[yes_no_cols] = data[yes_no_cols] == 'yes'
        # Create feature space
        X = data[features].as_matrix().astype(float)
        X = scaler.transform(X)
        # Make prediction
        churn_prob = clf.predict_proba(X)
        response['churn_prob'] = churn_prob[:,1]
        # Calculate expected loss by churn
        response['expected_loss'] = response['churn_pr
h']
        response = response.sort('expected_loss', asc
        # Return response DataFrame
        return response

yh = Yhat(
    "e[at]yhathq.com",
    "MY API KEY",
    "http://cloud.yhathq.com/"
)

response = yh.deploy("PythonChurnModel",ChurnModel,globals())
```

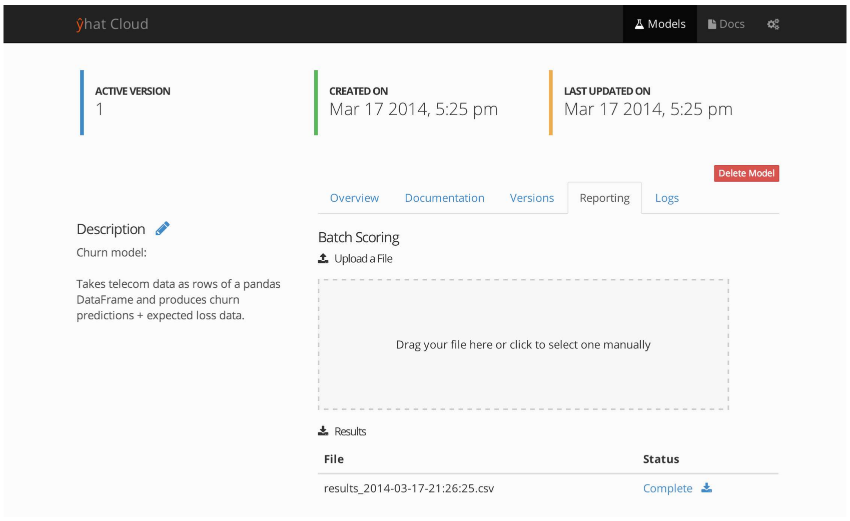
```
Are you sure you want to deploy? (y/N): y
```

Yhat batch mode

There comes a point when data science tools need to stop being scripts on an EC2 instance and start solving problems. In this case, empowering a retention team by warning them about customers likely to churn.

Our *M.O.* here at Yhat is finding ways to make data science applicable and practical as quickly as possible. Because my `execute()` function takes and returns a `DataFrame`, Yhat allows me to invoke my routine through a batch-scoring mode. The concept is simple. Upload a csv file from anywhere, the model pipeline is executed and the user gets a csv file back. This means that my method for scoring customers for churn-risk can be utilized by anyone else at my company regardless of their technical know-how, understanding of machine learning, or technical dependencies like `Python` or `R`.

Logging into cloud.yhathq.com and selecting my model gives me the following screen. The csv file I uploaded was the training data I created in the last section--in practice, this would be a new file exported from our CRM or customer database in the same format as the one I used to train the model.



ACTIVE VERSION
1

CREATED ON
Mar 17 2014, 5:25 pm

LAST UPDATED ON
Mar 17 2014, 5:25 pm

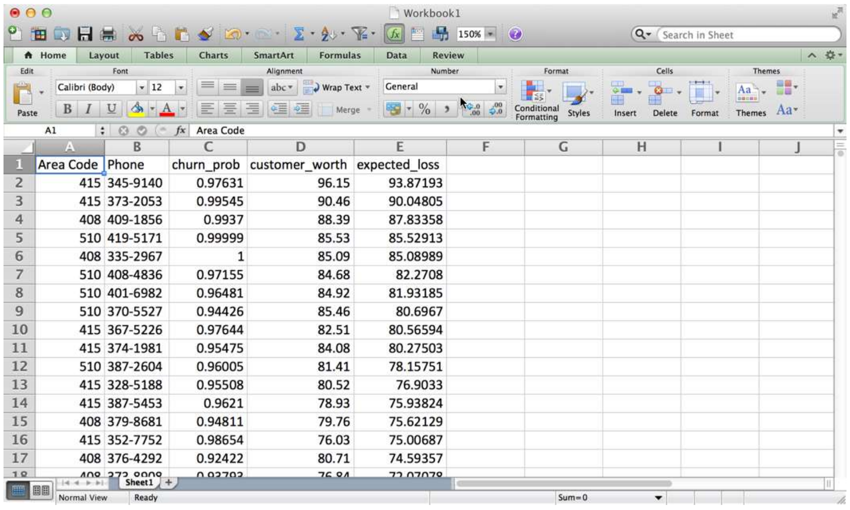
Description
Churn model:

Takes telecom data as rows of a pandas DataFrame and produces churn predictions + expected loss data.

Batch Scoring

File	Status
results_2014-03-17-21:26:25.csv	Complete

Wait a second, download the resulting file at the bottom of the page when it's ready, and open it in Excel.

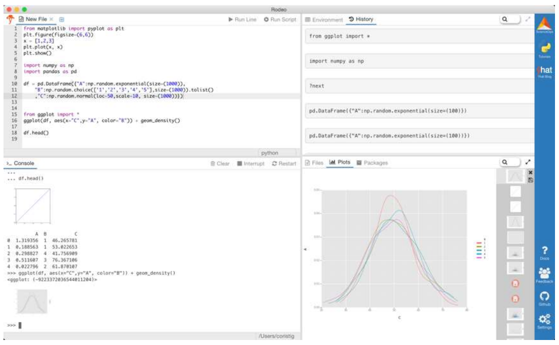


	Area Code	Phone	churn_prob	customer_worth	expected_loss
2	415	345-9140	0.97631	96.15	93.87193
3	415	373-2053	0.99545	90.46	90.04805
4	408	409-1856	0.9937	88.39	87.83358
5	510	419-5171	0.99999	85.53	85.52913
6	408	335-2967	1	85.09	85.08989
7	510	408-4836	0.97155	84.68	82.2708
8	510	401-6982	0.96481	84.92	81.93185
9	510	370-5527	0.94426	85.46	80.6967
10	415	367-5226	0.97644	82.51	80.56594
11	415	374-1981	0.95475	84.08	80.27503
12	510	387-2604	0.96005	81.41	78.15751
13	415	328-5188	0.95508	80.52	76.9033
14	415	387-5453	0.9621	78.93	75.93824
15	408	379-8681	0.94811	79.76	75.62129
16	415	352-7752	0.98654	76.03	75.00687
17	408	376-4292	0.92422	80.71	74.59357
18	408	373-8009	0.92703	76.84	72.07078

And there you go. Over 800 customers ranked and analyzed via drag and drop.

Learn More Facebook Twitter Linkedin Reddit

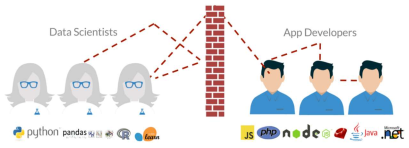
Our Products



Rodeo: a native Python editor built for doing data science on your desktop.

DOWNLOAD IT NOW!

scienceOPS



ScienceOps: deploy predictive models in production applications without IT.

LEARN MORE

More great reads, straight to your inbox. Subscribe to the Yhat blog to get emailed when we publish a new post!

Email Address

SUBSCRIBE

ŷhat

Yhat (pronounced Y-hat) provides data science solutions that let data scie
applications without IT or custom coding.