Week #7 - Python Practice

Indicating and Describing “Heavy” XYZ Customers

In our last Practice we worked with text data to learn a bit about regular expressions and processing text data with pandas. We're now going to turn back to the XYZ Company's data to do some data modification, recoding, and description.

Crank up a Canopy IPython session, and import pandas (as pd), numpy (np), DataFrame and Series from pandas, and possibly SQLAlchemy. “Possibly,” depending on whether you saved the XYZ customer data in that SQLite3 RDB xyz.db. You'll need the data that was in xyzcust10, so wherever and however you stored them, you need them back in Canopy for this session. In what follows we're going to read the data from the SQLite3 RDB.

Now, here's why you need the data. The XYZ marketing overlords want to learn more about who their good customers are. From their perspective, a “good” customer is one that is buying a lot from XYZ. Not surprising, eh? To start with XYZ marketing wants you to explore the data that was in the xyzcust data file. This is the data that in Session 2 you read in from the file xyzcust10.xyz, and that you subsequently wrote out to the SQLite3 RDB xyz.rb as a table. (You did, right?) XYZ marketing wants you to identify XYZ's current and past “heavy buyers,” and to describe where they came from, i.e. through what channel they initially purchased from XYZ. Based on your results, XYZ may subsequently request some more fine-grained analyses of its “heavies.”

I'm going to connect to xyz.rb, which I've put in my current working directory. I'll then look to see what the tables are in it.

In [1]:
```
import os

import cPickle as pickle

import pandas as pd  # panda's nickname is pd

import numpy as np  # numpy as np

import sqlalchemy

from pandas import DataFrame, Series, Categorical     # for convenience
from sqlalchemy import create_engine
from sqlalchemy import schema
from sqlalchemy import inspect  # helps us look at the tables in xyz.db
```

In [2]:
```
engine=create_engine('sqlite:///xyz.db')        # the db is in my current
```

In [3]:
```
insp=inspect(engine)        # get data about xyz that can be examined.
```

In [4]:
```
insp.get_table_names()
```

Out[4]:
```
[u'xyzcust', u'xyztrans']
```

This last statement should return the names of the tables in xyz.db. Let's use pandas to read the xyzcust table into a DataFrame called xyzcust:

How many customer records are in xyzcust?

There are a couple of variables in xyzcust that could be used to define "heavy buyers." They include:

- LTD_SALES and LTD_TRANSACTIONS: life-to-date sales in dollars and purchase transactions, respectively;
- YTD_SALES_2009 and YTD_TRANSACTIONS_2009: year-to-date sales and tranactions, respectively.

The "LTD" variables aren't that useful since we don't know how long each customer has been a customer. That leaves the YTD measures. They at least refer to the same time period across all customers. Did you know this is 2009? Well, it is here.

"Heavy" is in the eye of the beholder, and we need to define what it means, here. Let's take a look at the distribution of the data on the YTD variables.

In [5]:
```
xyzcust=pd.read_sql_table('xyzcust',engine)
xyzcust[['YTD_SALES_2009','YTD_TRANSACTIONS_2009']].describe()
```

Out[5]:

	YTD_SALES_2009	YTD_TRANSACTIONS_2009
count	30179.000000	30179.000000
mean	236.283972	0.844760
std	2117.042293	1.475401
min	0.000000	0.000000
25%	0.000000	0.000000
50%	0.000000	0.000000
75%	207.000000	1.000000
max	351000.000000	36.000000

which gives some summary statistics. Hm. They make sense for the sales measure, but maybe not so much for the transactions, since this measure is a count measure. pandas is treating it simply as an integer, so taking its average, standard deviation, etc., is somewhat sensible. To get a better feel for this variable's data, we can look at the decile values of its distribution:

In [6]:
```
pct=[x/10. for x in range(1,10)]        # a simple list comprehension
```

In [7]:
```
xyzcust[['YTD_SALES_2009','YTD_TRANSACTIONS_2009']].describe(percentiles=
```

Out[7]:

	YTD_SALES_2009	YTD_TRANSACTIONS_2009
count	30179.000000	30179.000000
mean	236.283972	0.844760
std	2117.042293	1.475401
min	0.000000	0.000000
10%	0.000000	0.000000
20%	0.000000	0.000000
30%	0.000000	0.000000
40%	0.000000	0.000000
50%	0.000000	0.000000
60%	57.000000	1.000000
70%	144.000000	1.000000
80%	288.000000	1.000000
90%	609.000000	2.000000
max	351000.000000	36.000000

You'll note a couple of things from your result. One is that half of the customers had have no recorded transactions. And, the 90% percentile is just two transactions. There is a customer with 36 transactions, which is a lot compared to customers on the whole. If 36 transactions is the maximum value, what percentile would it correspond to it? It appears that the transaction data distribution has a positive skew. That is, it has a long tail to the right. The same is true of the sales data. This is often the case for data that are left truncated. In this case, both are left truncated at zero. Neither can have negative values, at least in principle.

XYZ has somewhat arbitrarily classified its customers into three BUYER_STATUS groups, defined as follows:

- Active - A customer with a purchase within the last 12 months as of Dec. 31, 2009
- Lapsed - A customer with a purchase 13-24 months ago as of Dec. 31, 2009
- Inactive - A customer with a purchase >24 months ago as of Dec. 31, 2009

How many of each kind of customers are there?

In [8]:
```
# xyzcust.BUYER_STATUS.<tab>    # pick the method you need.

xyzcust.BUYER_STATUS.describe   # pick the method you need.
```

Out[8]:
```
<bound method Series.describe of 0        INACTIVE
1          ACTIVE
2          ACTIVE
3        INACTIVE
4          ACTIVE
5          ACTIVE
6          ACTIVE
7          ACTIVE
8        INACTIVE
9        INACTIVE
10         ACTIVE
11         LAPSED
12         ACTIVE
13       INACTIVE
14         ACTIVE
15       INACTIVE
16         LAPSED
17         ACTIVE
18         ACTIVE
19         ACTIVE
20       INACTIVE
21         ACTIVE
22       INACTIVE
23       INACTIVE
24         ACTIVE
25         ACTIVE
26         LAPSED
27         ACTIVE
28         ACTIVE
29         LAPSED
           ...
30149      ACTIVE
30150      ACTIVE
30151      ACTIVE
30152      ACTIVE
30153      ACTIVE
30154    INACTIVE
30155      LAPSED
30156      LAPSED
30157      ACTIVE
30158      LAPSED
30159    INACTIVE
```

```
30160      LAPSED
30161      LAPSED
30162      ACTIVE
30163      ACTIVE
30164    INACTIVE
30165      ACTIVE
30166      ACTIVE
30167      ACTIVE
30168    INACTIVE
30169      ACTIVE
30170      ACTIVE
30171    INACTIVE
30172    INACTIVE
30173      LAPSED
30174      ACTIVE
30175      ACTIVE
30176    INACTIVE
30177    INACTIVE
30178      ACTIVE
Name: BUYER_STATUS, dtype: object>
```

pandas has automatically typed this variable by default as a type "object," but it could be considered to be a categorical variable. We can make it categorical. By doing so it acquires a different set of attributes and methods:

In [9]:
```
#xyzcust.BUYER_STATUS.count

buyer_status = Categorical(xyzcust['BUYER_STATUS'])
```

In [10]:
```
buyer_status.describe()
```

Out[10]:

	counts	freqs
categories		
ACTIVE	13332	0.441764
INACTIVE	8996	0.298088
LAPSED	7851	0.260148

If you do:

In [11]:
```
buyer_status.unique
```

Out[11]:
```
<bound method Categorical.unique of [INACTIVE, ACTIVE, ACTIVE, INACTIV
E, ACTIVE, ..., ACTIVE, ACTIVE, INACTIVE, INACTIVE, ACTIVE]
Length: 30179
Categories (3, object): [ACTIVE, INACTIVE, LAPSED]>
```

you'll see that pandas has ordered the categories ACTIVE<INACTIVE<LAPSED. If you think that the order should be ACTIVE>INACTIVE>LAPSED instead (isn't ACTIVE “better than” INACTIVE?), you can reorder them:

In [12]:
```
buyer_status_ordered=buyer_status.reorder_categories(['LAPSED','INACTIVE'
```

And so:

In [13]:
```
buyer_status_ordered.describe()
```

Out[13]:

	counts	freqs
categories		
LAPSED	7851	0.260148
INACTIVE	8996	0.298088
ACTIVE	13332	0.441764

The rows above are now ordered “low category” to “high category.”

Now getting back to the defining heavy buyers, it's logical that XYZ's heavy buyers should be ACTIVE buyers, since XYZ thinks of them as being customers who are buying a lot, not who just may have bought a lot in the past. So let's look at just the ACTIVES' most recent purchasing:

In [14]:
```
xyzcust.YTD_SALES_2009[buyer_status_ordered=="ACTIVE"].describe(percentil
```

Out[14]:
```
count     13332.000000
mean        534.864536
std        3160.079031
min           3.000000
10%          60.000000
20%          93.000000
30%         132.000000
40%         180.000000
50%         246.000000
60%         333.000000
70%         468.000000
80%         684.000000
90%        1154.700000
max      351000.000000
Name: YTD_SALES_2009, dtype: float64
```

You can see from the percentiles that the data appear to have a positive skew, and that the largest value is more than $35,000. You can get a closer look at how most of the data are distributed in a histogram by ignoring values greater than $900:

In [15]:
```
xyzcust.YTD_SALES_2009[buyer_status_ordered=="ACTIVE"].hist(bins=30,range
```

Out[15]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1104f2c50>
```

This should display a figure box for you with a histogram with 30 vertical bars going from 0 to 900.

Hm. Are there any “actives” with no purchases this year? (In 2009, that is.)

In [16]:
```
xyzcust.YTD_SALES_2009[(xyzcust.BUYER_STATUS=="ACTIVE") & (xyzcust.YTD_SA
```

Out[16]:
```
0
```

Are there any?

So, assuming that we'll use YTD_SALES_2009 to define who the heavy buyers are, how much should a buyer need to spend to be a “heavy?” So now XYZ marketing steps in, and without recourse to data they decide they want to define the heavies as those who have spend more than at least 67% of all ACTIVEs in 2009. To create a variable that tells

which customers are heavy buyers, a “heavy indicator” that can be used for selecting and analyzing heavy customers, we need to know what 2009 purchase amount separates them from the rest of XYZ's customers. The heavies will have spent above that amount in 2009, and the rest (the 'lights?') will have spent less. That amount corresponds to the 67th percentile of the distribution of ACTIVE's 2009 purchase amounts. And the answer is:

In [17]:
```
xyzcust.YTD_SALES_2009[(buyer_status_ordered=="ACTIVE") ].quantile(0.67)
```

Out[17]: 423.0

Let's assign this value as a constant to:

In [18]:
```
heavyCut= 423  #heavyCut is a constant
```

Now let's create a heavy buyer indicator variable by using heavyCut. This can be done in more than one way. In what follows we first create a two category variable, and then we use it to create two, dummy coded indicator variables.

In [19]:
```
heavyCat=Categorical(np.where(xyzcust.YTD_SALES_2009>heavyCut,1,0))
```

heavyCat is a pandas Categorical object, an array of sorts on which values fall into one of a number of discrete categories, of which there are two in this case. Heavy buyers have a category of “1” on heavyCat, and the rest of the buyers have a “0.” Let's give those two categories more descriptive names.

In [20]:
```
heavyCat.describe()
```

Out[20]:

	counts	freqs
categories		
0	25795	0.854733
1	4384	0.145267

In [21]:
```
heavyCat.rename_categories(['regular','heavy'],inplace=True)
```

Note the somewhat risky inplace=True option, which means that heavyCat is modified “in place.” A generally safer alternative is to put this statement without the inplace option on the right hand side of an assignment to a new Categorical object. But if we messed up here, backing up a step or two wouldn't be all that painful as we still have

all the necessary ingredients to get to where we need to go. (As long as your session hasn't crashed, of course.)

In [22]:
```
heavyCat.describe()
```

Out[22]:

	counts	freqs
categories		
regular	25795	0.854733
heavy	4384	0.145267

Your results may be different than the above. Why might that be? Also, why do you think that the heavy customers might comprise only about 15% of the sample in this table? (See the “freqs” column which shows relative frequencies.)

If you just type heavyCat.describe() at the command prompt you should get a pretty table.

We named the “non-heavy” customers “regular” as we weren't sure how light they might be.

Now let's use heavyCat to make some dummy indicator variables.

In [23]:
```
buyerType=pd.get_dummies(heavyCat)
```

In [24]:
```
buyerType.head()
```

Out[24]:

	regular	heavy
0	1	0
1	0	1
2	1	0
3	1	0
4	1	0

buyerType is a DataFrame. How many columns does it have? What are their names?

We're almost done with this Practice, but before we wrap things up let's spend a moment or two considering those rather large values in YTD_SALES_2009 and YTD_TRANACTIONS_2009. These were a little surprising, right? Unexpected data values are often referred to as

surprising, right? Unexpected data values are often referred to as "outliers" in the sense that they are unusual, unexpected, and in some way very different from other data values. In the case of these variables, the customers with the largest values are:

In [25]:
```
xyzcust[(xyzcust.YTD_SALES_2009==max(xyzcust.YTD_SALES_2009))|(xyzcust.YT
```

Out[25]:

	index	ACCTNO	ZIP	ZIP4	LTD_SALES	LTD_TRANSACTIONS	YTD_SALES_2
1756	1773	PPHDLSPGP	60067	8002	33993.0	175	6429.0
1855	1872	LLSYGYDAY	60074	0	2247750.0	84	351000.0

You can separate the above into statements if all the brackets and parens in it are giving you grief. Or, you could use the logical statements to create an index to select rows from xyzcust.

Now, the question is, are the data values for these customers valid, or are they erroneous in some way? If you could assume that some statistical distribution should describe the data, then you could use it to identify "surprising' data points. The Normal, or Gaussian, distribution is often used for this purpose. Looking back at the histogram you plotted earlier, do you think the data are likely to be from a Normal distribution? Both variables are measures that are left-truncated at zero, meaning that they can't have negative values. YTD_TRANSACTIONS_2009 measures counts of purchase transactions, so it's not really a continuous variable. What shape or distribution would you say either variable's data should have? This is obviously a difficult question to answer, and especially so if you don't know about the circumstances from which the data arose.

How to deal with extreme values that might be "outliers" requires good judgment and some domain expertise. Many tactics used involve throwing out extreme data points or modifying them in some way so as to reduce their influence on statistical estimators. "Winsorizing" is a venerable method for dealing with outliers that's pretty much an ad hoc method, and can be appropriate for situations in which outliers are due to some kind of measurement error. It consists of collapsing the tails of a data distribution so that extreme values below or above specific percentile cutoffs, e.g. the values corresponding to 5% and 95%, are replaced with the values at those percentages. We can Winsorize YTD_SALES_2009 by collapsing the extremes of its data as follows:

First, we need the quantiles we'll use for trimming. Let's use 5% and 95%.

In [26]:
```
pct5, pct95 = xyzcust.YTD_SALES_2009[(buyer_status_ordered=="ACTIVE") ].q
```

In [27]:
```
pct5, pct95
```

Out[27]:
```
(42.0, 1798.3499999999967)
```

```
So the value at the 5th percentile, is 42, and at the 95th is about
1798.
```

In [28]:
```
winSales=xyzcust.YTD_SALES_2009     # just making a copy here.
```

In [29]:
```
winSales[winSales>pct95].head()
```

Out[29]:
```
12      2064.0
24      1875.0
77      1815.0
137    14448.0
228     5586.0
Name: YTD_SALES_2009, dtype: float64
```

In [30]:
```
len(winSales[winSales>pct95])
```

Out[30]:
```
667
```

In [31]:
```
winSales[winSales>pct95]=pct95          # collapsing the extreme high val
```

```
/Users/Zeeshan/Library/Enthought/Canopy_64bit/User/lib/python2.7/site-
packages/ipykernel/__main__.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
(http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-vi
ew-versus-copy)
  if __name__ == '__main__':
```

In [32]:
```
#Sanity Test
winSales[winSales==pct95].head()
```

Out[32]:
```
12     1798.35
24     1798.35
77     1798.35
137    1798.35
228    1798.35
Name: YTD_SALES_2009, dtype: float64
```

In [33]:
```
winSales[winSales<pct5].head()
```

Out[33]:
```
0     0.0
3     0.0
7    33.0
8     0.0
9     0.0
Name: YTD_SALES_2009, dtype: float64
```

In [34]:
```
winSales[winSales<pct5]=pct5 # smooshing up the low values
```

```
/Users/Zeeshan/Library/Enthought/Canopy_64bit/User/lib/python2.7/site-
packages/ipykernel/__main__.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
(http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-vi
ew-versus-copy)
  if __name__ == '__main__':
```

In [35]:
```
winSales[winSales==pct5].head()
```

Out[35]:
```
0    42.0
3    42.0
7    42.0
8    42.0
9    42.0
Name: YTD_SALES_2009, dtype: float64
```

That's all there is too it, although in this case Winsorizing is a questionable thing to do. Note that deleting or modifying data should never be done without giving it very careful consideration.

One last thing: let's crosstabulate buyerType and CHANNEL_ACQUISITION to see if there's any variation in Type across channels. But first let's add our heavyCat and buyerType variables as columns to our xyzcust DataFrame. We'll save this DataFrame for use in a later Practice.

You can add a column like this:

In [36]:
```
xyzcust['heavyCat']=heavyCat
```

In [37]:
```
# Sanity Test
xyzcust[['ACCTNO', 'YTD_SALES_2009', 'YTD_SALES_2009', 'heavyCat']].head(
```

Out[37]:

	ACCTNO	YTD_SALES_2009	YTD_SALES_2009	heavyCat
0	WDQQLLDQL	42.0	42.0	regular
1	WQWAYHYLA	1263.0	1263.0	heavy
2	GSHAPLHAW	129.0	129.0	regular
3	PGGYDYWAD	42.0	42.0	regular
4	LWPSGPLLS	72.0	72.0	regular

Now add the columns from buyerType to xyzcust.

Let's take a look at how CHANNEL_ACQUISITION AND heavyCat might be related:

In [38]:
```
# In case you get an error (it appears it is a bug in pandas.crosstab lat
#
# pd.crosstab(xyzcust.heavyCat,xyzcust.CHANNEL_ACQUISITION,margins=True)
#
# Try the following command instead

pd.crosstab(xyzcust.heavyCat.astype('string'),xyzcust.CHANNEL_ACQUISITION
```

Out[38]:

CHANNEL_ACQUISITION	CB	IB	RT	All
heavyCat				
heavy	356	703	3325	4384
regular	1506	3472	20817	25795
All	1862	4175	24142	30179

What you should get is a frequency table with row and column margin totals. It might be more informative with row, column, and cell percentages. We'll get to this sort of thing in an upcoming Practice.

By the way, if you had assigned the result of the above command to a name, what you'd have is your table in a DataFrame that you could use for additional analyses.

So for now, save your new version of xyzcust (the one that includes

heavyCat and the buyer type indicators) for future use. The format is your choice. You can save it to xyz.db, pickle it, or use some other method. It's up to you. Just be sure you'll be able to find it when you need it in a future Practice.

Deliverable : If you were to define the heavies as those who have spent more than at least 51% of all ACTIVEs in 2009, what will be the heavyCut value?

In [39]:
```
xyzcust.YTD_SALES_2009[(buyer_status_ordered=="ACTIVE") ].quantile(0.51)
```

Out[39]: 252.0

In [ ]: