Kiva Classifying Delinquent Loans

Belle Peng

May 8, 2014

Part I

Introduction

Kiva Microfunds (commonly known by its domain name, Kiva.org) is a non-profit organization that allows people to lend money via the Internet to low-income / underserved entrepreneurs and students in over 70 countries. Kiva's mission is "to connect people through lending to alleviate poverty. The Kiva platform has attracted a community of more than 1 million lenders from around the world. Kiva itself does not collect any interest on the loans it facilitates and Kiva lenders do not make interest on loans. Kiva is purely supported by grants, loans, and donations from its users, corporations, and national institutions. Kiva allows users to go onto their website, browse for a loan they want to lend to, make a loan as little as $25, get repaid and repeat! Since the mission of Kiva is to alleviate poverty, the success of the entrepreneurs is very important. Therefore, I am interested in researching whether there are loan characteristics that makes a loan more likely to be delinquent.

A delinquent loan is defined as a loan that is late on payment. This includes loans that are default (completely gave up on payment), and loans that are in repayment but are late. To find characteristics related to delinquent loans, I used classification methods to identify influential features that distinuish delinquent loans from non-delinquent loans.

Goal: Classify delinquent loans. Find influential characteristics of delinquent loans.

1 Data Source

I obtained the data from the Kiva Builder API: http://build.kiva.org/docs/data/loans. It contains 2769 .json files split into three folders: Loans, Lenders, and Loans_Lenders. The Loans folder contains data on 659,960 Loans; the Lenders folder contains data on 1,224,850 Lenders; and finally the Loans_Lenders folder contains 5,413,147 rows of data that connects the loans with the lenders data. Each loan can have multiple lenders, and each lender can loan to multiple loans as well. Based on my initial analysis, there are approximately 3.5% delinquent loans. Below, I will document the steps I took to perform the Data Extraction and Preparation, Exploratory Data Analysis, Modeling, and Conclusions.

Part II

Extraction and Data Prep

This Extraction and Data Prep step only needs to be run once, which generates a cleaned-up .csv file. The rest of my analysis involves reading in the output file from this section, which contains all of the Loans, Lenders, and

Loans_Lenders fields of my sampled loans. NOTE: due to the size and complexity of the data, running the code here will take a long time and most likely crash iPython Notebook. I ran the code on the cloud or SCF machine or on my personal laptop, and still required hours of run-time, possibly running out of disk memory (which will require the SCF guys' help). Therefore, if you wisely decide not to run the extraction code in this notebook and go straight to the analysis, I have attached my sampled data so you can use that directly. You will eventually need my sampled data anyways, since drawing a different sample may change the results slightly.

The Kiva data comes in three sets: Loans, Lenders, and Loans_Lenders (which is a look up file that connects the Loans data with the Lenders data). There are 1320 .json files in Loans, 1225 .json files in Lenders, and 224 .json files in Loans_Lenders. The purpose of this notebook is to extract *select* data useful for my analysis out of the .json files and combine the 1320 .json files into one single Loans data, so I have all of the data in one place. I looked at all of the available fields in various dictionaries, and selected only the ones of interest to pull out, as shown in the code. Similarly for Lenderes and Loans_Lenders. Included in the code below are also some basic data cleaning, export to .csv, then concatenate all of the .csv in unix.

Due to the complexity of the data, merging the Loans and Lenders data was the most challenging and time-consuming part of this project. The many-to-many joint between lenders and loans datasets creates a multi-dimensional dataframe that is difficult and time-consuming to work with. Due to the time and resources constraint, I chose to reduce the lenders data as summary statsitics rather than lender-level details. I did this using Pandas Panels Dataframe.

2 Extract Loans

Due to the size of the files, I had to split up the 1320 files into running 200 files at a time, for 7 times. So each time I un-comment out one of the "for i in range(200, 400):" lines, and one of the "loansFrame.to_csv(‘loans1.csv’, sep=‘’̑, encoding=‘utf-8’, na_rep=‘N/A’, header=False)" lines, leave the rest commented out. Once I finish running all 7, I go into Unix and concatenate.

```
In [1]: import os
        import json
        from pandas import DataFrame, Series
        import pandas as pd
        os.chdir('/Users/bellepeng/Documents/Stat222_Capstone')
        #os.chdir('/accounts/grad/belle.peng/Desktop/iPython Notebook')
        #you will need to reset your working directory here
```

```
In [ ]: #Get all the file names
        loansDir='kiva_ds_json/loans'

        loansTable = []
        jsonfiles = [fname for fname in os.listdir(loansDir) if fname[-5:]==".json"]
        for i in range(200): #1
        #for i in range(200, 400): #2
        #for i in range(400, 600): #3
        #for i in range(600, 800): #4
        #for i in range(800, 1000): #5
        #for i in range(1000, 1200): #6
        #for i in range(1200, 1320): #7
            with open(loansDir+'/'+jsonfiles[i], "r") as myfile:
                loansjson=json.loads(myfile.read())
                loansArr=loansjson['loans']

                #pull out fields from dictionary, append them to each loan
                for j in range(len(loansArr)):
                 if not loansArr[j]['payments']: dt='N/A'
                 else: dt=loansArr[j]['payments'][0]['settlement_date']
                 loansArr[j]['settlement_date']=dt

                 if not loansArr[j]['location']['town']: town='N/A'
```

```
         else: town=loansArr[j]['location']['town']
         loansArr[j]['town']=town

         if not loansArr[j]['location']['country']: country='N/A'
         else: country=loansArr[j]['location']['country']
         loansArr[j]['country']=country

         if not loansArr[j]['terms']['disbursal_amount']: disbursal_amount='N/A'
         else: disbursal_amount=loansArr[j]['terms']['disbursal_amount']
         loansArr[j]['disbursal_amount']=disbursal_amount

         if not loansArr[j]['terms']['disbursal_date']: disbursal_date='N/A'
         else: disbursal_date=loansArr[j]['terms']['disbursal_date']
         loansArr[j]['disbursal_date']=disbursal_date

         if not loansArr[j]['terms']['repayment_term']: repayment_term='N/A'
         else: repayment_term=loansArr[j]['terms']['repayment_term']
         loansArr[j]['repayment_term']=repayment_term

         if not loansArr[j]['terms']['loss_liability']['nonpayment']: loss_liability='
         else: loss_liability=loansArr[j]['terms']['loss_liability']['nonpayment']
         loansArr[j]['loss_liability']=loss_liability

        #Put the loans into master table
        loansTable.extend(loansArr)
        print jsonfiles[i]
loansFrame = DataFrame(loansTable)
```

In [ ]:

```
#delete some fields
fields=['basket_amount', 'borrowers', 'location', 'description', 'image', 'journal_tot
for i in fields:
    del loansFrame[i]
```

In [ ]:

```
#export
loansFrame.to_csv('loans1.csv', sep='\t', encoding='utf-8', na_rep='N/A', header=False
#loansFrame.to_csv('loans2.csv', sep='\t', encoding='utf-8', na_rep='N/A', header=Fals
#loansFrame.to_csv('loans3.csv', sep='\t', encoding='utf-8', na_rep='N/A', header=Fals
#loansFrame.to_csv('loans4.csv', sep='\t', encoding='utf-8', na_rep='N/A', header=Fals
#loansFrame.to_csv('loans5.csv', sep='\t', encoding='utf-8', na_rep='N/A', header=Fals
#loansFrame.to_csv('loans6.csv', sep='\t', encoding='utf-8', na_rep='N/A', header=Fals
#loansFrame.to_csv('loans7.csv', sep='\t', encoding='utf-8', na_rep='N/A', header=Fals
```

In [ ]:

```
#unix code to concatenate all the loans files
!cat loans1.csv loans2.csv loans3.csv loans4.csv loans5.csv loans6.csv loans7.csv > al
#Testing import
#loans = pd.read_table("allLoans.csv")
```

3 Extract Lenders

Similar to extracting Loans above, I extract the data and delete fields that I am not interested in feeding into my model.

In [ ]:

```
#Get all the file names
lendersDir='kiva_ds_json/lenders'

lendersTable = []
jsonfiles = [fname for fname in os.listdir(lendersDir) if fname[-5:]==".json"]
for i in range(len(jsonfiles)):
    with open(lendersDir+'/'+jsonfiles[i], "r") as myfile:
        lendersjson=json.loads(myfile.read())
        lendersArr=lendersjson['lenders']
        lendersTable.extend(lendersArr)
        print i

frame = DataFrame(lendersTable)

#delete some fields
fields=[’image’, ’loan_because’, ’name’, ’occupational_info’, ’personal_url’]
for i in fields:
    del frame[i]

#export
frame.to_csv(’AllLenders.csv’, sep=’\t’, encoding=’utf-8’, na_rep=’N/A’)
#import
#lenders = pd.read_table("allLenders.csv")
```

4 Extract Loans_Lenders

The file sizes are smaller here, so I can achieve the extraction in one step, no concatenating was necessary.

In [ ]:

```
os.chdir(’/Users/bellepeng/Documents/Stat222_Capstone’)
#os.chdir(’/accounts/grad/belle.peng/Desktop/Capstone’)

#Get all the file names
llDir=’kiva_ds_json/loans_lenders’

Table = {}
jsonfiles = [fname for fname in os.listdir(llDir) if fname[-5:]==".json"]
for i in range(len(jsonfiles)):
#for i in range(224):
    with open(llDir+’/’+jsonfiles[i], "r") as myfile:
        jsonfile=json.loads(myfile.read())
        Arr=jsonfile[’loans_lenders’]
        for j in range(len(Arr)):
         key = Arr[j][’id’]
         Table[key] = Arr[j][’lender_ids’]
        print i

#ll2 = DataFrame(Table)

#export
ll.to_csv(’allLL.csv’, sep=’\t’, encoding=’utf-8’, na_rep=’N/A’)
#import
#LL = pd.read_table("allLL.csv")
```

5 Data Prep

This section of code preps the data for further analysis.

1. Read in the loans, lenders, and loans_lenders data
2. Draw a simple random sample of 10000 loans
3. Select the corresponding loans_lenders for the 10000 loans
4. For each of the 10000 loans, look up the corresponding lenders data (each loan can have multiple lenders)
5. Summarize the corresponding lenders data from step 4. For example, if a loan has 5 lenders, I will find the average number of loans these lenders lend to (avg_loan_count), the average number of invitees of these 5 lenders (the avg_invitee_count), and the number of countries they came from (country_count).
6. Concatenate the lenders data and delete useless fields
7. Merge the summarized Lenders with Loans, and output

8. Convert to correct format, remove useless fields, export data

In [ ]:

```
import os
import json
from pandas import DataFrame, Series
import pandas as pd
import numpy as np
from datetime import datetime
from dateutil import parser
```

In [ ]:

```
allloans = pd.read_table("allLoans.csv")
lenders = pd.read_table("allLenders.csv")

# Get loans_lenders data
llDir=’kiva_ds_json/loans_lenders’
#llDir=’Capstone/kiva_ds_json/loans_lenders’
Table = {}
jsonfiles = [fname for fname in os.listdir(llDir) if fname[-5:]==".json"]
for i in range(len(jsonfiles)):
    with open(llDir+’/’+jsonfiles[i], "r") as myfile:
        jsonfile=json.loads(myfile.read())
        Arr=jsonfile[’loans_lenders’]
        for j in range(len(Arr)):
         key = Arr[j][’id’]
         Table[key] = Arr[j][’lender_ids’]
    print i
```

Here I am drawing a simple random sample of 10000 loans and merging to the corresponding lenders information using the “loans_lenders” file. Note that some loans may have multiple lenders.

In [ ]:

```
# Sample loans
import random
random.seed(999)
s=random.sample(allloans[’id’], 10000)
loans=allloans[allloans[’id’].isin(s)] #take the sample from loans
loans=loans.set_index(loans[’id’], drop =True)
```

In [ ]:

```
#remove nas 224000, removed 664, 223336 remained | 525674, 524288
len(Table)
for key in Table.keys():
if (type(Table[key])!=list): Table.pop(key)
```

In [ ]:

```
# Find the sampled loans in loans_lenders data (many loans don’t have lenders, some ha
sample={}
i=0
for key in s:
i=i+1
print i, key
if key in Table.keys():
sample[key]=Table[key]
```

This step merges the 10000 loans to lenders data.

In [ ]:

```
LendersPerLoan = {}
for key in sample.keys()[0:500]:
#for key in sample.keys()[500:1000]:
#for key in sample.keys()[1000:1500]:
#for key in sample.keys()[1500:2000]:
#for key in sample.keys()[2000:2500]:
#for key in sample.keys()[2500:3000]:
```

```
#for key in sample.keys()[3000:3500]:
#for key in sample.keys()[3500:len(sample)]:
lender_ids_for_loan = sample[key]
idx = lenders['lender_id'].isin(lender_ids_for_loan)
LendersPerLoan[key] = lenders[idx]
print key
```

This step turns it into Pandas Panels Data Frame, so that it's multi-dimensional, the purpose is to calculate summary statistics later.

In [ ]:

```
LendersPerLoan = pd.Panel(LendersPerLoan)
```

For each loan, there may be multiple lenders. I am calculating some summary statistics of the Lenders for each Loan, such as average number of loans these lenders lend to, average number of invitees, and the number of countries the lenders came from. Using these summary statistics, I then merge it back to the Loans data, so I have a 2-dimensional dataframe, which contains both loans and lenders information, then export to csv.

In [ ]:

```
# for-loop for average loan count and other counts # 462308
avg_loan_count = []
avg_invitee_count = []
country_count=[]
for k in range(len(LendersPerLoan.keys())):
key = LendersPerLoan.keys()[k]
avg_loan_count.append(LendersPerLoan[key]['loan_count'].mean())
avg_invitee_count.append(LendersPerLoan[key]['invitee_count'].mean())
country_count.append(pd.Series.nunique(LendersPerLoan[key]['country_code']))
```

In [ ]:

```
# combine my lender stats into a dataframe
lenDF4000 = {'avg_loan_count':avg_loan_count, 'avg_invitee_count':avg_invitee_count,
'country_count': country_count, 'id':LendersPerLoan.keys()}
lenDF4000 = DataFrame(lenDF4000, index=LendersPerLoan.keys())
lenDF4000.to_csv('lenDF4000.csv', sep='\t', encoding='utf-8', na_rep='N/A')
#lenDF500 = pd.read_table("lenDF500.csv")
```

In [ ]:

```
# Concatenate and output
pieces=[lenDF500, lenDF1000, lenDF1500, lenDF2000, lenDF2500, lenDF3000, lenDF3500, le
lenderSum=pd.concat(pieces)
```

In [ ]:

```
delete=['Unnamed: 0', 'Unnamed: 0.1']
for i in delete:
    del lenderSum[i]
lenderSum.to_csv('lenderSumHead.csv', sep='\t', encoding='utf-8', na_rep='N/A', header

lenderSum= pd.read_table("lenderSumHead.csv")
```

In [ ]:

```
# Merge to loans dataset
lenderSum['id']=lenderSum['id'].convert_objects(convert_numeric=True)
mergedDF=pd.merge(left=loans, right=lenderSum, on='id', how='left', sort=True)
mergedDF.to_csv('sample10000.csv', sep='\t', encoding='utf-8', na_rep='N/A', header=Tr
#df=pd.read_table('sample10000.csv')
```

Below I am doing some data cleaning and converting to the correct format, then finally export my data for analysis, called "sample10000.csv", that way I can just import that file each time for analysis without having to run all of code in this notebook.

In [ ]:

```
### Convert to correct format
# Booleans
df[’bonus_credit_eligibility’].value_counts()
df[’bonus_credit_eligibility’][df[’bonus_credit_eligibility’]==’True’]=True
df[’delinquent’].value_counts()
df[’delinquent’][df[’delinquent’]==’N/A’]=False
df[’delinquent’][df[’delinquent’]==’True’]=True
df[’delinquent’].astype(bool)
```

In [ ]:

```
# Continuous
df[’paid_amount’]=df[’paid_amount’].convert_objects(convert_numeric=True)
df[’currency_exchange_loss_amount’]=df[’currency_exchange_loss_amount’].convert_object
df[’disbursal_amount’]=df[’disbursal_amount’].convert_objects(convert_numeric=True)
df[’funded_amount’]=df[’funded_amount’].convert_objects(convert_numeric=True)
df[’loan_amount’]=df[’loan_amount’].convert_objects(convert_numeric=True)
df[’lender_count’]=df[’lender_count’].convert_objects(convert_numeric=True)
df[’repayment_term’]=df[’repayment_term’].convert_objects(convert_numeric=True)
df[’avg_invitee_count’]=df[’avg_invitee_count’].convert_objects(convert_numeric=True)
df[’avg_loan_count’]=df[’avg_loan_count’].convert_objects(convert_numeric=True)
df[’country_count’]=df[’country_count’].convert_objects(convert_numeric=True)
```

In [ ]:

```
#Including the Fund_Year as a categorical variable. Due to limited time, I am unable t
#however I want to at least incorporate some date information into my classification m
#include more dates or run time-series analysis.
for d in df.index:
df[’funded_date’][d]=pd.to_datetime(df[’funded_date’][d])

fund_yr=[]
for i in df.index:
if (df[’funded_date’][i]==’N/A’): yr=’N/A’
else: yr=df[’funded_date’][i].year
fund_yr.append(yr)

df[’fund_yr’]=fund_yr
df[’fund_yr’]=df[’fund_yr’].convert_objects(convert_numeric=True)
```

In [ ]:

```
#remove some useless fields and output final cleaned up data
f=(u’disbursal_date’, u’funded_date’, u’paid_date’, u’partner_id’, u’planned_expiratio
for i in f:
del df[i]
df.dtypes
df.to_csv(’sample10000.csv’, sep=’\t’, encoding=’utf-8’, na_rep=’N/A’, header=True)
df=pd.read_table(’sample10000.csv’)
```

Part III

Exploratory Data Analysis

The goal of this analysis is to classify Delinquent Loans (1) versus Non-Delinquent loans (0), where “delinquent” is a binary variable. This Notebook takes the input of “sample10000.csv”, which is a sample of 10000 loans produced from the “Extraction & Data Prep” step. There are actually slightly less than 10000 loans, because some loans that were sampled were complete NAs.

In [2]:

```
import os
import json
from datetime import datetime
from dateutil import parser
from pandas import DataFrame, Series
import pandas as pd
import numpy as np
import statsmodels.api as sm
from sklearn import linear_model
import sklearn as sk
import pylab as pl
%pylab inline
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
from pandas.tools.plotting import scatter_matrix
import matplotlib.cm as cm
import pylab as pl

os.chdir(’/Users/bellepeng/Documents/Stat222_Capstone’)
#os.chdir(’/accounts/grad/belle.peng/Desktop’)
#os.chdir(’/accounts/grad/belle.peng/Desktop/iPython Notebook’)
#you will need to reset your working directory here
df=pd.read_table(’sample10000.csv’)
#df.pop(’Unnamed: 0’)
```

```
Populating the interactive namespace from numpy and matplotlib

WARNING: pylab import has clobbered these variables: [’datetime’]
‘%pylab --no-import-all‘ prevents importing * from pylab and numpy
```

Below I will be creating different tables and summaries to understand each variable and decide what to feed into my model

In [3]:

```
## Analyizing the Categorical variables
len(df[’activity’].unique()) #49 categories
df[’activity’].value_counts() # groups
len(df[’status’].unique()) #13
df[’status’].value_counts()
len(df[’country’].unique()) #82
df[’country’].value_counts()
len(df[’loss_liability’].unique())
df[’loss_liability’].value_counts()
len(df[’sector’].unique())
df[’sector’].value_counts();
```

In [4]:

```
### Analyze the Booleans
df[’bonus_credit_eligibility’].value_counts()
df[’delinquent’].value_counts()
df[’status’].value_counts()
```

Out [4]:

```
paid                7645
in_repayment        1676
defaulted            205
expired              154
inactive_expired     122
refunded              69
fundraising           56
deleted               40
funded                11
reviewed              10
inactive               9
issue                  3
```

```
dtype: int64
```

The statuses “paid”, “in_repaymen”, and “defaulted” are the biggest categories and have clear definitions. The other categories are small and delinquency is not applicable, therefore excluded from modling.

```
In [3]: ### Analyze the Continuous variables
        df['paid_amount']=df['paid_amount'].convert_objects(convert_numeric=True)
        df['currency_exchange_loss_amount']=df['currency_exchange_loss_amount'].convert_object
        df['disbursal_amount']=df['disbursal_amount'].convert_objects(convert_numeric=True)
        df['funded_amount']=df['funded_amount'].convert_objects(convert_numeric=True)
        df['loan_amount']=df['loan_amount'].convert_objects(convert_numeric=True)
        df['lender_count']=df['lender_count'].convert_objects(convert_numeric=True)
        df['repayment_term']=df['repayment_term'].convert_objects(convert_numeric=True)
        df['avg_invitee_count']=df['avg_invitee_count'].convert_objects(convert_numeric=True)
        df['avg_loan_count']=df['avg_loan_count'].convert_objects(convert_numeric=True)
        df['country_count']=df['country_count'].convert_objects(convert_numeric=True)
        df['fund_yr']=df['fund_yr'].convert_objects(convert_numeric=True)
```

```
In [6]: df['currency_exchange_loss_amount'].describe()
        df['disbursal_amount'].describe()
        df['funded_amount'].describe()
        df['loan_amount'].describe()
        df['lender_count'].describe()
        df['repayment_term'].describe()
        df['intercept'] = 1.0 #for logisitic regression
```

```
In [7]: #look at each status
        df.head(10)['status']
        df[df['status']=='paid']['delinquent'].value_counts()
        df[df['status']=='defaulted']['delinquent'].value_counts()
        df[df['status']=='in_repayment']['delinquent'].value_counts();
```

```
In [8]: # subsetting delinquency and statuses
        pd.crosstab(df.delinquent, df.status)
        #just sanity check, i.e. default should all be delinquent, paid should
        # not be delinquent, and in_repayment should have both delinq and non-delinq
```

```
Out [8]:
 status      defaulted  deleted  expired  funded  fundraising
 in_repayment  \
 delinquent
 False               0       40      154      11           56
 1515
 True              205        0        0       0            0
 161

 status      inactive  inactive_expired  issue  paid  refunded
 reviewed
 delinquent
 False              9               122      3  7645        68
 10
 True               0                 0      0     0         1
 0

 [2 rows x 12 columns]
```

```
In [9]: #Look at delinquency and countries
        table_country=pd.crosstab(df.delinquent, df.country)
        table_country.astype('float').div(table_country.sum(axis=0), axis=1)
```

```
Out [9]:
country     Afghanistan  Albania  Armenia  Azerbaijan  Belize   Benin  
\
delinquent
False          0.787234        1        1           1       1  0.9625
True           0.212766        0        0           0       0  0.0375

country      Bolivia  Bosnia and Herzegovina  Brazil  Bulgaria  
Burkina Faso  \
delinquent
False       0.986547                       1       1         1
1
True        0.013453                       0       0         0
0

country      Burundi  Cambodia  Cameroon  Chad  Chile  Colombia  Congo  
\
delinquent
False       0.928571         1  0.972222     1      1  0.986111      1
True        0.071429         0  0.027778     0      0  0.013889      0

country     Costa Rica  Cote D’Ivoire
delinquent
False             0.86              1  ...
True              0.14              0  ...

[2 rows x 73 columns]
```

```
In [10]: #Look at delinquency and acitivities
         table_activity=pd.crosstab(df.delinquent, df.activity);
         table_activity.astype('float').div(table_activity.sum(axis=0), axis=1)
```

```
Out [10]:
activity    Agriculture  Animal Sales      Arts  Auto Repair    Bakery  
\
delinquent
False              0.96      0.980296  0.823529     0.966667  0.920455
True               0.04      0.019704  0.176471     0.033333  0.079545

activity    Barber Shop  Beauty Salon  Bicycle Repair  Bicycle Sales  
\
delinquent
False                 1      0.971963               1              1
True                  0      0.028037               0              0

activity    Blacksmith  Bookstore  Bricks  Butcher Shop      Cafe  \
delinquent
False         0.846154   0.909091       1      0.931034  0.955224
True          0.153846   0.090909       0      0.068966  0.044776
```

```
activity     Call Center  Carpentry  Catering    Cattle  Cement
Cereals
delinquent
False                  1   0.970588         1  0.957746       1
0.943182 ...
True                   0   0.029412         0  0.042254       0
0.056818 ...

[2 rows x 143 columns]
```

```
In [11]: table_liability=pd.crosstab(df.delinquent, df.loss_liability)
         table_liability.astype('float').div(table_liability.sum(axis=0), axis=1)
```

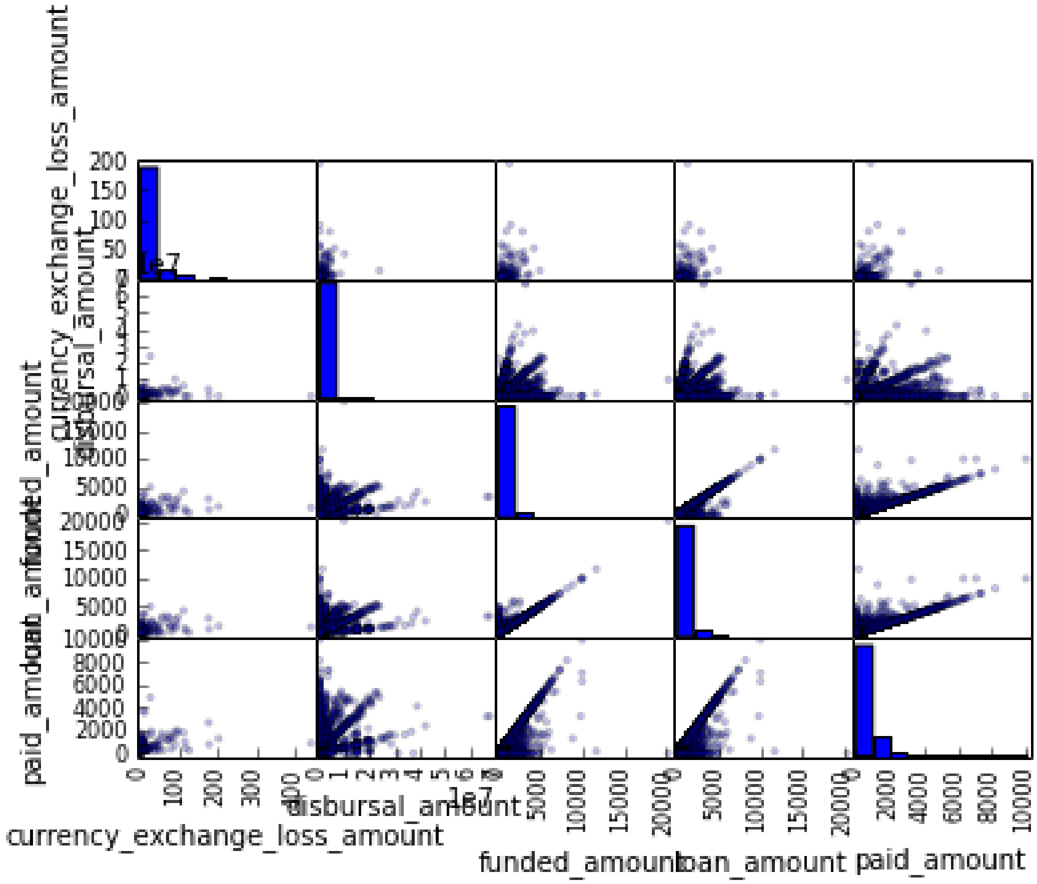
```
Out [11]:
         loss_liability    lender   partner
         delinquent
         False           0.958807  0.980345
         True            0.041193  0.019655

         [2 rows x 2 columns]
```

```
In [12]: table_sector=pd.crosstab(df.delinquent, df.sector);
         table_sector.astype('float').div(table_sector.sum(axis=0), axis=1);
```

Below I want to look at the relationship between the continuous variables. I see that the data is skewed and that paid_amount, loan_amount, and funded_amount have positive relationship, which I definitely want to feed into my model. There appears to be possible multiple groups in disbursal_amount, so I will also select this variable to feed into my model. Although the relationship in currency_exchange_loss_amount with the other amounts are not as clear, there seems to be some positive relationship, so I will also include this variable.

```
In [4]: ### Pairs Plot - just to look at, not for presentation
        import matplotlib.pyplot as plt
        from matplotlib.patches import Polygon
        plt.close('all')
        df_plot=df[["currency_exchange_loss_amount", "delinquent", "disbursal_amount",
        "funded_amount", "lender_count", "loan_amount", "paid_amount", "repayment_term",
        "avg_invitee_count", "avg_loan_count", "country_count"]]
        df_plot=df[["currency_exchange_loss_amount", "disbursal_amount", "funded_amount", "loa
        fig = plt.figure(figsize=(10,8)) #, dpi=1600
        axes = pd.tools.plotting.scatter_matrix(df_plot, alpha=0.2)
        #plt.savefig('scatter_matrix.png')
```

```
<matplotlib.figure.Figure at 0x106b04310>
```



In [5]:

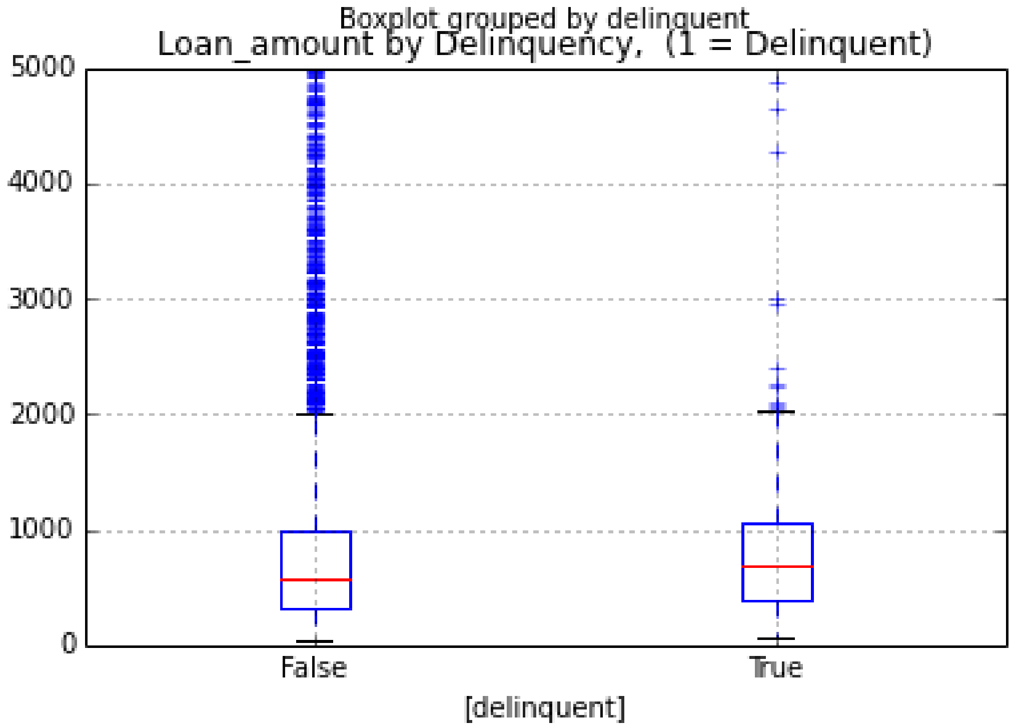
```
### Converting Data Formats
df[’paid_amount’]=df[’paid_amount’].convert_objects(convert_numeric=True)
df[’currency_exchange_loss_amount’]=df[’currency_exchange_loss_amount’].convert_object
df[’disbursal_amount’]=df[’disbursal_amount’].convert_objects(convert_numeric=True)
df[’funded_amount’]=df[’funded_amount’].convert_objects(convert_numeric=True)
df[’loan_amount’]=df[’loan_amount’].convert_objects(convert_numeric=True)
df[’lender_count’]=df[’lender_count’].convert_objects(convert_numeric=True)
df[’repayment_term’]=df[’repayment_term’].convert_objects(convert_numeric=True)
df[’avg_invitee_count’]=df[’avg_invitee_count’].convert_objects(convert_numeric=True)
df[’avg_loan_count’]=df[’avg_loan_count’].convert_objects(convert_numeric=True)
df[’country_count’]=df[’country_count’].convert_objects(convert_numeric=True)
df2=df
```

In [29]:

```
# boxplot of loan amount by delinquency #
df.boxplot(column=’loan_amount’, by=’delinquent’)
plt.ylim(0, 5000)
plt.title("Loan_amount by Delinquency,  (1 = Delinquent)")
#plt.savefig(’Fig1-Loan_amount_by_Delinquency.png’)
```

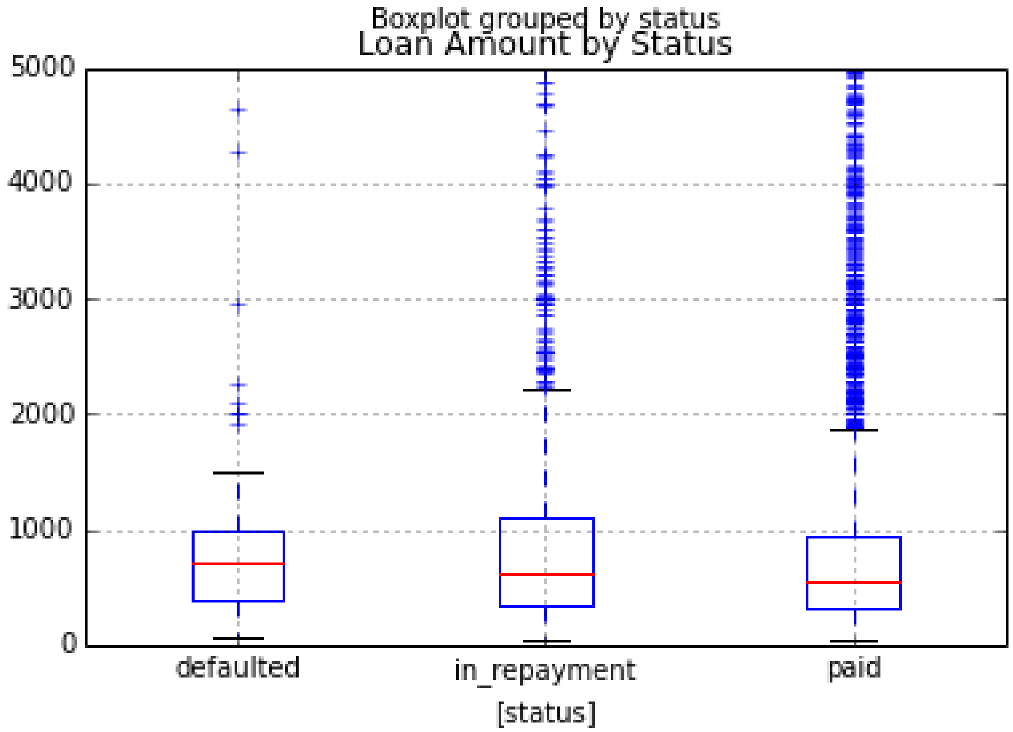
Out [29]:

```
<matplotlib.text.Text at 0x10c3565d0>
```



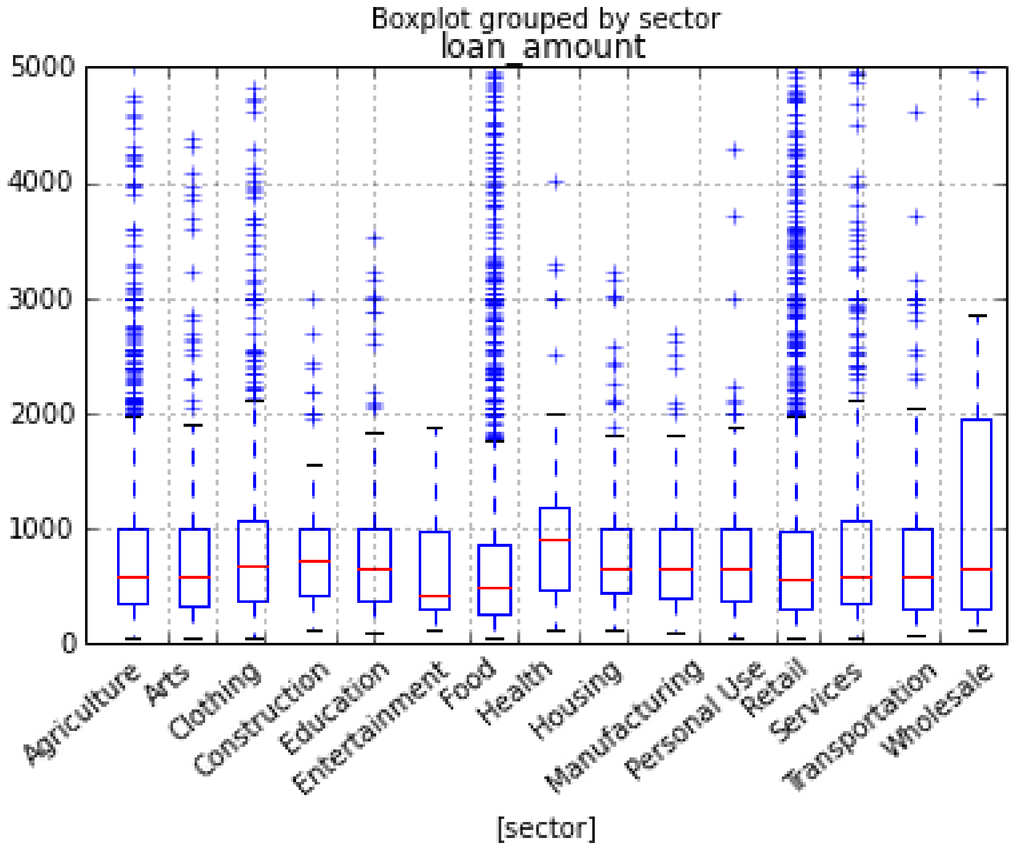
I suppected that there is a difference in loan_amount between delinquent and non-delinquent loans. So I made the box plot below to look at the distribution of the loan_amounts by delinquency. It appears the distribution is similar, both have a lot of outliers, it's just that there are a lot more non-delinquent loans. This shows me that most of the loans have small amounts, but some loans

In [30]:
```
df[df.status.isin([’paid’, ’defaulted’, ’in_repayment’])].boxplot(column=’loan_amount’
plt.ylim(0, 5000)
plt.title("Loan Amount by Status")
plt.savefig(’Fig-Loan_amount_by_Status.png’)
```



I also suspected there may be a difference in loan_amount distributions by status. I am only mainly interested in “paid”, “in_repayment”, and “defaulted”. Again, similar distribution of loan amounts, lots of outliers on the right tail. However, “paid” and “in_repayment” seem to have slightly higher amounts that the “defaulted” loans. This is perhaps because Kiva lenders are careful, only lending out to the more successful loans and if they notice any signs of possible default, they don’t lend to these loans.

```
In [19]: # boxplot of loan amount by sector #
         df.boxplot(column='loan_amount', by='sector')
         #plt.xticks(np.arange(0.4,15.4), rotation=40)
         plt.xticks(np.array([0.2,   1.6,   2.4,   3.2,   4.4,   5.0,   6.6,   7.4,   8.4,
                   9.2,  10.4,  11.7,  12.4,  13.1,  14.4]), rotation=40)
         plt.ylim(0, 5000)
         plt.show()
         plt.savefig('Fig1-Loan_amount_by_Sector.png')
```



```
<matplotlib.figure.Figure at 0x115b53d10>
```

Sectors have similar distribution. Also many outliers.Based on my understanding of the data from the EDA step, below are the variables I chose for modeling:

(unless indicated, it's a loans attribute)

loan_amount

funded_amount

paid_amount

disbursal_amount

currency_ex_loss

repayment_term

lender_count

avg_invitee_count (lenders data)

avg_loan_count (lenders data)

country_count (lenders data)

delinquent

activity

bonus_credit_eligibility

country

fund_yr

sector

status

loss_liability

Part IV

Modeling & Classification

In [6]:

```
### Converting Data Formats
df[’paid_amount’]=df[’paid_amount’].convert_objects(convert_numeric=True)
df[’currency_exchange_loss_amount’]=df[’currency_exchange_loss_amount’].convert_object
df[’funded_amount’]=df[’funded_amount’].convert_objects(convert_numeric=True)
df[’loan_amount’]=df[’loan_amount’].convert_objects(convert_numeric=True)
df[’lender_count’]=df[’lender_count’].convert_objects(convert_numeric=True)
df[’repayment_term’]=df[’repayment_term’].convert_objects(convert_numeric=True)
df[’avg_invitee_count’]=df[’avg_invitee_count’].convert_objects(convert_numeric=True)
df[’avg_loan_count’]=df[’avg_loan_count’].convert_objects(convert_numeric=True)
df[’country_count’]=df[’country_count’].convert_objects(convert_numeric=True)
df2=df
```

In [7]:

```
# Factorize Booleans
df2.bonus_credit_eligibility, _ =pd.factorize(df2.bonus_credit_eligibility) #turn bool
df2.delinquent, _ =pd.factorize(df2.delinquent)
```

In [8]:

```
##### Fill NAs #####
#df2=df2.fillna(0)
df2=df2.replace(np.NaN, 0)
# only keep these statuses because others wouldn’t be expecting a payment
exclude=df2[df2.status.isin([’paid’, ’in_repayment’, ’defaulted’])==False]
df2=df2[df2.status.isin([’paid’, ’in_repayment’, ’defaulted’])]
#Fill paid_amount, avg_invitee_count, avg_loan_count, country_count with 0
#add currency_exchange_loss_amount to paid_amt, then remove it
df2.paid_amount=df2.paid_amount+df2.currency_exchange_loss_amount
df2.pop(’currency_exchange_loss_amount’);
df2.pop(’disbursal_amount’);
```

In [9]:

```
### Convert Categorical Variables and delete unused ones ###
sector_dummy = pd.get_dummies(df2[’sector’], prefix=’sector’)
sector_dummy.pop(’sector_Food’)
country_dummy = pd.get_dummies(df2[’country’], prefix=’country’)
country_dummy.pop(’country_United States’)
activity_dummy = pd.get_dummies(df2[’activity’], prefix=’activity’)
activity_dummy.pop(’activity_Agriculture’)
status_dummy = pd.get_dummies(df2[’status’], prefix=’status’)
status_dummy.pop(’status_paid’)
loss_dummy = pd.get_dummies(df2[’loss_liability’], prefix=’loss_liability’)
loss_dummy.pop(’loss_liability_lender’)
fundyr_dummy = pd.get_dummies(df2[’fund_yr’], prefix=’fund_yr’)
fundyr_dummy.pop(’fund_yr_2005.0’)
df2 = df2.join(sector_dummy.ix[:, ’sector’:])
df2 = df2.join(country_dummy.ix[:, ’country’:])
df2 = df2.join(activity_dummy.ix[:, ’activity’:])
```

```
df2 = df2.join(status_dummy.ix[:, ’status’:])
df2 = df2.join(loss_dummy.ix[:, ’loss_liability’:])
df2 = df2.join(fundyr_dummy.ix[:, ’fund_yr’:])
df2.pop(’sector’)
df2.pop(’country’)
df2.pop(’activity’)
df2.pop(’status’)
df2.pop(’loss_liability’)
df2.pop(’fund_yr’)
df2.pop(’id’)
df2.pop(’Unnamed: 0’);
```

I am splitting my data into 80% training and 20% test. Since I have a lot of data, I can afford to use 20% for testing. I am expecting approximately 280 delinquent loans in my training data, which is enough for making inference.

In [10]:

```
####### Split into Test and Training #####
np.random.seed(123)
df2[’is_train’] = np.random.uniform(0, 1, len(df2)) <= .8
train, test = df2[df2[’is_train’]==True], df2[df2[’is_train’]==False]
testOrig_index=df2[df2[’is_train’]==False].index
trainOrig_index=df2[df2[’is_train’]==True].index
train.pop(’is_train’)
test.pop(’is_train’)
features = train.columns[:1] + train.columns[2:]
    #sum(features=="delinquent") #test to make sure certain columns aren’t included, s
X=train[features]
Y=train[’delinquent’]
testX=test[features]
testY=test[’delinquent’]
```

6 Logistics Regression

The first method I am using is Logistics Regression, because I am trying to classify a binary response variable, and I like this method for its intepretability. This Logistics Regression Model is run using L-1 Penalty. One big advantage is variable selection. I used cross-validation to iterate over different penalty levels, c's, to pick the optimal c.

In [36]:

```
from sklearn.linear_model import LogisticRegression
from sklearn import cross_validation
n_features = X.shape[1]

# Use CV to pick tuning param, C
CV_errors=[]
c_s=np.arange(0.1, 15.1, 0.1)
c=0.1
while c < 15:
    #print(c)
    classifier=LogisticRegression(C=c, penalty=’l1’)
    scores = np.mean(cross_validation.cross_val_score(classifier, X, Y, cv=5))
    CV_errors.append(scores)
    c=c+0.1
```

In [37]:

```
#find the c with the minimum cv error
pickC=pd.DataFrame({’CV_errors’:CV_errors, ’c’:c_s})
pickC.CV_errors.describe() #low SD, all inbetween around 98%
pickC.c[pickC.CV_errors==min(pickC.CV_errors)] #0.1
```

Out [37]:

```
0    0.1
Name: c, dtype: float64
```

In [38]:

```
# Run Model with the selected C
c=0.1
classifier=LogisticRegression(C=c, penalty=’l1’, fit_intercept=True)
LR=classifier.fit(X, Y)
#LR_decision=LR.decision_function(X)
LR_pred_test = LR.predict(testX)
LR.score(testX, testY)
```

Out [38]:

```
0.98119290703922624
```

In [39]:

```
# CV with the finalized model
CV_LR= cross_validation.cross_val_score(classifier, X, Y, cv=5)
print "CV Classification Rate Mean is ", CV_LR.mean()
print "CV Classification Rate Standard Deviation is ", CV_LR.std()
```

```
CV Classification Rate Mean is  0.98369210698
CV Classification Rate Standard Deviation is  0.00276754121795
```

Prediction on test data and cross-validation classification are pretty good, at 98%+. Let’s take a look at the confusion matrix.

In [42]:

```
# Confusion Matrix 0.327869
confusionLR=pd.crosstab(testY, LR_pred_test, rownames=[’Actual’], colnames=[’Predictio
confusionLR.div(confusionLR.astype(float).sum(axis=1), axis=0)
```

Out [42]:

```
Prediction         0         1
Actual
0           0.999438  0.000562
1           0.409639  0.590361

[2 rows x 2 columns]
```

The confusion matrix shows us that the model did very well for the non-delinquent loans, but not that well on the delinquent loans, with 40.96% error rate. This makes sense, since most of the loans are non-delinquent, just by guessing a loan is non-delinquent, you would get ~96.5% correct (recall that ~3.5% are non-delinquent in the overall population). So let’s focus the next few models on reducing the classification error rate on the delinquent loans while keeping up the good prediction on the non-delinquent loans. But first let’s look at what the Logistics Model selected as the influential variables.

In [40]:

```
# Look at the coefficients selected by LR
LR.intercept_
LR_coef=pd.DataFrame({’coef’: LR.coef_[0], ’coef_abs’: abs(LR.coef_[0])}, index=featur
LR_coef=LR_coef.sort(’coef_abs’, ascending=False)
LR_coef.head(50)
LR_coef.coef[(LR_coef.coef_abs>0)] #selected variables
```

Out [40]:

```
status_defaulted           7.718436
status_in_repayment        4.437271
intercept                 -1.780652
fund_yr_2013.0            -1.299484
fund_yr_2012.0             1.147961
fund_yr_2014.0            -0.879070
bonus_credit_eligibility  -0.495518
```

```
country_Kenya                 0.413129
repayment_term               -0.053028
country_count                -0.050980
lender_count                 -0.002713
avg_invitee_count            -0.001440
loan_amount                   0.000607
paid_amount                  -0.000482
funded_amount                -0.000313
avg_loan_count               -0.000044
Name: coef, dtype: float64
```

Status is important, this makes intuitive sense. Also the fund years seem to have influence, for example if a loan was funded in 2012, it is negatively correalted with delinquency. Further, if a loan qualifies for bonus credits then it is less likely to delinquent, while the country Kenya is more likely to default. Another interesting one is that if a loan has longer repayment term, it is less likely to default, which I think can be interpreted in many ways. If a loan has longer to repay its debt, then it is less likely to default.

In [43]:

```
#Look at the misclassified test data
misclass_test_LR=np.where(LR_pred_test!= testY)[0]
misclass_LR_orig=testOrig_index[misclass_test_LR]
df.ix[misclass_LR_orig]
len(misclass_LR_orig) #35 misclassified in test data
```

Out [43]:

```
35
```

In [45]:

```
## misclassified training data
LR_predX=LR.predict(X)
misclass_train_LR=np.where(LR_predX!= Y)[0]
misclass_train_LR_orig=trainOrig_index[misclass_train_LR]
df.ix[misclass_train_LR_orig]
len(misclass_train_LR_orig) #118 misclassified in training
```

Out [45]:

```
118
```

What about the misclassified data in the test set and the training set? Here I am extracting these data points to look at them to see if I can identify any pattern. If I identify patterns, then my model did not capture this information, so I am expecting to see the data scattered across. Sure enough, I was not able to see any pattern both from the data here and from the plots (shown later).

7 Random Forest

To improve on the LR above, I am trying out Random Forest to classify loan delinquency. Since the Logistics Regression above has about ~41% misclassification rate on the Delinquent loans, I want to see if Random Forest reduces this misclassification. One primary advantage of the Random Forest is it's ease in interpretability. I used cross-validation to pick the optimal number of trees to grow.

The misclassification rate on delinquent loans using Random Forest (RF) compared to Logistics Regression (LR) reduced from ~41% to ~33%, and 0% misclassification on the non-delinquent loans.

In [46]:

```
from sklearn.ensemble import RandomForestClassifier
# Use cross validation to find the  optimum number of trees to grow
CV_errors=[]
n_s=np.arange(1, 50, 1)
```

```
n=1
while n < 50:
    #print(n)
    RF = RandomForestClassifier(n_estimators=n, n_jobs=-1)
    scores = np.mean(cross_validation.cross_val_score(RF, X, Y, cv=5))
    CV_errors.append(scores);
    n=n+1
```

In [52]:

```
pickN=pd.DataFrame({'CV_errors':CV_errors, 'n':n_s})
#pickN.CV_errors.describe() #all between 0.9715 and 0.9855, sd=0.002
pickN.n[pickN.CV_errors==min(pickN.CV_errors)]
#pickN.sort('CV_errors', ascending=True)
```

Out [52]:

```
0    1
Name: n, dtype: int64
```

Even though CV says the optimum number of trees should be 1, that's basically the same as Classification tree, which I will run later. For purpose of comparing results to LR and Classification Tree, I decided to picked 40 trees to grow, at n=40 it still has CV classification rate of 98%.

In [53]:

```
# Run final model with selected n_estimator
n=40
RF = RandomForestClassifier(n_estimators=n, n_jobs=1)
RF.fit(X, Y)
RF.score(testX, testY) #0.9904
RF_pred_test = RF.predict(testX)
RF_prob_test = RF.predict_proba(testX) #check out the prob some more
```

In [54]:

```
# Cross Validation
CV_RF= cross_validation.cross_val_score(RF, X, Y, cv=10)
print "CV Classification Rate Mean is ", CV_RF.mean()
print "CV Classification Rate Standard Deviation is ", CV_RF.std()
#~0.98578027716408911
#~0.0037543185377078869
```

```
CV Classification Rate Mean is  0.98630178955
CV Classification Rate Standard Deviation is  0.0035592022252
```

Again prediction on test data and CV classification are above 98%, what about the confusion matrix?

In [56]:

```
# Confusion Matrix - missclass:0.295082
confusionRF=pd.crosstab(testY, RF_pred_test, rownames=['Actual'], colnames=['Predictio
confusionRF.div(confusionRF.astype(float).sum(axis=1), axis=0)
```

Out [56]:

```
Predictions         0         1
Actual
0            0.999438  0.000562
1            0.313253  0.686747

[2 rows x 2 columns]
```

Confusion matrix shows us that the prediction error on non-delinquent loans is again very small, and prediction error on delinquent loans decreased to around 31%.

In [55]:
```
# Top Features
RF_coef=pd.DataFrame({'coef': RF.feature_importances_, 'coef_abs': abs(RF.feature_impo
RF_coef=RF_coef.sort('coef_abs', ascending=False)
RF_coef.coef.head(15)
RF_coef.coef[(RF_coef.coef_abs>0)]
```

Out [55]:
```
status_defaulted            0.264452
paid_amount                 0.177595
status_in_repayment         0.050897
avg_invitee_count           0.047724
avg_loan_count              0.039523
country_count               0.036355
repayment_term              0.036153
loan_amount                 0.035083
funded_amount               0.033172
lender_count                0.031351
fund_yr_2012.0              0.021768
fund_yr_2013.0              0.014284
country_Kenya               0.011070
bonus_credit_eligibility    0.010983
country_Mexico              0.008944
...
activity_Perfumes                     6.798827e-06
activity_Dental                       5.526420e-06
activity_Recycled Materials           4.951009e-06
activity_Vehicle Repairs              4.023098e-06
activity_Jewelry                      3.113263e-06
activity_Musical Performance         -3.066209e-06
activity_Education provider          -2.116288e-06
activity_Home Energy                 -1.558522e-06
activity_Milk Sales                  -1.202694e-06
country_Yemen                        -1.184205e-06
sector_Entertainment                 -3.078759e-07
country_Ukraine                       6.637279e-08
country_Bulgaria                      6.636653e-08
activity_Personal Medical Expenses    6.635869e-08
activity_Mobile Phones               -4.740693e-08
Name: coef, Length: 199, dtype: float64
```

The influential characteristics have some similarity to LR, and some new ones. For example, loan status remains to be an important feature, so is repayment term and fund year. However, paid amount, funded amount, loan amount, country count of the lenders, and average invitee counts are now also considered influential in this model. The interesting one is repayment term now has a positive correlation rather than negative correlation as we saw from the LR model. So here, the longer repayment term contributes to delinquency, while the opposite is true from the LR model. Some of the countries and activities are also showing up here in this model as influential in explaining delinquency. Let's see what the Classification tree tells us next.

In [57]:
```
#Look at misclassified points
misclass_RF=np.where(RF_pred_test!= testY)[0]
misclass_RF=testOrig_index[misclass_RF]
df.ix[misclass_RF]
len(misclass_RF) #27 misclassified points in test data
```

```
Out [57]:
          27
```

Again I examine the misclassified points and expect no patter, which is the case. This will be shown in the plots later.

7.1 Classification Tree

From my Random Forest analysis above, the optimum number of tree to grow is actually 1 from cross validation, so here I am trying the Classification Tree method, looking for better classification rate, and interpretability.

The misclassification rate on delinquent loans using Classification Tree (CT) reduced compared to both Random Forest (RF) and Logistics Regression (LR). CT is ~24%, RF is ~33%, and LR is ~41%. Also 0% misclassification on the non-delinquent loans.

```
In [11]: from sklearn.tree import DecisionTreeClassifier
         from sklearn import cross_validation
         from sklearn import tree as tree
         treeclf=DecisionTreeClassifier(max_depth=5)
         treeclf.fit(X, Y)
         treeclf.score(testX, testY) #0.9952
         treeclf.predict_proba(testX)
         tree_pred_test=treeclf.predict(testX)
```

```
In [12]: # Cross Validaiton
         CV_tree= cross_validation.cross_val_score(treeclf, X, Y, cv=5)
         print "CV Classification Rate Mean is ", CV_tree.mean()
         print "CV Classification Rate Standard Deviation is ", CV_tree.std()
         #~0.99034572733202864
         #~0.00048814838705466229
```

```
         CV Classification Rate Mean is  0.990084801044
         CV Classification Rate Standard Deviation is  0.000488148387055
```

The classification rate on test data is about 99.5%, and the CV classification rate is at 99%, improved even from the previously two models.

```
In [13]: # confusion matrix
         confusionTree=pd.crosstab(testY, tree_pred_test, rownames=['Actual'], colnames=['Predi
         confusionTree.div(confusionTree.astype(float).sum(axis=1), axis=0)
```

```
Out [13]:
         Predictions         0         1
         Actual
         0            1.000000  0.000000
         1            0.240964  0.759036

         [2 rows x 2 columns]
```

The confusion matrix above shows us that the classification tree was able to predict the non-delinquent loans correctly 100% of the time, while its classification error rate on delinquent loans is 24%, reduced from LR and RF.

```
In [14]: # Top Features
         tree_coef=pd.DataFrame({'coef': treeclf.feature_importances_, 'coef_abs': abs(treeclf.
         tree_coef=tree_coef.sort('coef_abs', ascending=False)
         tree_coef.coef.head(12)
         tree_coef.coef[(tree_coef.coef_abs>0)]
```

Out [14]:

```
status_defaulted       0.697229
fund_yr_2012.0         0.102238
repayment_term         0.091296
status_in_repayment    0.045195
avg_loan_count         0.026941
country_Mexico         0.014718
funded_amount          0.011369
activity_Poultry       0.006419
country_Cameroon       0.004594
Name: coef, dtype: float64
```

The top features are consistent with the other two models: status, fund year 2012, repayment term, average loan count, Mexico, funded amount and activity Poultry.

In [15]:

```
# Misclass
misclass_tree=np.where(tree_pred_test!= testY)[0]
misclass_tree_orig=testOrig_index[misclass_tree] #19
df.ix[misclass_tree]
len(misclass_tree) #20 misclassified points
```

Out [15]:

```
20
```

Again no patterns in the misclassified data, which is what we are looking for.

In [16]:

```
## look at the results
tree_predX=treeclf.predict(X)
confusion_train=pd.crosstab(Y, tree_predX, rownames=[’Actual’], colnames=[’Predictions
misclass_train_tree=np.where(tree_predX!= Y)[0]
misclass_train_tree_orig=trainOrig_index[misclass_train_tree]
pd.crosstab(Y, tree_predX, rownames=[’Actual’], colnames=[’Predictions’])
```

Out [16]:

```
Predictions     0    1
Actual
0            7381    1
1              66  217

[2 rows x 2 columns]
```

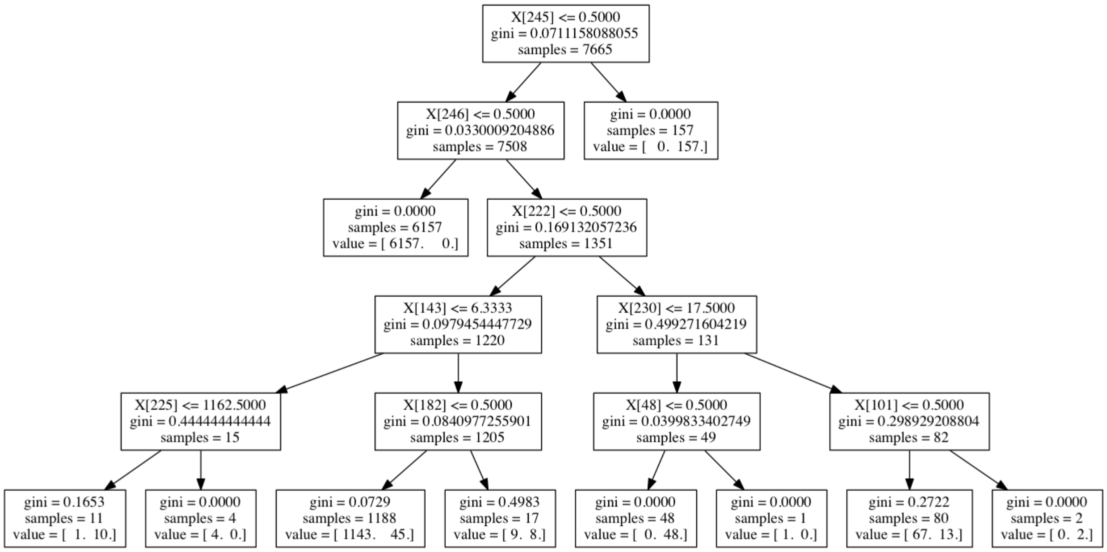
We can visiualize where the tree splits and what variables were used for splitting above.

In [42]:

```
### This plots the actual tree, which requires installing pydot and graphviz on your c
import pydot
import pyparsing
from sklearn.externals.six import StringIO
with open("DecisionTree.dot", ’w’) as f:
    f = tree.export_graphviz(treeclf, out_file=f)
dot_data = StringIO()
tree.export_graphviz(treeclf, out_file=dot_data, feature_names = X.columns)
#graph = pydot.graph_from_dot_data(dot_data.getvalue())
#graph.write_pdf("DecisionTree.pdf")
f.close()
#This ran before but stopped running when I tried again, so I am attaching the image p
# during midterm from the same code.
```

In [44]:
```
from IPython.display import Image
Image(filename='/Users/bellepeng/Documents/Stat222_Capstone/Graphs/DecisionTree.png')
```

Out [44]:

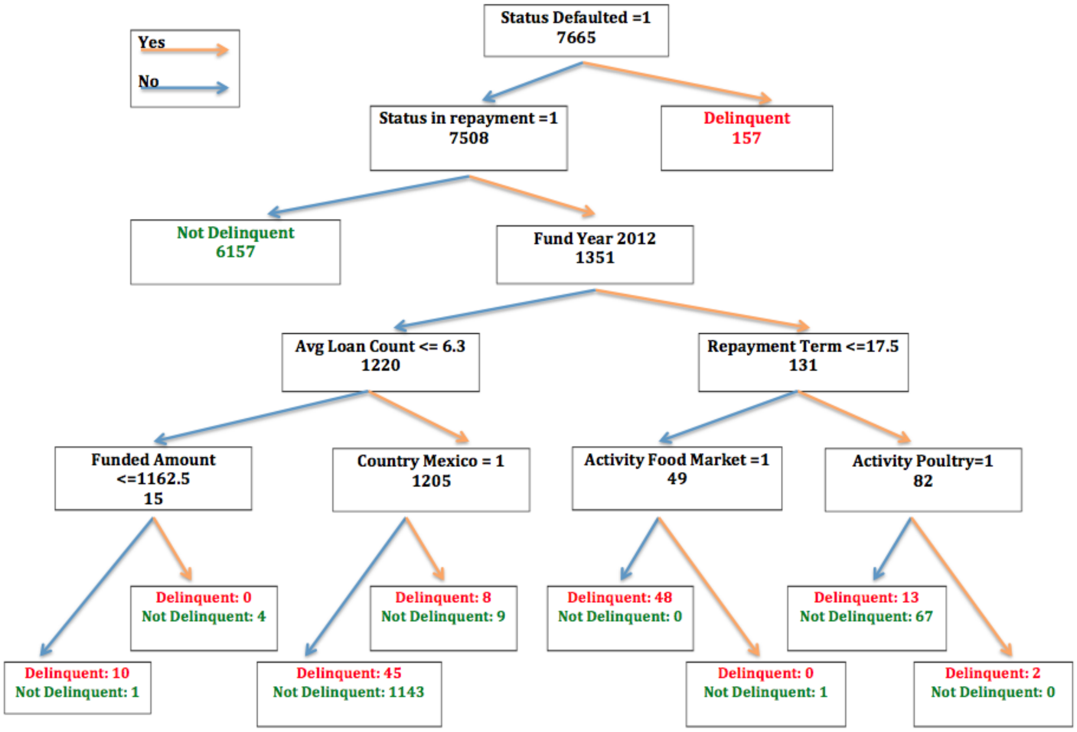


The code above produced a visualization of the Decision Tree with the gini criterion, but it's not clear what the splitting variables are, so I re-drew it for interpretability, shown below. We can see that it's splitting Status, Fund Year, Repayment Term, Average Loan Count, Mexico, etc. These are my influential variables from this model:

1. status_defaulted
2. fund_yr_2012.0
3. repayment_term
4. status_in_repayment
5. avg_loan_count
6. country_Mexico
7. funded_amount
8. activity_Poultry
9. country_Cameroon

In [48]:
```
Image(filename='/Users/bellepeng/Documents/Stat222_Capstone/Graphs/DecisionTree2.png')
```

Out [48]:



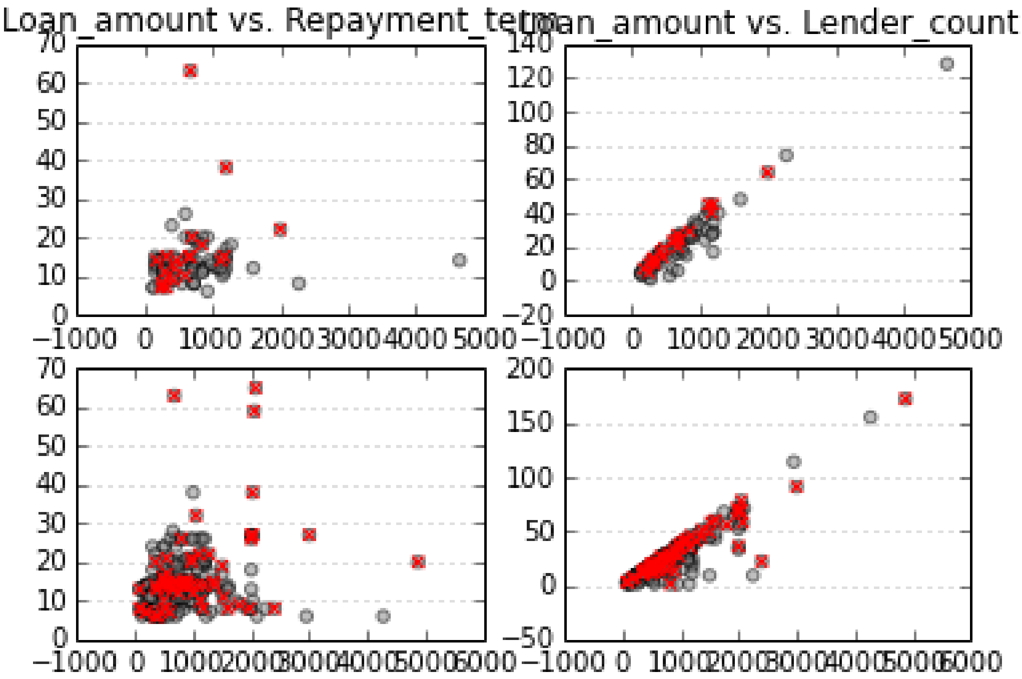
7.2 Plotting Some Results

In [69]:

```
###### Misclassified Points ########
#### Define Plot Data ####
plt_test_ix=testY[testY==1].index
plt_test_data=testX[(testX.loan_amount<=5000)].ix[plt_test_ix] #just the delinquent lo
plt_train_ix=X[Y==1].index
plt_train_data=X[(X.loan_amount<=5000)].ix[plt_train_ix]
```

In [70]:

```
### tree misclassified points - no pattern - good ###
f, axarr = plt.subplots(2, 2)
axarr[0,0].scatter(plt_test_data.loan_amount[(testY==1)], plt_test_data.repayment_term
axarr[0,0].scatter(plt_test_data.loan_amount[misclass_tree_orig], plt_test_data.repaym
axarr[0,0].set_title("Loan_amount vs. Repayment_term")
axarr[0,0].grid(b=True, which=’major’, axis=’y’, color=’0.5’)

axarr[0,1].scatter(plt_test_data.loan_amount[(testY==1)], plt_test_data.lender_count[(
axarr[0,1].scatter(plt_test_data.loan_amount[misclass_tree_orig], plt_test_data.lender
axarr[0,1].set_title("Loan_amount vs. Lender_count")
axarr[0,1].grid(b=True, which=’major’, axis=’y’, color=’0.5’)

axarr[1,0].scatter(plt_train_data.loan_amount[(Y==1)], plt_train_data.repayment_term[(
axarr[1,0].scatter(plt_train_data.loan_amount[misclass_train_tree_orig], plt_train_dat
axarr[1,0].grid(b=True, which=’major’, axis=’y’, color=’0.5’)

axarr[1,1].scatter(plt_train_data.loan_amount[(Y==1)], plt_train_data.lender_count[(Y=
axarr[1,1].scatter(plt_train_data.loan_amount[misclass_train_tree_orig], plt_train_dat
axarr[1,1].grid(b=True, which=’major’, axis=’y’, color=’0.5’)
plt.show()
plt.savefig('Fig3-Scatterplot of Misclassified Tree.png')
```



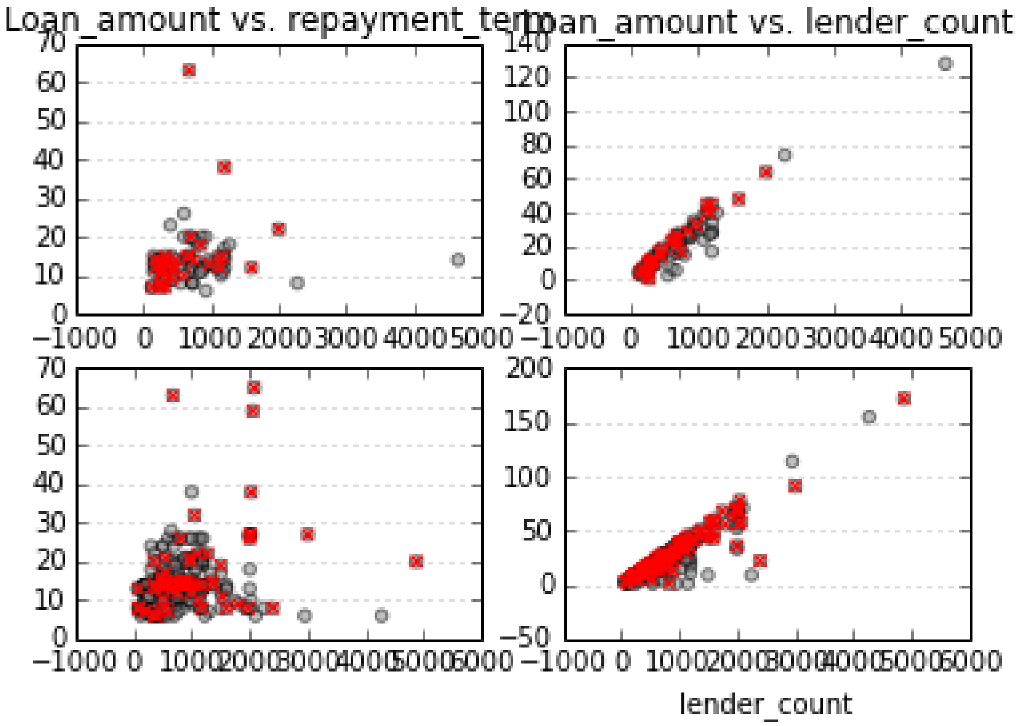
```
<matplotlib.figure.Figure at 0x10c373610>
```

I examined plotting over many different variables, I chose to show "Loan amount vs Repayment term" (left two graphs) and "Loan amount vs Lender count" (right two graphs) to visualize the misclassified points in the best model - classification tree. The top two graphs are from test data, while the bottom two graphs are from training data. The grey are the correctly classified points while the red are the misclassified points. We see similar pattern between training and test data (both correctly and incorrectly classified points), which is good, that means our training and test data are in fact similar. These three variables demonstrate that there is no particular pattern that separates the misclassified points. If there is, then I would include that distinguishing characteristic into my model.

In [71]:

```
### Logistic Regression misclassified points - no pattern - good ###
f, axarr = plt.subplots(2, 2)
axarr[0,0].scatter(plt_test_data.loan_amount[(testY==1)], plt_test_data.repayment_term
axarr[0,0].scatter(plt_test_data.loan_amount[misclass_LR_orig], plt_test_data.repaymen
axarr[0,0].set_title("Loan_amount vs. repayment_term")
plt.xlabel("loan_amount")
plt.xlabel("repayment_term")
axarr[0,0].grid(b=True, which='major', axis='y', color='0.5')

axarr[0,1].scatter(plt_test_data.loan_amount[(testY==1)], plt_test_data.lender_count[(
axarr[0,1].scatter(plt_test_data.loan_amount[misclass_LR_orig], plt_test_data.lender_c
axarr[0,1].set_title("Loan_amount vs. lender_count")
plt.xlabel("loan_amount")
plt.xlabel("lender_count")
axarr[0,1].grid(b=True, which='major', axis='y', color='0.5')

axarr[1,0].scatter(plt_train_data.loan_amount[(Y==1)], plt_train_data.repayment_term[(
axarr[1,0].scatter(plt_train_data.loan_amount[misclass_train_tree_orig], plt_train_dat
plt.xlabel("loan_amount")
plt.xlabel("repayment_term")
axarr[1,0].grid(b=True, which='major', axis='y', color='0.5')
```

```
axarr[1,1].scatter(plt_train_data.loan_amount[(Y==1)], plt_train_data.lender_count[(Y=
axarr[1,1].scatter(plt_train_data.loan_amount[misclass_train_LR_orig], plt_train_data.
plt.xlabel("loan_amount")
plt.xlabel("lender_count")
axarr[1,1].grid(b=True, which='major', axis='y', color='0.5')
plt.show()
plt.savefig('Fig4-Scatterplot of Misclassified LR.png')
```



```
<matplotlib.figure.Figure at 0x10c295f50>
```

This is a similar graph as above for Logistics Regression. Again I am showing "loan amount vs repayment term" and "loan amount vs lender count" just to demonstrate that there is no pattern in any variable, but I have examined other variables as well. The top two graphs are from test data, while the bottom two graphs are from training data. The grey are the correctly classified points while the red are the misclassified points. We see similar pattern between training and test data (both correctly and incorrectly classified points), which is good, that means our training and test data are in fact similar. These three variables demonstrate that there is no particular pattern that separates the misclassified points. If there is, then I would include that distinguishing characteristic into my model.

Combining all of my results from all three of my models above, I conclude that these are the influential features that explain loan delinquency: (+ means positive correlation across all methods, +- means positive correlation from one method and negative correlation from another method)

1. status_defaulted +
2. status_in_repayment +
3. fund_yr_2012 +
4. repayment_term (small correlation +-)
5. avg_loan_count (small correlation +-)
6. funded_amount (small correlation +)
7. loan_amount (small correlation +)

Part V

Test Suite

This section tests the data cleaning portion of my code that calculates the combined paid amount by adding paid amount with currency exchange loss amount. Because the currency exchange loss amount is so small and too many NAs that adding it to paid amount makes more sense than to include it into the model. This section also tests my use of a function to convert the NAs to 0 and make sure it worked on some test data. The tests successfully passed.

In [1]:
```
import nose
```

In [2]:
```
%%file test_calc_paid_amt.py
def calc_paid_amt(paid_amount, currency_exchange_loss_amount):
    paid_amount=paid_amount+currency_exchange_loss_amount
    return paid_amount

def test_1():
    result = round(calc_paid_amt(10, 1), 2)
    print ’Testing Calculation of the Paid Amount’, result
    assert result == 11
def test_2():
    result = round(calc_paid_amt(-999, -1), 2)
    print ’Testing Calculation of the Paid Amount’, result
    assert result == -1000
def test_3():
    result = round(calc_paid_amt(-999, 0), 2)
    print ’Testing Calculation of the Paid Amount’, result
    assert result == -999
```

```
Writing test_calc_paid_amt.py
```

In [3]:
```
!cd /accounts/grad/belle.peng/Dropbox
!nosetests -v test_calc_paid_amt.py
```

```
/bin/sh: 1: cd: can’t cd to /accounts/grad/belle.peng/Dropbox
test_calc_paid_amt.test_1 ... ok
test_calc_paid_amt.test_2 ... ok
test_calc_paid_amt.test_3 ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

In [4]:
```
%%file test_na.py
import pandas as pd
import numpy as np

def replaceNA(obj):
    obj=pd.DataFrame(data=obj)
    obj=obj.replace(np.NaN, 0)
    return obj

def test_1():
    result = replaceNA([np.NaN]*10)
    print ’The type of object is’, result
    assert result == [0]*10
def test_2():
    result = replaceNA(np.array([5, 5, np.NaN, np.NaN, np.NaN]))
    print 'The type of object is', result
    assert result == [5, 5, 0, 0, 0]
```

```
Writing test_na.py
```

In [5]:

```
!nosetests -v test_na.py
```

```
test_na.test_1 ... ok
test_na.test_2 ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.013s

OK
```

Part VI

Convert iPython Notebook into PDF

In [ ]:

```
!ipython nbconvert --to latex --post PDF --SphinxTransformer.author='Belle Peng' Kiva.
```