CS156 Assignment 2 - Quang Tran

February 15, 2019

This assignment attempts to build a model that aids making decision of how much one should apply for a loan. The data used is downloaded from The Lending Club

1 Data Cleaning

1.1 Download and read data

For each category (rejected or approved), we downloaded data from 2018 Q3 as those are the most recent ones. **Assumption:** More recent data reflect better the distribution of the current.

Let us read in the data files.

```
In [1]: import numpy as np
        import scipy as sp
        import matplotlib as mpl
        import matplotlib.cm as cm
        import matplotlib.pyplot as plt
        import pandas as pd
        import time
        import random
        from sklearn.model_selection import train_test_split
```

```
In [2]: df_app = pd.read_csv("./data2/LoanStats_2018Q3.csv", skiprows=[0], sep=',',na_values=[
        df_app.head()
```

```
/anaconda3/lib/python3.6/site-packages/IPython/core/interactiveshell.py:2785: DtypeWarning: Co
  interactivity=interactivity, compiler=compiler, result=result)
```

```
Out[2]:     id  member_id  loan_amnt  funded_amnt  funded_amnt_inv        term  \
        0  NaN        NaN    10000.0      10000.0          10000.0   36 months
        1  NaN        NaN    15000.0      15000.0          15000.0   60 months
        2  NaN        NaN     8500.0       8500.0           8500.0   36 months
        3  NaN        NaN    35000.0      35000.0          35000.0   36 months
        4  NaN        NaN    28000.0      28000.0          28000.0   60 months

          int_rate  installment grade sub_grade       ...         \
        0    6.67%       307.27     A        A2       ...
        1   18.94%       388.62     D        D2       ...
```

```
2   14.47%       292.46     C        C2       ...
3    6.67%      1075.43     A        A2       ...
4   14.47%       658.36     C        C2       ...

  hardship_payoff_balance_amount hardship_last_payment_amount  \
0                            NaN                          NaN
1                            NaN                          NaN
2                            NaN                          NaN
3                            NaN                          NaN
4                            NaN                          NaN

  disbursement_method  debt_settlement_flag debt_settlement_flag_date  \
0          DirectPay                    N                        NaN
1               Cash                    N                        NaN
2               Cash                    N                        NaN
3               Cash                    N                        NaN
4               Cash                    N                        NaN

  settlement_status settlement_date settlement_amount  settlement_percentage  \
0               NaN             NaN               NaN                    NaN
1               NaN             NaN               NaN                    NaN
2               NaN             NaN               NaN                    NaN
3               NaN             NaN               NaN                    NaN
4               NaN             NaN               NaN                    NaN

   settlement_term
0              NaN
1              NaN
2              NaN
3              NaN
4              NaN

[5 rows x 145 columns]
```

```
In [4]: df_rej = pd.read_csv("./data2/RejectStats_2018Q3.csv", skiprows=[0], sep=',',na_values=
        df_rej.head()
```

```
Out[4]:    Amount Requested Application Date          Loan Title  Risk_Score  \
0            3000.0       2018-07-01  Debt consolidation         NaN
1           40000.0       2018-07-01      Major purchase         NaN
2           16000.0       2018-07-01  Debt consolidation         NaN
3           40000.0       2018-07-01  Debt consolidation         NaN
4          300000.0       2018-07-01       Business Loan         NaN

  Debt-To-Income Ratio Zip Code State Employment Length  Policy Code
0                 100%    925xx    CA          < 1 year            0
1                7.45%    335xx    FL          < 1 year            0
2               34.93%    156xx    PA          < 1 year            0
```

```
3              27.87%    957xx    CA          < 1 year            0
4                 -1%    258xx    TN          < 1 year            0
```

1.2 The road map to solve the problem

We now take a look at the column of funded amount (`funded_amnt`) and compare it with the requested amount (`loan_amnt`):

```
In [7]: np.sum(df_app['loan_amnt'] != df_app['funded_amnt'])

Out[7]: 2

In [8]: np.sum(df_app['loan_amnt'] != df_app['funded_amnt'])/df_app.shape[0]

Out[8]: 1.5601110799088896e-05
```

We see that there are only 2 instances where, if the request is approved, the requested amount differs from the actual amount funded. This accounts for a negligible portion of our huge data set (1.6×10^{-5}). We can safely assume that **the funded amount is equal to the requested amount**, given that the project is accepted. Hence, our overarching task of "predicting the largest loan amount that will be successfully funded for given individual" **boils down to the task of classifying whether an individual is approved or rejected**, as once the individual is accepted, the amount of fund is always equal to the amount requested (i.e., no need for regression for the amount funded). It is also worth pointing out that the cases where the requested amount does not equal the funded amount are those with missing data for those values, not because the funded amount is higher or lower than the requested one.

```
In [9]: pos = df_app['loan_amnt'] != df_app['funded_amnt']
        df_app[pos][['loan_amnt','funded_amnt']]

Out[9]:         loan_amnt  funded_amnt
        128194        NaN          NaN
        128195        NaN          NaN
```

1.3 Find common features

We find features shared by both types of data set. Below is the summary.

Reject	Approve
Amount Requested	loan_amnt
Loan Title	title
Debt-To-Income Ratio	dti
Zip Code	zip_code
State	addr_state
Employment Length	emp_length
Policy Code	policy_code

Therefore, we will drop all the columns that are not tabulated above.

```
In [10]: common_feat_app = ['loan_amnt', 'title', 'dti', 'zip_code',
                            'addr_state', 'emp_length','policy_code']
         common_feat_rej = ['Amount Requested', 'Loan Title',
                            'Debt-To-Income Ratio', 'Zip Code',
                           'State', 'Employment Length', 'Policy Code']

         # drop unshared columns
         df_app = df_app[common_feat_app]
         df_rej = df_rej[common_feat_rej]
         # shapes of new sets
         print('Approve shape:', df_app.shape)
         print('Reject shape:', df_rej.shape)
```

```
Approve shape: (128196, 7)
Reject shape: (2585245, 7)
```

1.4 Check missing values

We will check to see if there are missing values (and if so, how many) in each of the common features listed above.

```
In [11]: def check_na(df, feature_list):
             for ft in feature_list:
                 tot_missing = np.sum(pd.isna(df[ft]))
                 perc_missing = tot_missing/len(df)
                 print('%d %s missing, accounting for %.2f'%
                       (tot_missing, ft, perc_missing))

         # for approve data
         print('Approve data')
         print('-'*50)
         check_na(df_app, common_feat_app)
         print('\nReject data')
         print('-'*50)
         check_na(df_rej, common_feat_rej)
```

```
Approve data
--------------------------------------------------
2 loan_amnt missing, accounting for 0.00
2 title missing, accounting for 0.00
309 dti missing, accounting for 0.00
2 zip_code missing, accounting for 0.00
2 addr_state missing, accounting for 0.00
10389 emp_length missing, accounting for 0.08
2 policy_code missing, accounting for 0.00

Reject data
--------------------------------------------------
```

```
0 Amount Requested missing, accounting for 0.00
0 Loan Title missing, accounting for 0.00
0 Debt-To-Income Ratio missing, accounting for 0.00
0 Zip Code missing, accounting for 0.00
0 State missing, accounting for 0.00
104087 Employment Length missing, accounting for 0.04
0 Policy Code missing, accounting for 0.00
```

We see that except for the length of employment, all the features have a negligible number of missing values (accounting for virtually zero percent of the data sets). That said, we can discard any observations that have values for these features missing. A more concerning case is with the employment length, where $\approx$ 8% and $\approx$ 4% of the approve and reject data, repecstively, have missing values for this feature. What is worrying about disregarding these cases is that we don't know why they are missing. If we knew that they are missing at random, then discarding them would pose no considerable effect, especially given the small percentage the missing cases represent in the data sets. However, our analysis would be biased if there is a systematic difference between the missing cases and the other ones. To this end, we will impute the missing `emp_length` with its mean.

1.4.1 Drop missing values (except for `emp_length` and `Employment Length`)

```
In [12]: # drop df_app
         df_app_dropped = df_app.dropna(subset=['loan_amnt',
                                                'title',
                                                'dti',
                                                'zip_code',
                                                'addr_state',
                                                'policy_code'])
         # drop df_rej
         df_rej_dropped = df_rej # nothing to drop
         print('Shape of df_app after dropping:', df_app_dropped.shape)
         print('Shape of df_rej after dropping:', df_rej_dropped.shape)
```

```
Shape of df_app after dropping: (127887, 7)
Shape of df_rej after dropping: (2585245, 7)
```

```
In [13]: # change the column names of the reject data set to
         # match with those of the approve data set
         df_rej_dropped.columns = df_app_dropped.columns
         df_rej.head(2)
```

Out[13]:

	loan_amnt	title	dti	zip_code	addr_state	emp_length	policy_code
0	3000.0	Debt consolidation	100%	925xx	CA	< 1 year	0
1	40000.0	Major purchase	7.45%	335xx	FL	< 1 year	0

1.4.2 Compare values of two data sets

Now we check values of each feature in each data set to see if they are represented by the same coding.

1. loan_amnt

```
In [14]: print(df_app_dropped.loan_amnt.dtype)
         print(df_rej_dropped.loan_amnt.dtype)
```

```
float64
float64
```

2. title

```
In [15]: print(df_app_dropped.title.dtype)
         print(df_rej_dropped.title.dtype)
         print(df_app_dropped.title.unique())
         print(df_rej_dropped.title.unique())
```

```
object
object
['Credit card refinancing' 'Debt consolidation' 'Home improvement'
 'Medical expenses' 'Home buying' 'Major purchase' 'Other' 'Vacation'
 'Moving and relocation' 'Car financing' 'Business' 'Green loan']
['Debt consolidation' 'Major purchase' 'Business Loan'
 'Credit card refinancing' 'Other' 'Medical expenses' 'Car financing'
 'Green loan' 'Moving and relocation' 'Home buying' 'Home improvement'
 'Business' 'Vacation' 'home_improvement']
```

We see that all the categories presented in the approve data set is also mentioned in the rejected data set. The only differences are that some are capitalized and some are not (for example, 'other' and 'Other', some comtain words with blank spaces while other contain a sub bar (e.g, 'Home improvement' and 'home_imrovement'). For this reason, we lowercase all the categories and replace 'home_improvement' with 'home improvement'.

```
In [16]: df_app_dropped['title'] = df_app_dropped['title'].str.lower()
         df_rej_dropped['title'] = df_rej_dropped['title'].str.lower()
         df_app_dropped['title'] = df_app_dropped['title'].replace('home_improvement',
                                                                   'home improvement')
         df_rej_dropped['title'] = df_rej_dropped['title'].replace('home_improvement',
                                                                   'home improvement')
         df_rej_dropped['title'] = df_rej_dropped['title'].replace('business loan',
                                                                   'business')
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

```
See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
  """Entry point for launching an IPython kernel.
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
  after removing the cwd from sys.path.
```

3. dti

```
In [20]: df_rej_dropped['dti'] = df_rej_dropped['dti'].str.replace('%','')
         df_rej_dropped['dti'] = pd.to_numeric(df_rej_dropped['dti'])
         print(df_app_dropped.dti.dtype)
         print(df_rej_dropped.dti.dtype)
```

```
float64
float64
```

4. zip_code

```
In [21]: print(df_app_dropped.zip_code.dtype)
         print(df_rej_dropped.zip_code.dtype)
```

```
object
object
```

5. addr_state

```
In [22]: print(df_app_dropped.addr_state.dtype)
         print(df_rej_dropped.addr_state.dtype)
         print(df_app_dropped.addr_state.unique())
         print(df_rej_dropped.addr_state.unique())
```

```
object
object
['NJ' 'VA' 'AZ' 'NH' 'CA' 'WI' 'FL' 'TX' 'MA' 'NC' 'MD' 'MN' 'CO' 'NY'
 'MI' 'MO' 'WV' 'KY' 'IN' 'CT' 'LA' 'ME' 'IL' 'OH' 'GA' 'TN' 'NV' 'OK'
 'PA' 'ND' 'AR' 'UT' 'OR' 'HI' 'WA' 'SC' 'MS' 'AL' 'NE' 'DE' 'DC' 'RI'
 'KS' 'ID' 'MT' 'NM' 'VT' 'SD' 'WY' 'AK']
['CA' 'FL' 'PA' 'TN' 'AZ' 'HI' 'OR' 'KY' 'MT' 'ID' 'LA' 'IL' 'WI' 'MI'
 'WA' 'NY' 'AR' 'OH' 'TX' 'NV' 'NC' 'SC' 'MA' 'MO' 'NJ' 'CO' 'VA' 'MS'
 'OK' 'NM' 'WV' 'VT' 'MN' 'GA' 'AL' 'UT' 'KS' 'SD' 'DC' 'IN' 'NE' 'ND'
 'MD' 'ME' 'RI' 'NH' 'WY' 'CT' 'DE' 'AK' 'IA']
```

6. policy_code

```
In [23]: print(df_app_dropped.policy_code.dtype)
         print(df_rej_dropped.policy_code.dtype)
         print(df_app_dropped.policy_code.unique())
         print(df_rej_dropped.policy_code.unique())
         print(np.sum(df_rej_dropped['policy_code'] == 0)/len(df_rej_dropped))
```

```
float64
int64
[1.]
[0 2]
0.994885204303654
```

While the dictionary shows that the policy code can only take values of 1 or 2, the 0 values are present predominantly in the reject data set. Due to this conflict of information, we decided to exlude this feature from the models.

```
In [24]: df_app_dropped = df_app_dropped.drop(['policy_code'], axis=1)
         df_rej_dropped = df_rej_dropped.drop(['policy_code'], axis=1)
```

7. emp_length

```
In [25]: print(df_app_dropped.emp_length.dtype)
         print(df_rej_dropped.emp_length.dtype)
         print(df_app_dropped.emp_length.unique())
         print(df_rej_dropped.emp_length.unique())
```

```
object
object
['10+ years' '2 years' '4 years' '< 1 year' '8 years' nan '6 years'
 '3 years' '5 years' '7 years' '1 year' '9 years']
['< 1 year' nan '2 years' '1 year' '5 years' '9 years' '10+ years'
 '8 years' '3 years' '4 years' '7 years' '6 years']
```

Because there is a natural order of the employment length (e.g., 3 years are longer than 1 year), we will encode this feature as a numerical one.

```
In [26]: # approve data
         df_app_dropped['emp_length'] = df_app_dropped['emp_length'].replace('< 1 year',0)
         df_app_dropped['emp_length'] = df_app_dropped['emp_length'].replace('1 year',1)
         df_app_dropped['emp_length'] = df_app_dropped['emp_length'].replace('10+ years',10)
         for i in range(2,10):
             txt = str(i) + ' years'
             df_app_dropped['emp_length'] = df_app_dropped['emp_length'].replace(txt,
                                                                    i)
         # reject data
```

```
df_rej_dropped['emp_length'] = df_rej_dropped['emp_length'].replace('< 1 year',
                                                                    0)
df_rej_dropped['emp_length'] = df_rej_dropped['emp_length'].replace('1 year',
                                                                    1)
df_rej_dropped['emp_length'] = df_rej_dropped['emp_length'].replace('10+ years',
                                                                    10)
for i in range(2,10):
    txt = str(i) + ' years'
    df_rej_dropped['emp_length'] = df_rej_dropped['emp_length'].replace(txt,
                                                                    i)
```

1.4.3 Add labels

```
In [27]: df_app_dropped['rejected'] = 0
         df_rej_dropped['rejected'] = 1

In [28]: frames = [df_app_dropped, df_rej_dropped]
         df = pd.concat(frames)

In [29]: df.head()
```

Out[29]:

	loan_amnt	title	dti	zip_code	addr_state	emp_length	rejected
0	10000.0	credit card refinancing	19.22	070xx	NJ	10.0	0
1	15000.0	debt consolidation	25.60	245xx	VA	10.0	0
2	8500.0	debt consolidation	6.33	852xx	AZ	2.0	0
3	35000.0	debt consolidation	17.07	030xx	NH	4.0	0
4	28000.0	debt consolidation	13.24	958xx	CA	10.0	0

1.4.4 Impute `emp_length`

To impute the feature, we will use the mean of `emp_length`. We first split the data into training and testing (this is for later classification). Then we use the mean of the `emp_length` in the training data set to fill in the missing values in both training and testing sets.

Train-val-test split Before we do the splitting, we will one hot encode all the categoricals.

```
In [30]: df.head()
```

Out[30]:

	loan_amnt	title	dti	zip_code	addr_state	emp_length
0	10000.0	credit card refinancing	19.22	070xx	NJ	10.0
1	15000.0	debt consolidation	25.60	245xx	VA	10.0
2	8500.0	debt consolidation	6.33	852xx	AZ	2.0

```
3    35000.0       debt consolidation  17.07    030xx         NH         4.0
4    28000.0       debt consolidation  13.24    958xx         CA        10.0

   rejected
0         0
1         0
2         0
3         0
4         0
```

```
In [31]: df_encoded = pd.get_dummies(df,prefix=['title', 'zip_code', 'addr_state'],
                        columns=['title', 'zip_code', 'addr_state'])
```

```
In [32]: # dataframe train and test
         train, test = train_test_split(df_encoded, test_size=0.2)
```

Impute

```
In [34]: mean_emp_length = train['emp_length'].mean()
         train['emp_length'].fillna((mean_emp_length), inplace=True)
```

```
/anaconda3/lib/python3.6/site-packages/pandas/core/generic.py:5430: 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
  self._update_inplace(new_data)
```

```
In [35]: test['emp_length'].fillna((mean_emp_length), inplace=True)
```

```
/anaconda3/lib/python3.6/site-packages/pandas/core/generic.py:5430: 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
  self._update_inplace(new_data)
```

1.5 Feature Scaling

As I intend to use regularization in the logistic regression model, feature scaling is necessary. Currently the `loan_amnt` and `dti` differ each other by several orders of magnitude. We will scale both of them by substracting the mean and divide them by their respective standard deviations. To do this, (**overfitting**) we use the means and the standard deviations from the training set to apply to both sets, as this prevents info leakage (as opposed to using means and standard deviations of both training and testing sets).

```
In [36]: mean_loan_amnt = train['loan_amnt'].mean()
         sd_loan_amnt = train['loan_amnt'].std()
         mean_dti = train['dti'].mean()
         sd_dti = train['dti'].std()
```

```
In [37]: train['loan_amnt'] = (train['loan_amnt'] - mean_loan_amnt)/sd_loan_amnt
         train['dti'] = (train['dti'] - mean_dti)/sd_dti
         test['loan_amnt'] = (test['loan_amnt'] - mean_loan_amnt)/sd_loan_amnt
         test['dti'] = (test['dti'] - mean_dti)/sd_dti

/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
  """Entry point for launching an IPython kernel.
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html

/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
  This is separate from the ipykernel package so we can avoid doing imports until
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
  after removing the cwd from sys.path.


In [38]: X_train = train.drop(['rejected'], axis=1)
         y_train = train['rejected']

In [39]: X_test = test.drop(['rejected'], axis=1)
         y_test = test['rejected']

In [40]: X_train, X_val, y_train, y_val = train_test_split(X_train,
                                                           y_train,
                                                           test_size=.2)
```

2 Modeling and Evaluation

2.1 Downsample data (#modelmetric)

We have established that we will solve the problem by classification. If we think ahead how we will evaluate such a model, an appropriate metric to use is the accuracy. This is because there is no obvious reason for why we would try to avoid one error type more than the other. We assume

that a positive case is a rejected case. Type I error (false alarm) occurs when we predict a case that is approved as rejected. Sounds bad! Type II error (miss detection) happens when a case is rejected but we predict it as approved. Sounds equally bad! In decision making, the former error may make the applicant diffident and not apply for the loan and therefore miss the opportunity. Type II error may makes one overconfident and submit an application that will fail them. Neither one is an obvious worse case, so we will opt for **accuracy** as our metric to evaluate the model's performance. However, we can easily achieve a 95% accuracy by predicting every application that comes in the door as "rejected", because that is how our data distribution currently looks like (see the cell below), and our machine is worthless. To avoid this, we will downsample the class rejected (1) to make even proportions of the two classes. We only do this on the training set and not the validation set or test set, because: * If we downsampled on the test set, we would be assuming and forcing some kind of distribution on the unseen data, which makes our machine unrealistic. * If we downsampled on the val set, the validation error rate does not give a good estimate of the test error rate as the distribution of the data would be vastly different.

```
In [41]: print(np.sum(y_train)/len(y_train))
```

```
0.9530022967005375
```

```
In [42]: y_train_reset_index = y_train.reset_index(drop=True)
         X_train_reset_index = X_train.reset_index(drop=True)
         indices = np.where(y_train_reset_index == 1)[0]
         rng = np.random.RandomState(13)
         rng.shuffle(indices)
         n_neg = (y_train_reset_index == 0).sum()
         y_train_downsampled = y_train_reset_index.drop(y_train_reset_index.index[indices[n_neg
         X_train_downsampled = X_train_reset_index.drop(X_train_reset_index.index[indices[n_neg
         print(np.sum(y_train_downsampled)/len(y_train_downsampled))
```

```
0.5
```

The code above is cut off. The last 3 lines read:

```
y_train_downsampled = y_train_reset_index.drop(y_train_reset_index.index[indices[n_neg:]])
X_train_downsampled = X_train_reset_index.drop(X_train_reset_index.index[indices[n_neg:]])
print(np.sum(y_train_downsampled)/len(y_train_downsampled))
```

2.2 Logistic Regression

Some parameters to tune: * **penalty:** This is the regularization scheme to prevent the coefficients from being exploded. There are two schemes available in `sklearn`, L1 and L2. L1 is the sum of absolutes, while L2 is the sum of squares. * **C**: this is the inverse of regularization strength, the term that the regularization is multiplied by.

```
In [46]: from sklearn.linear_model import LogisticRegression
         from sklearn.metrics import roc_auc_score
         best_params = {'penalty': None, 'C': None}
         best_acc = -1
```

```
        i = 0
        for penalty in ['l1', 'l2']:
            for C in [1/1e-3, 1/1e-1, 1, 1/1e1, 1/1e3]:
                clf = LogisticRegression(penalty=penalty,
                                         C=C)
                clf.fit(X_train_downsampled, y_train_downsampled)
                preds = clf.predict(X_val)
                acc = np.sum(preds==y_val)/len(y_val)
                if acc > best_acc:
                    best_acc = acc
                    best_params['penalty'] = penalty
                    best_params['C'] = C
                i += 1
                print('%d finished'%i)
```

```
1 finished
2 finished
3 finished
4 finished
5 finished
6 finished
7 finished
8 finished
9 finished
10 finished
```

```
In [47]: print('best params:', best_params)
         print('best val accuracy:', best_acc)
```

```
best params: {'penalty': 'l2', 'C': 0.1}
best val accuracy: 0.9495578217972315
```

2.2.1 Training and testing with best parameters

```
In [50]: penalty, C = best_params.values()
         clf = LogisticRegression(penalty=penalty, C=C)
         clf.fit(X_train_downsampled, y_train_downsampled)
         preds = clf.predict(X_test)
         acc = np.sum(preds==y_test)/len(y_test)
         print('Testing accuracy:', acc)
```

```
Testing accuracy: 0.949685511410232
```

2.3 Interpretation

The testing accuracy is fairly high despite the very differeny distributions of data in the training set and testing set. (In training set, we downsample the class rejected so there is a 50/50 split

between two classes.) The testing accuracy is also very close to the best validation accuracy, which means our model does not overfit.

There are a few limitations to the model:

1. There is a cap to the latest time the data is relevant. That means current decisions based on this model risk being inaccurate due to any possible time-dependent variables.
2. When new/more recent data come in, we have to build the model from scratch. With logistic regression model, there is no scheme to augment new data to the learned model.
3. Imputation is done by taking the means. This may or may not be appropriate. For example, if the data is missing not at random and the reason it is missing is that because people are unemployed (i.e., employment length = 0) so they do not report it. Our way of taking a mean would systematically overestimate the employment length.
4. We used downsampling. This essentially assumes a uniform disitribution of the two classes. A classifer that does not need this assumption and still maintains good accuracy on testing data set may yield better results.

How to use the model to decide how much to apply: The coefficient for `loan_amnt` is $-0.077 < 0$. This means the model predicts that the higher we apply for, the lower the chance of getting accepted, assuming that all else are equal. The probability that an application will be accepted as a function of the amount requested takes the form:

$$p(X_1) = \frac{1}{1 + e^{A - 0.077 X_1}}$$

where A is the sum of other variables weighted by their coefficients. As the above function is continuous and decreasing monotonically as X_1 increases, we then can let X_1 varies increasingly until the probability hits a threshold/risk at which we can tolerate. We see that there is a tradeoff between the maximum loan funded and the chance of being accepted.