CS156-Assignment 3

Akmarzhan Abylay

October 2020

1 Assignment 3

1.1 Data Cleaning

In this part I downloaded both datasets and combined them. Then, I split them into training and test sets.

1.1.1 Step 1: Download the data

[1]:
```
import numpy as np
from datetime import datetime as dt
import matplotlib.pyplot as plt
```

[2]:
```
import pandas as pd

accepted = pd.read_csv('accepted.csv', low_memory=False)
rejected = pd.read_csv('rejected.csv', low_memory=False)
```

1.1.2 Step 2: Filter the data

We have too many data points (30 million), and my hardware is not good enough to run all operations on this many points, which is why I will only consider 1% of it, which is around 300000 points. I will randomly sample them from the pool. We will still hope that the results will be roughly the same, since we are sampling randomly.

[3]:
```
accepted = accepted.sample(frac=0.01, replace=False, random_state=1)
rejected = rejected.sample(frac=0.01, replace=False, random_state=1)
```

[4]:
```
print("Available columns in accepted:", len(list(accepted)))
accepted.head(2)
```

```
Available columns in accepted: 151
```

[4]:
```
                id  member_id  loan_amnt  funded_amnt  funded_amnt_inv  \
1557444  132555889        NaN    11000.0      11000.0          11000.0
1089926   69743499        NaN    12000.0      12000.0          12000.0

               term  int_rate  installment grade sub_grade  ...  \
```

```
1557444   36 months     10.90       359.61     B         B4  ...
1089926   36 months      6.99       370.48     A         A3  ...

        hardship_payoff_balance_amount hardship_last_payment_amount  \
1557444                            NaN                          NaN
1089926                            NaN                          NaN

        disbursement_method  debt_settlement_flag debt_settlement_flag_date  \
1557444           DirectPay                     N                       NaN
1089926                Cash                     N                       NaN

        settlement_status settlement_date settlement_amount  \
1557444               NaN             NaN               NaN
1089926               NaN             NaN               NaN

        settlement_percentage settlement_term
1557444                   NaN             NaN
1089926                   NaN             NaN

[2 rows x 151 columns]
```

[5]:
```
print("Available columns in rejected:", len(list(rejected)))
rejected.head(2)
```

```
Available columns in rejected: 9
```

[5]:
```
          Amount Requested Application Date              Loan Title  \
25221177           10000.0       2017-05-09      debt_consolidation
15860144           20000.0       2018-11-11  Credit card refinancing

          Risk_Score Debt-To-Income Ratio Zip Code State Employment Length  \
25221177         NaN               20.03%    922xx    CA          < 1 year
15860144         NaN                18.6%    484xx    MI          < 1 year

          Policy Code
25221177          0.0
15860144          0.0
```

As we see, we have many more columns in the accepted dataset, and the encoding is very different. We will manually pick the columns and change their names, depending on what suits them better. I will only use the columns that will also be applicable to the rejected dataset. The only corresponding column I couldn't find is the column for the risk score.

[6]:
```
acc = accepted[["loan_amnt", "issue_d",  "title",
                    "dti", "zip_code", "addr_state", "emp_length",␣
 ↪"policy_code"]]
rej = rejected.drop(columns="Risk_Score")
```

```
rej.columns = ["loan_amnt", "issue_d",  "title",
                     "dti", "zip_code", "addr_state", "emp_length", "policy_code"]
rej.head()
```

[6]:
```
          loan_amnt     issue_d                     title     dti  zip_code  \
25221177    10000.0  2017-05-09        debt_consolidation  20.03%     922xx
15860144    20000.0  2018-11-11  Credit card refinancing   18.6%     484xx
6035463      5800.0  2018-08-27        Debt consolidation  15.79%     906xx
12636787    11450.0  2017-08-19             Car financing    100%     754xx
6076264      3000.0  2018-08-29        Debt consolidation   7.67%     916xx

         addr_state emp_length  policy_code
25221177         CA   < 1 year          0.0
15860144         MI   < 1 year          0.0
6035463          CA   < 1 year          0.0
12636787         TX   < 1 year          0.0
6076264          CA   < 1 year          0.0
```

[7]:
```
acc.head()
```

[7]:
```
         loan_amnt   issue_d                    title    dti zip_code  \
1557444    11000.0  May-2018  Credit card refinancing  16.10    169xx
1089926    12000.0  Jan-2016       Debt consolidation  19.84    773xx
71683       5000.0  Nov-2015       Debt consolidation  16.95    490xx
356146     20000.0  Apr-2015  Credit card refinancing  34.56    900xx
1396198     5000.0  Nov-2018       Debt consolidation  35.58    330xx

        addr_state emp_length  policy_code
1557444         PA    8 years          1.0
1089926         TX    9 years          1.0
71683           MI     1 year          1.0
356146          CA    8 years          1.0
1396198         FL  10+ years          1.0
```

1.1.3 Step 3: Clean the data

Let's get rid of NA values. We also should be careful if NAs make up more than a significant amount of data (e.g., 10%). Then, we might need to work on our missing values more and replace them with other values depending on our assumptions. We are fine with our example.

[8]:
```
pd.options.mode.chained_assignment = None #ignore warnings

print("# rows before NA drop:", acc.shape[0])
acc.dropna(how='any', axis=0, inplace=True) #removing NAs
print("# rows after NA drop:", acc.shape[0])
```

```
# rows before NA drop: 22607
# rows after NA drop: 20864
```

[9]:
```
print("# rows before NA drop:", rej.shape[0])
rej.dropna(how='any', axis=0, inplace=True) #removing NAs
print("# rows after NA drop:", rej.shape[0])
```

```
# rows before NA drop: 276487
# rows after NA drop: 266905
```

Our date is in the form of month-year for the accepted data and year-month-day for the rejected data. We can rearrange it and assume that the day for the accepted date is 15th (on average, uniform distribution). We could have also deleted the day from the rejected data, but we don't want to lose information.

[10]:
```
acc.issue_d = list(map(lambda date: dt.strptime("15-" + str(date), '%d-%b-%Y'),
                                             acc.issue_d))
acc.head()
```

[10]:
```
         loan_amnt    issue_d                   title    dti zip_code  \
1557444    11000.0 2018-05-15  Credit card refinancing  16.10    169xx
1089926    12000.0 2016-01-15       Debt consolidation  19.84    773xx
71683       5000.0 2015-11-15       Debt consolidation  16.95    490xx
356146     20000.0 2015-04-15  Credit card refinancing  34.56    900xx
1396198     5000.0 2018-11-15       Debt consolidation  35.58    330xx

        addr_state emp_length  policy_code
1557444         PA    8 years          1.0
1089926         TX    9 years          1.0
71683           MI     1 year          1.0
356146          CA    8 years          1.0
1396198         FL  10+ years          1.0
```

We also can see that the DTI ratio for the accepted data is given in plain numbers, while the numbers for the rejected data are given in percentages. We can just remove the percentage sign and convert our value to float. Also, our dates are just strings, which is why we need to transfer them to DateTime.

[11]:
```
rej.dti = list(map(lambda dti: float(dti[:-1]), rej.dti))
rej.issue_d = list(map(lambda date: dt.strptime(str(date), '%Y-%m-%d'),
                                             rej.issue_d))
rej.head()
```

[11]:
```
          loan_amnt    issue_d                   title     dti zip_code  \
25221177    10000.0 2017-05-09      debt_consolidation   20.03    922xx
15860144    20000.0 2018-11-11 Credit card refinancing   18.60    484xx
6035463      5800.0 2018-08-27      Debt consolidation   15.79    906xx
12636787    11450.0 2017-08-19           Car financing  100.00    754xx
6076264      3000.0 2018-08-29      Debt consolidation    7.67    916xx
```

```
          addr_state emp_length  policy_code
25221177          CA   < 1 year          0.0
15860144          MI   < 1 year          0.0
6035463           CA   < 1 year          0.0
12636787          TX   < 1 year          0.0
6076264           CA   < 1 year          0.0
```

[12]:
```python
acc.head()
```

[12]:
```
         loan_amnt    issue_d                    title    dti zip_code  \
1557444    11000.0 2018-05-15  Credit card refinancing  16.10    169xx
1089926    12000.0 2016-01-15       Debt consolidation  19.84    773xx
71683       5000.0 2015-11-15       Debt consolidation  16.95    490xx
356146     20000.0 2015-04-15  Credit card refinancing  34.56    900xx
1396198     5000.0 2018-11-15       Debt consolidation  35.58    330xx

         addr_state emp_length  policy_code
1557444          PA    8 years          1.0
1089926          TX    9 years          1.0
71683            MI     1 year          1.0
356146           CA    8 years          1.0
1396198          FL  10+ years          1.0
```

Now that we have looked through our data, we made sure all of the columns match. If we inspect the columns `policy_code` column is plain 1's for the accepted data and 0 or 2 for the rejected data, which means that it is correlated with the approval status. We can just change the policy code to represent whether the loan is approved or not.

[13]:
```python
print(acc.policy_code.unique())
print(rej.policy_code.unique())
```

```
[1.]
[0. 2.]
```

[14]:
```python
acc.rename(columns = {'policy_code':'approved'}, inplace = True)
rej.rename(columns = {'policy_code':'approved'}, inplace = True)
rej['approved'] = 0.
```

1.1.4 Step 4: Combining the data

[15]:
```python
data = pd.concat([acc, rej])
data.head()
```

[15]:
```
         loan_amnt    issue_d                    title    dti zip_code  \
1557444    11000.0 2018-05-15  Credit card refinancing  16.10    169xx
1089926    12000.0 2016-01-15       Debt consolidation  19.84    773xx
```

```
71683        5000.0 2015-11-15       Debt consolidation  16.95     490xx
356146      20000.0 2015-04-15  Credit card refinancing  34.56     900xx
1396198      5000.0 2018-11-15       Debt consolidation  35.58     330xx

        addr_state emp_length  approved
1557444         PA    8 years       1.0
1089926         TX    9 years       1.0
71683           MI     1 year       1.0
356146          CA    8 years       1.0
1396198         FL  10+ years       1.0
```

We also see that all zipcodes have an "xx" in the end, so we can just remove it. The employment length can be encoded through ascending numbers (ordinal encoding), since they have a numerical meaning.

[16]:
```
data.zip_code = data. zip_code.str.slice(stop=3)
data.emp_length.unique()
```

[16]:
```
array(['8 years', '9 years', '1 year', '10+ years', '3 years', '4 years',
       '< 1 year', '2 years', '7 years', '6 years', '5 years'],
      dtype=object)
```

[17]:
```
#I could have done it with scikit, but I wanted to try doing it with pandas
data.emp_length.replace({'< 1 year': 0, '1 year': 1, '2 years': 2,
                         '3 years': 3, '4 years': 4, '5 years': 5,
                         '6 years': 6, '7 years': 7, '8 years': 8,
                         '9 years': 9, '10+ years': 10}, inplace = True)
data.head()
```

[17]:
```
         loan_amnt    issue_d                    title    dti zip_code  \
1557444    11000.0 2018-05-15  Credit card refinancing  16.10      169
1089926    12000.0 2016-01-15       Debt consolidation  19.84      773
71683       5000.0 2015-11-15       Debt consolidation  16.95      490
356146     20000.0 2015-04-15  Credit card refinancing  34.56      900
1396198     5000.0 2018-11-15       Debt consolidation  35.58      330

        addr_state  emp_length  approved
1557444         PA           8       1.0
1089926         TX           9       1.0
71683           MI           1       1.0
356146          CA           8       1.0
1396198         FL          10       1.0
```

1.1.5 Step 5: Preprocessing

Now, we need to deal with the categorical variables from the other columns. Since they don't have a numerical meaning, if we just encode them as different numbers representing the categories, it wouldn't make sense and the model will think that a higher number is necessarily more significant numerically, which is not true. We will use dummy variables for easier analysis. Also, we will need to change the DateTime type to ordinal numbers for the date since otherwise, the scikit won't work.

[18]:
```
data['issue_d'] = pd.to_datetime(data['issue_d'])
data['issue_d'] = data['issue_d'].map(dt.toordinal)
```

[19]:
```
print("Unique number of titles: ", len(data.title.unique()))
print("Unique number of zip codes: ", len(data.zip_code.unique()))
print("Unique number of states: ", len(data.addr_state.unique()))
```

```
Unique number of titles:  2076
Unique number of zip codes:  933
Unique number of states:  51
```

We have three columns with categorical variables: `title`, `zip_code` and `addr_state`. The first two have a very large number of unique categories (over 100000+ and 1888), which is why we will leave the most common 20 and categorize others as one category "OTHER". I will then make dummies for each category.

[20]:
```
top_title = [cat for cat in data.title.value_counts().sort_values(ascending =␣
 ↪False).head(20).index]
data.title = np.where(data.title.isin(top_title), data.title, 'OTHER')
```

[21]:
```
top_zip = [cat for cat in data.zip_code.value_counts().sort_values(ascending =␣
 ↪False).head(20).index]
data.zip_code = np.where(data.zip_code.isin(top_zip), data.zip_code, 'OTHER')
```

[22]:
```
print("New unique titles: ", len(data.title.unique()))
print("New unique zip_codes: ", len(data.zip_code.unique()))
```

```
New unique titles:  21
New unique zip_codes:  21
```

[23]:
```
#we could also do it with OneHotEncoder but it is easier
#to do it with pandas

dum_title = pd.get_dummies(data.title)
dum_zip = pd.get_dummies(data.zip_code)
dum_state = pd.get_dummies(data.addr_state)

data = pd.concat((data, dum_title, dum_zip, dum_state), axis = 1)
data.drop(['title','zip_code','addr_state'], axis = 'columns', inplace = True)

data.head()
```

[23]:

```
          loan_amnt  issue_d    dti  emp_length  approved  Business  \
1557444     11000.0   736829  16.10           8       1.0         0
1089926     12000.0   735978  19.84           9       1.0         0
71683        5000.0   735917  16.95           1       1.0         0
356146      20000.0   735703  34.56           8       1.0         0
1396198      5000.0   737013  35.58          10       1.0         0

         Business Loan  Car financing  Credit card refinancing  \
1557444              0              0                        1
1089926              0              0                        0
71683                0              0                        0
356146               0              0                        1
1396198              0              0                        0

         Debt consolidation  ...  SD  TN  TX  UT  VA  VT  WA  WI  WV  WY
1557444                   0  ...   0   0   0   0   0   0   0   0   0   0
1089926                   1  ...   0   0   1   0   0   0   0   0   0   0
71683                     1  ...   0   0   0   0   0   0   0   0   0   0
356146                    0  ...   0   0   0   0   0   0   0   0   0   0
1396198                   1  ...   0   0   0   0   0   0   0   0   0   0

[5 rows x 98 columns]
```

We need to split our data into training and testing data, since we don't want to use all of our data to train the model and then assess the accuracy on the same data, because of data leakage.

[24]:

```python
X = data.iloc[:, data.columns != 'approved'].values
y = data.approved.values
```

[25]:

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 0)
```

I will be working with a KNN classifier, and since there are different variances throughout the variables (i.e., loan_amnt with a large variance and dti with a small variance), we need to rescale the features.

For the KNN, the common dissimilarity is often based on the Euclidean distance. This means that if we have different scales for the features, then some of them might end up skewing our classification. To prevent this, and ensure that all features are equally important, we need to rescale the data.

[26]:

```python
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

1.2 Modeling and Evaluation

I will use the KNN first to predict whether a loan should be accepted or rejected and determine how much the bank should give. If the loan is rejected, the maximum loan amount is 0, so I will classify whether the loan is accepted/rejected first.

1.2.1 Step 1: Methods

As we see from the printed statements below, our data is HIGHLY skewed. Almost 93% of it is rejected, which means that if we just categorize everything as rejected, we will be, on average, 93% correct. We also care more about whether an applicant is a false negative since we don't want to give out loans to those people who are actually not eligible (risk-averse). That's why we will assess the specificity $\frac{TN}{TN+FP}$. The higher this number, the safer we are, since we make less False Positive classifications, where we say that the person is eligible for the loan they are actually not eligible for. This means that:

- The baseline accuracy is 93%. Everything below this performs worse than if we just classified everything as a rejection.
- Metrics: accuracy (additionally, specificity and ROC curve area).

[27]:
```
print("Accepted loans:", sum(y))
print("Fraction of accepted loans:", sum(y)/len(y))
print("Rejected loans:", len(y)-sum(y))
print("Fraction of rejected loans:", (len(y)-sum(y))/len(y))
print("Total number of data points:", len(y))
```

```
Accepted loans: 20864.0
Fraction of accepted loans: 0.07250259756957837
Rejected loans: 266905.0
Fraction of rejected loans: 0.9274974024304217
Total number of data points: 287769
```

[28]:
```
from sklearn.metrics import accuracy_score, roc_auc_score, confusion_matrix

def performance(y_test, y_pred):
    accuracy = accuracy_score(y_pred, y_test)
    auroc = roc_auc_score(y_pred, y_test)

    tn, fp, fn, tp = confusion_matrix(y_pred, y_test).ravel()
    specificity = tn / (tn + fp)

    return accuracy, auroc, specificity
```

1.2.2 Step 2: Grid Search

We can use GridSearchCV to perform an exhaustive search over the parameter values (i.e., number of neighbors). We can use different scoring techniques, as described above. I will use the traditional accuracy for our case. The function also is optimized through cross-validation, where the data is divided into k folds, and k-1 sections are used to train the model, and the left out section is used as a validation set. In CV, this is then done k times, and we calculate the average accuracy to get a better grasp of the test set error estimate. Cross-validation is usually used to reduce overfitting.

However, we have 300,000 data points, which means that it will take a lot of time to run all points. This is why we will only use 10% of the data -> around 30000 points, which is still a lot. We will assume that the results will be accurate since we sampled randomly.

[29]:
```
integers = np.random.choice(len(X_train), size=30000, replace=False)
X_frac = X_train[integers]
y_frac = y_train[integers]
```

[30]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
parameters = [{'n_neighbors': [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]}]
grid_search = GridSearchCV(estimator = KNeighborsClassifier(),
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = 10)

grid_search = grid_search.fit(X_frac, y_frac)
best_accuracy = grid_search.best_score_
best_parameters = grid_search.best_params_
print("Best Accuracy: {:.2f} %".format(best_accuracy*100))
print("Best Parameters:", best_parameters)
```

```
Best Accuracy: 94.92 %
Best Parameters: {'n_neighbors': 7}
```

We have an accuracy rate of around 95% for n_neighbors=3, which is better than classifying everything as rejection. We could have also made the plot for all three rates we are interested in: ROC area, accuracy, specificity. Depending on which ones have the best results, our number of neighbors could have changed. However, this was the easiest method to implement in this case. Now, let's assess the test data accuracy rate, ROC curve, and specificity rate.

1.2.3 Step 3: Test Set Accuracy

Since we found the best number of neighbors, we can now train our KNN with 3 neighbors on the whole train data set.

[31]:
```
classifier = KNeighborsClassifier(n_neighbors = 7)
classifier.fit(X_train, y_train)
```

[31]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=7, p=2,
                     weights='uniform')
```

[32]:
```
y_pred = classifier.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[52155  1246]
 [ 1298  2855]]
```

[33]:
```
accuracy_score(y_test, y_pred)
```

[33]:
```
0.9557980331514752
```

[34]:
```
performance(y_test, y_pred)
```

[34]:
```
(0.9557980331514752, 0.8359443252311625, 0.9757169850148729)
```

We got pretty good test set results. Around 96% is better than just classifying everything as rejection. We got a specificity of 97.6%, which is really good. Since specificity measures the proportion of negatives that are correctly identified, this means that we are pretty good in not getting False Positives. The ROC curve represents the plot of the true positives against the false positive rate. We got around 0.836, which means that our performance is quite good.

[37]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, labels=[0,1]))
```

```
              precision    recall  f1-score   support

           0       0.98      0.98      0.98     53401
           1       0.70      0.69      0.69      4153

    accuracy                           0.96     57554
   macro avg       0.84      0.83      0.83     57554
weighted avg       0.96      0.96      0.96     57554
```

Given that we have very skewed data, we have pretty good precision and recall for the accepted loans.

1.2.4 Step 4: Finding Max Loan

Now that we obtained a reasonably accurate loan approval classifier, we can move on to the next step of identifying what the max amount of loan a person could receive is. I am going to focus only on people who got approved and test how much more they could get. I am going to gradually increase the amount of money they would have requested (by $1) to see whether the new amount would still be approved for a loan. When the amount is no longer approved, I will reobtain the previous largest amount of loan (as a whole number), which was still approved. This will give us the max amount of the loan approved.

```
[39]: import _pickle as cPickle #I am not sure why but my code for the max loan␣
    ↪function didn't want to work until I did this
with open('classifier.pkl', 'wb') as fid:
    cPickle.dump(classifier, fid)
```

I took an arbitrary example of an approved loan, and fed it into the max loan finder function. It will work with any other example too.

```
[40]: print(sc.inverse_transform(X_test[20])[0])
print(classifier.predict([X_test[20]]))
```

```
5400.0
[1.]
```

```
[41]: import copy

def max_loan(inp, step):
    global classifier
    global sc

    amount = []
    inpo = copy.deepcopy(inp) #so that they don't get meesed up and
    #the actual input valued doesn't get changed

    if classifier.predict(sc.transform(inp))[0]==1: #only works if the loan is␣
    ↪approved initially
        while True: #iterated until the loan is rejected
            new = inpo[0][0]+step #new loan amount
            inpo[0][0] = new
            if classifier.predict(sc.transform(inpo))[0]==0: #classify and find␣
    ↪whether the loan is rejected
                break #if yes, then the previous loan amount was the biggest␣
    ↪that was approved
    value = inpo[0][0]-step

    if value<=0:
        print("Unfortunately, your loan wouldn't be accepted no matter of the␣
    ↪loan amount.")
    else:
        print ("Congratulations, your loan can be accepted for a maximum of␣
    ↪${n}".format(n=value))

max_loan([sc.inverse_transform(X_test[20])], 1)
```

```
Congratulations, your loan can be accepted for a maximum of $6150.0
```

As we could see from the code above, it is fairly easy to see what the maximum loan amount is. It is accurate to the whole numbers. Although it takes some time to run, we can change the step size and arrive at rougher estimates but still within an adequate range. I took an arbitrary test case where the loan is approved since we are only concerned with the approved loans, as rejected loans automatically have a maximum loan amount of 0, and people tend to not want to get loans less than what they wrote in the initial application.

Based on our algorithm, the maximum amount that test person 21 can get is `$6150`, whereas the amount initially stated in the application was `$5400`. This means that the person could have requested `$750` more and still be accepted for the loan. We can check that this applies below, where `$6150` is accepted, and `$6151` is rejected.

[42]:
```
X1 = sc.inverse_transform(X_test[20])
X1[0] = 6150

print(classifier.predict(sc.transform([X1])))
```

```
[1.]
```

[43]:
```
X1 = sc.inverse_transform(X_test[20])
X1[0] = 6151

print(classifier.predict(sc.transform([X1])))
```

```
[0.]
```

We can look at the classifier's accuracy and other metrics to assess how accurate our maximum loan finder algorithm is. Since we have obtained fairly good results and especially our specificity rate is very high (97.5%), we can say that this algorithm is pretty good at preventing False Positives. There is a low possibility that if the person is said to be accepted, they actually should be rejected, which is why our max loan finder is pretty good.

Also, the maximum loan amount can vary throughout time, but I am considering only specific points in time where everything is held constant. This is obviously an over-simplified model and algorithm, which wouldn't necessarily reflect reality, so it should serve as a rough estimate for the maximum loan amount.

I didn't know how else we can predict the maximum loan amount, so this was the only way I could think of. I am pretty sure there are other ways, which are more convenient and accurate.

In general, we could also try to run it on the full data set of 30 million data points, if we had the compatible hardware.