Capstone Project 1: Predicting Loan Defaults for Lending Club

Ameena Khan
Final Report

Background

Lending Club is an online peer-to-peer loan platform that allows individuals to take out personal loans of up to $40,000. Borrowers can easily apply for a loan online and will typically receive their money within a few days of submitting their application. Unlike a bank, the platform uses investors to fund loans and acts as the intermediary between investors and borrowers.

When an application is received and approved by Lending Club, it goes into their online platform. Investors can then select which loans to fund by either manually picking specific loans or having Lending Club automatically select a loan portfolio for them. As borrowers pay back their loans, investors receive monthly payments for the principal and interest on each loan.

Occasionally a borrower does not pay back a loan in full and Lending Club must "Charge Off" the loan. This typically happens once a loan payment is at least 150 days past due but can also occur earlier or later depending on the circumstances (i.e. a borrower files for bankruptcy). If a borrower charges off (AKA defaults) on their loan, there is limited recourse for an investor and the default can impact their return on investment.

Problem

Investors lose a significant amount of their potential earnings to loan defaults. As an example, Lending Club explains that a portfolio expecting to make 14% in annual interest, will lose approximately 8% due to defaults. After accounting for Lending Club's 1% fee, an investor can expect to make 5% annually on their investments.

Is there a way to further maximize the return on investment for an investor while decreasing the amount of interest lost to loan defaults?

In this project I will explore how much Lending Club loses to charged off loans and whether it is possible to create a model that predicts the risk of a specific borrower failing to pay off their loan.

Data

I will be using Lending Club's dataset that contains loan information from 2007-2011.

The original dataset includes over 140 features; however, I only want to focus on the information relevant to a borrower's application since that is what will be used to determine whether or not to reject an applicant.

With that in mind, I narrowed the dataset down to the following 16 features of interest:

- **Funded Amount:** The amount loaned to the borrower
- **Term:** The length of the loan (either 36 months or 60 months)
- **Interest Rate:** Interest rate on the loan
- **Installment:** Loan payments
- **Grade:** Lending Club assigned loan grade
- **Sub Grade:** Lending Club assigned loan sub-grade
- **Employment Title:** The job title supplied by the borrower when applying for a loan
- **Employment Length:** Borrowers length of employment
- **Home Ownership:** Home ownership status provided by borrower: RENT, OWN, MORTGAGE, OTHER
- **Annual Income:** Annual income provided by borrower
- **Verification Status:** Indicates if income was verified by LC
- **Issue Date:** Month and year the loan was issued
- **Loan Status:** Lists whether a loan is CURRENT or CHARGED OFF - this will be the predicted variable in my model
- **Purpose of Loan:** Purpose of loan provided by borrower
- **State of Borrower:** State of residence provided by borrower
- **DTI:** Debt to income ratio calculated using borrower's total monthly debt payments on the total debt obligations, divided by borrower's self-reported annual income

Cleaning Data

Missing Data

Overall the dataset from Lending Club was relatively clean and required minimal updates to prepare it for modeling.

The first thing I did was replace missing information as follows:

- **Employment Title:** Missing 2624 entries. Replaced all missing information with 'Unknown'. I also phad several titles with less than 20 counts, so I reclassified those as 'Other'.
- **Annual Income:** Four entries were missing income data so I replaced those with the mean annual income of $69,136.56.

```
# Replace NaN in Employment Title with 'Unknown'
df['emp_title'] = df['emp_title'].fillna('Unknown')

# Replace values of < 20 with 'Other'
df = df.assign(emp_title=df.groupby('emp_title')['emp_title'].transform(lambda x: x if x.size>=25 else 'Other'))

# Calculate the mean of annual_inc
inc_mean = df['annual_inc'].mean()

# Replace all the missing values in annual_inc with the mean annual income
df['annual_inc'] = df['annual_inc'].fillna(inc_mean)
```

Date Issued

In case I wanted to further explore the month and year that a loan was issued, I decided to create two additional columns:

- **Month Issued:** the month a loan was issued
- **Year Issued:** the year a loan was issue

```
# Split issue_d into separate columns with the year and month in each column
df[['issue_year','issue_month']] = df['issue_d'].str.split('-',expand=True)

# replace the incorrect years with the correct year format
df['issue_year'] = df['issue_year'].replace({'7': '07',
                                              '8':'08',
                                           '9':'09'})

# drop issue_d column
df = df.drop('issue_d', 1)
```

Loan Status

Since loan status is what I will be using as my independent variable throughout this project, I decided to turn it into a binomial variable as follows:

- Fully Paid: 0
- Charged Off: 1

Note that Fully Paid means that the loan is currently up to date with all payments and is in good standing. It does not necessarily mean that the loan has been repaid in full.

```
# consolidate loan status into two catagories
df['loan_status'] = df['loan_status'].replace({'Does not meet the credit policy. Status:Fully Paid':'Fully Paid',
                                              'Does not meet the credit policy. Status:Charged Off':'Charged Off'})

# convert fully paid to 0 and charged off to 1
df['loan_status'] = df['loan_status'].replace({'Fully Paid':0,
                                              'Charged Off':1})
```

Overview of Each Feature:

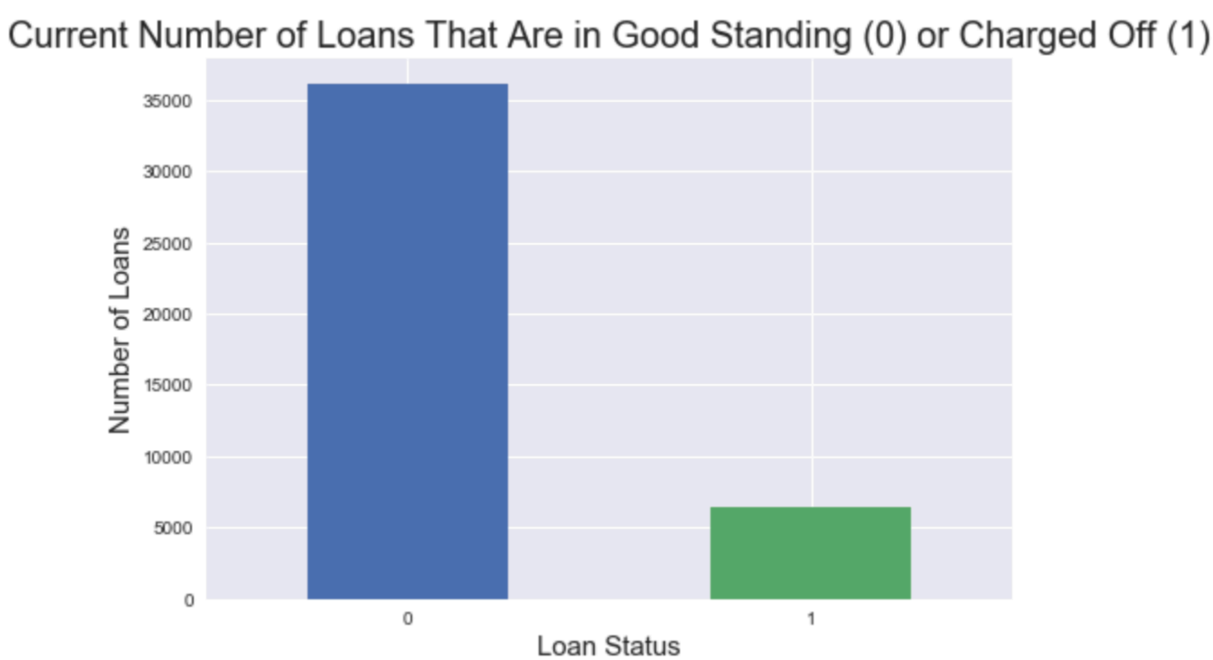
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 42535 entries, 0 to 42537
Data columns (total 18 columns):
funded_amnt            42535 non-null float64
funded_amnt_inv        42535 non-null float64
term                   42535 non-null object
int_rate               42535 non-null float64
installment            42535 non-null float64
grade                  42535 non-null object
sub_grade              42535 non-null object
emp_title              42535 non-null object
emp_length             41423 non-null object
home_ownership         42535 non-null object
annual_inc             42535 non-null float64
verification_status    42535 non-null object
loan_status            42535 non-null int64
purpose                42535 non-null object
addr_state             42535 non-null object
dti                    42535 non-null float64
issue_year             42535 non-null object
issue_month            42535 non-null object
dtypes: float64(6), int64(1), object(11)
memory usage: 6.2+ MB
```

Descriptive Statistics:

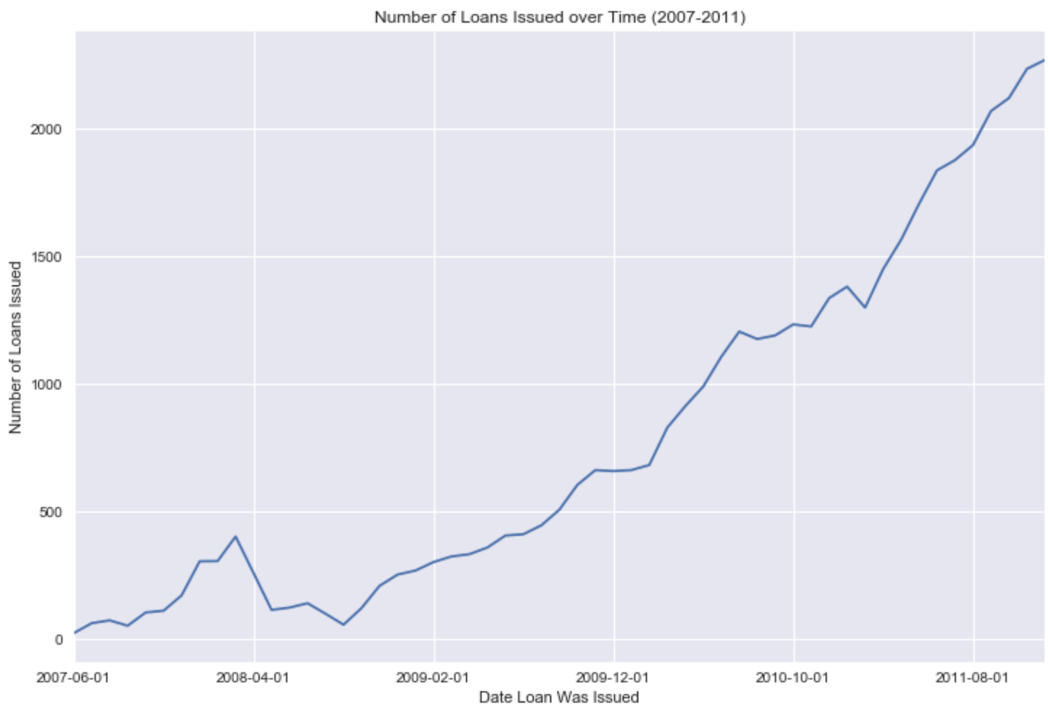
	funded_amnt	funded_amnt_inv	int_rate	installment	annual_inc	loan_status	dti
count	42535.000000	42535.000000	42535.000000	42535.000000	4.253500e+04	42535.000000	42535.000000
mean	10821.585753	10139.830603	12.165016	322.623063	6.913656e+04	0.151193	13.373043
std	7146.914675	7131.686447	3.707936	208.927216	6.409334e+04	0.358241	6.726315
min	500.000000	0.000000	5.420000	15.670000	1.896000e+03	0.000000	0.000000
25%	5000.000000	4950.000000	9.630000	165.520000	4.000000e+04	0.000000	8.200000
50%	9600.000000	8500.000000	11.990000	277.690000	5.900000e+04	0.000000	13.470000
75%	15000.000000	14000.000000	14.720000	428.180000	8.250000e+04	0.000000	18.680000
max	35000.000000	35000.000000	24.590000	1305.190000	6.000000e+06	1.000000	29.990000

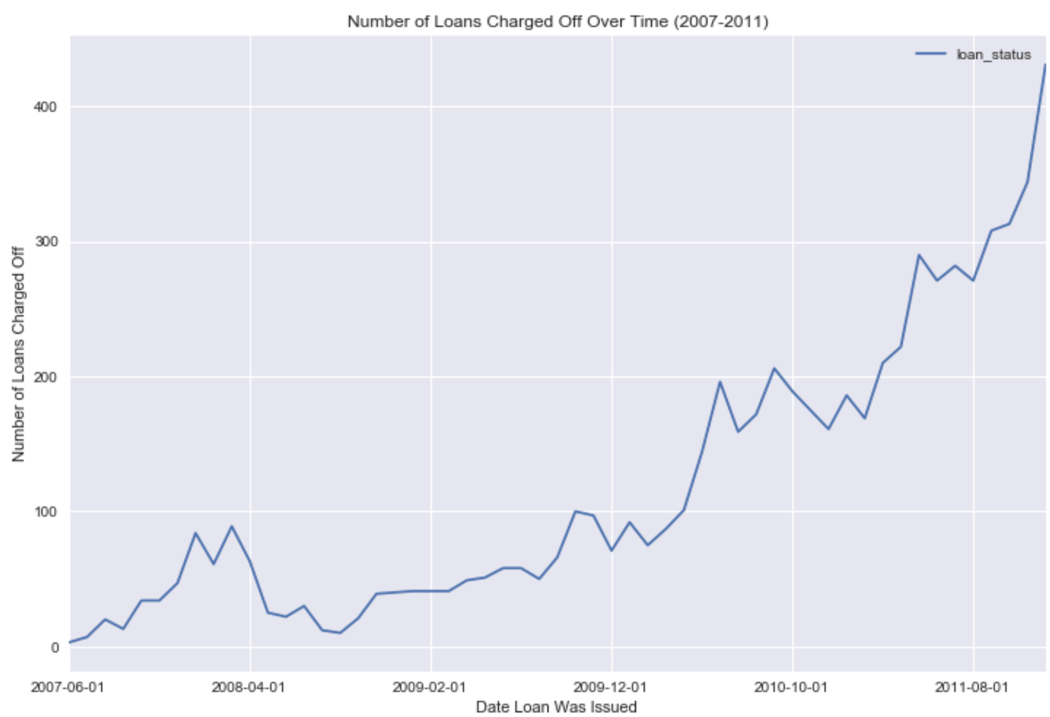
Exploratory Data Analysis

From 2007-2011, Lending Club issued over $460 million dollars in loans. Of the 42,535 loans issued during that time, 15.1 percent of them were charged off. These loans totaled over $73.9 million dollars. While this amount does not consider how much a borrower repaid before the loan was charged off or how much money Lending Club investors will lose in interest that would have been paid on the loan, it's safe to say that is still a lot of money Lending Club investors are losing!



Trends Over Time





Looking at the graphs, it appears the number of loans that are charged off has remained proportionally consistent over time. Additional statistical analysis will allow me to see if there is a more significant relationship here.

Machine Learning

Pre-Processing

In order to prepare the data for the machine learning portion of my project, I performed some additional pre-processing steps.

First, I wrote some code that would allow me to quickly drop features from my analysis in order to test which combination of features would produce the strongest model. I did this by first creating a list with all of the feature names. If I wish to include a feature in my model, I added a hash to the beginning of that line of code. Any un-hashed features will be dropped and not included in model.

```
# drop feature columns that should be excluded from the model
drop = [
#    'funded_amnt',
#    'funded_amnt_inv',
#    'term',
#    'int_rate',
#    'installment',
    'grade',
    'sub_grade',
    'emp_title',
#    'emp_length',
#    'home_ownership',
#    'annual_inc',
#    'verification_status',
    'loan_status',
    'purpose',
    'addr_state',
#    'dti',
    'issue_year',
#    'issue_month'
]

# categorical features used in the model, to be converted later
dummies = [
    'term',
#    'grade',
#    'sub_grade',
#    'emp_title',
    'emp_length',
    'home_ownership',
    'verification_status',
#    'purpose',
#    'addr_state',
#    'issue_year',
    'issue_month'
]
```

I then created the 'y' variable which contains the 'loan status' variable since that is what I am trying to predict with my model. Once the y variable was created, I dropped all of the features that were un-hashed and double checked that the 'loan status' column was dropped.

```
# store the labels for prediction
y = df['loan_status'].values
```

```
# drop features that will not be used in the model
df_prep = df.drop(columns=drop)
df_prep.info()
```

I then converted the categorical features into dummy variables and created my 'X' variable which will be used to train and test the model.

```
# Create dummy columns
df_prep = pd.get_dummies(data=df_prep, columns=dummies)
```

```
X = df_prep.values
```

Finally, I used Scikit-Learn's train_test_split to create the variables I will use to train and test the model.

```
# create train and test data sets for model analysis
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

```
print('Test shape:   ', X_test.shape[0])
print('Dimensions:', X_test.shape[1])
```

```
Test shape:    8507
Dimensions: 39
```

The data is now ready for modeling!

Random Forest Classifier

The first model I used to predict whether an applicant will default on their loan is the Random Forest Classifier. A Random Forest is an ensemble method of machine learning that uses several models at once to classify an outcome. It gets its name because it uses a large number of independent decision trees in order to optimize for the strongest performing model.

In order to properly evaluate the performance of my model, I first ran the Random Forest Classifier out of the box in order to understand its baseline performance. I ran it multiple times with different features selected using my drop method outlined in the pre-processing section.

These are my results from my strongest performing baseline model:

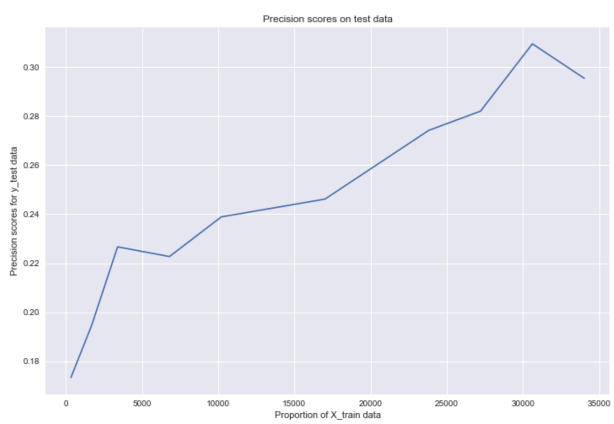
```
Test accuracy: 0.84
[[7082   95]
 [1271   59]]
             precision    recall  f1-score   support

          0     0.8478    0.9868    0.9120      7177
          1     0.3831    0.0444    0.0795      1330

avg / total     0.7752    0.8394    0.7819      8507
```

The baseline model has an accuracy of 84% even though it incorrectly classified 1271 of the defaulted loans as being in good standing. This means the model could essentially misclassify all of the default loans and still be considered highly accurate. Obviously, this is not a good metric for evaluating my model's performance since I am hoping to create a model that accurately predicts whether or not an applicant will default on their loan.

Before going any further, I decided to double check that I had enough data for my model to see whether that explains the results I got in the baseline model. I tested this by running the model on 1%, 5%, 10%, 20%, 30%, 50%, 70%, 80%, 90% and 100% of the data. I used the following code to run my test:

```
from sklearn.metrics import precision_score
from sklearn import tree

ps = [340, 1701, 3402, 6805, 10208, 17014, 23819, 27222, 30625, 34028]
test_scores = []
train_scores = []

for p in ps:
    X_train2 = X_train[:p]
    y_train2 = y_train[:p]
    rf2 = tree.DecisionTreeClassifier(max_depth = 7, random_state=0)
    rf2.fit(X_train2, y_train2)
    y_rf2 = rf2.predict(X_test)
    y_rf3 = rf2.predict(X_train2)
    prec_test = precision_score(y_test, y_rf2)
    prec_train = precision_score(y_train2, y_rf3)
    test_scores.append(prec_test)
    train_scores.append(prec_train)

plt.plot(ps, test_scores)
plt.xlabel('Proportion of X_train data')
plt.ylabel('Precision scores for y_test data')
plt.title('Precision scores on test data')
plt.show()


plt.plot(ps, train_scores)
plt.xlabel('Proportion of X_train data')
plt.ylabel('Precision scores for y_train data')
plt.title('Precision scores for training data')
plt.show()
```

I got the following results which indicated that I do have enough data:



Once I established that I have enough data to run the model, I used GridSearchCV with precision as my scoring metric.

GridSearchCV is a method for selecting the best hyperparameters for a model by running multiple iterations of your model with each iteration testing a different combination of hyperparameters. The GridSearchCV will then identify which combination of parameters creates the strongest score based on your performance metric.

I decided to start with precision as my performance metric. Precision measures the fraction of relevant instances among the retrieved instances. In my case, precision measures the fraction of loans that are actually in good standing based on the total number of loans the model identified as being in good standing.

I initially included balance and n_estimators in my list of parameters to test, but found it to be too computationally expensive with the model taking over 24 hours to run. I decided to set class weight to 'balanced' and n_estimators to 300 for all iterations of the GridSearchCV. The test took just shy of 3 hours to run and resulted in the following:

```
# run RandomForestClassifier using parameters identified in GridSearchCV with class weights balanced

rf = RandomForestClassifier(class_weight='balanced',
            max_depth=25, min_samples_leaf=1, min_samples_split=10,
            n_estimators=300, n_jobs=2, random_state=0)

rf.fit(X_train, y_train)

y_pred = rf.predict(X_test)

errors = abs(y_pred - y_test)

print('Test accuracy: ', 1.0 - round(np.mean(errors), 2), sep='')

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred, sample_weight=None, digits=4))
```

```
Test accuracy: 0.83
[[6937  240]
 [1171  159]]
             precision    recall  f1-score   support

          0     0.8556    0.9666    0.9077      7177
          1     0.3985    0.1195    0.1839      1330

avg / total     0.7841    0.8341    0.7945      8507
```

While this did slightly increase the number of applications accurately classified as defaults, it also increased the number of loans in good standing that were misclassified as defaults.

I decided to perform GridSearchCV once more, but this time the performance metric was True Negatives, or the number of accurate predictions for defaults.

```python
# Create function for GridSearchCV
def fit_model(X, y):

    # Create cross-validation sets from the training data
    cv_sets = ShuffleSplit(X.shape[0], n_iter = 5, test_size = 0.20, random_state = 0)

    #  Create a decision tree regressor object
    regressor = RandomForestClassifier(class_weight='balanced', n_estimators=300)

    #  Create a dictionary for the parameters to test
    params = { 'max_depth': [3, 5, 15, 25],
              'min_samples_split': [10, 20, 30, 50],
              'min_samples_leaf': [1, 10, 20, 30, 50],
              'n_jobs': [1, 2]}

    #  Transform 'performance_metric' into a scoring function using 'make_scorer'
    scoring_fnc = make_scorer(performance_metric)

    #  Create the grid search object
    grid = GridSearchCV(regressor, params, scoring_fnc, cv=cv_sets)

    # Fit the grid search object to the data to compute the optimal model
    grid = grid.fit(X, y)

    # Return the optimal model after fitting the data
    return grid.best_estimator_, grid

def performance_metric(y_true, y_predict):
    cm = confusion_matrix(y_true, y_predict)
    score = cm[1][1]

    return score
# model optimizing for true negatives
# run RandomForestClassifier using parameters identified in GridSearchCV with class weights balanced

rf = RandomForestClassifier(class_weight='balanced', max_depth=5, max_features='auto',
            min_samples_leaf=1, min_samples_split=30,
            n_estimators=300, n_jobs=1,random_state=0)

rf.fit(X_train, y_train)

y_pred = rf.predict(X_test)

errors = abs(y_pred - y_test)

print('Test accuracy: ', 1.0 - round(np.mean(errors), 2), sep='')

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred, sample_weight=None, digits=4))
```

```
Test accuracy: 0.63
[[4487 2690]
 [ 492  838]]
             precision    recall  f1-score   support

          0     0.9012    0.6252    0.7382      7177
          1     0.2375    0.6301    0.3450      1330

avg / total     0.7974    0.6260    0.6768      8507
```

The model accurately predicted over 60% of the defaults, but it is also inaccurately classifying a much higher number of loans in good standing.

k-Nearest Neighbors

The k Nearest Neighbors algorithm groups the data into classes based on similarities. It then uses those classes to make classification predictions. K refers to the number of data points in each class cluster.

First, I ran the model with no changes to get my baseline:

```
# run KNeighborsClassifier using default parameters
knn = KNeighborsClassifier()

knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)

print("kNN Accuracy: ", accuracy_score(y_test, y_pred_knn))

print(confusion_matrix(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn, sample_weight=None, digits=4))
```

```
kNN Accuracy:  0.8268484777242271
[[6976  201]
 [1272   58]]
             precision    recall  f1-score   support

          0     0.8458    0.9720    0.9045      7177
          1     0.2239    0.0436    0.0730      1330

avg / total     0.7486    0.8268    0.7745      8507
```

Similar to the Random Forest Classifier, most of the applications were classified as being in good standing with most of the default loans also inaccurately classified as being in good standing.

I once again performed GridSearchCV and optimized for precision:

```
# run KNeighborsClassifier using results from GridSearchCV (code below)
knn = KNeighborsClassifier(n_neighbors=25, leaf_size=10, weights='uniform')

knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)

print("kNN Accuracy: ", accuracy_score(y_test, y_pred_knn))

print(confusion_matrix(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn, sample_weight=None, digits=4))
```

```
kNN Accuracy:  0.8441283648759845
[[7176    1]
 [1325    5]]
             precision    recall  f1-score   support

          0     0.8441    0.9999    0.9154      7177
          1     0.8333    0.0038    0.0075      1330

avg / total     0.8424    0.8441    0.7735      8507
```

Optimizing for precision resulted in nearly ALL of the applications being classified as good standing. Not good!

I then performed GridSearchCV and optimized for the True Negatives just like I did above:

```
# version where i optimize for true negatives
# run KNeighborsClassifier using results from GridSearchCV (code below)
knn = KNeighborsClassifier(algorithm='auto', leaf_size=10, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=9, p=2,
           weights='distance')

knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)

print("kNN Accuracy: ", accuracy_score(y_test, y_pred_knn))

print(confusion_matrix(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn, sample_weight=None, digits=4))
```

```
kNN Accuracy:  0.8202656635711767
[[6910  267]
 [1262   68]]
             precision    recall  f1-score   support

          0     0.8456    0.9628    0.9004      7177
          1     0.2030    0.0511    0.0817      1330

avg / total     0.7451    0.8203    0.7724      8507
```

This model performed only slightly better than the baseline model and will require additional tuning, so I do not think k Nearest Neighbors is an appropriate model for this project.

Conclusions and Next Steps

My next steps will be to take a deeper look at what is happening with the data to cause such skewed results in the model's classifications. Is it a class weight imbalance? Do I need to perform some additional statistical analysis to normalize some of the data? Am I using the right combination of features?

I will be examining these questions and more as I try to optimize the true negatives and build a model that adequately identifies true negatives while minimizing the number of false negatives.

I do not want to turn away someone who is not at risk of defaulting because that is a lost opportunity for investors and a negative customer experience for borrowers.

I would also like to explore what the threshold is between approving someone who is at risk of defaulting and rejecting someone who would not default. Is there a sweet spot between the number of good applicants who are turned away by an imperfect model and bad applicants that are approved that will maximize profits and minimize loss for investors?

Finally, I would like to do some additional exploration into the potential biases produced by the model? Does it inadvertently discriminate based on race or region? If the model is unfairly biased against marginalized individuals, what steps can be taken to help Lending Club support those communities?

Project Code

A detailed breakdown of all of the code used for this project can be found in the project code.