Capstone 1 Final Report
Lending Club Loan Default Modeling

Thomas Loeber

Overview
This project analyzes data from the peer-to-peer lending platform Lending Club. The goal is to predict whether a borrower will pay back their loan.

Audience and Problem
The hypothetical client of is either Lending Club itself, or any other lending institution that does not have enough of its own data yet with which to do a similar analysis. Predicting loan defaults is of the major problems banks are facing, and achieving high accuracy has an immediate effect on their bottom line. The resulting model is used to decide whether to accept a customer's application for a new loan – and if so, what loan amount to offer and at which interest rate. In addition, the results could also be used to target marketing material at specific audiences that are most profitable.

Note that, while gaining the last bit of additional accuracy gets increasingly hard in any machine learning application, it also yields disproportionate results in a case such as this: If we are able to predict the default risk more accurately than our competitors, we will be able to identify customers which are particularly profitable: Not necessarily those whose default risk is low, because most of those will likely take their business elsewhere unless we offer them a commensurable low interest rate. Rather, the most profitable customers will be those whose default risk is low r*elative to how most of our competitors view them*. In order to attract those customers, we will want to offer them a slightly lower interest rate than they would get elsewhere, but this discount will be small enough to still yield a greater expected return to the lender. Conversely, we will also identify a segment of customers whose risk of default is larger than implied by the models of our competitors. While we do not want to turn those customers away, we will want to offer them a commensurately higher interest rate, which will drive most of them away to competitors who priced their risk too low.

Data Source
A subset of the data is publicly available on Lending Club's website, but in order to access all the data I had to create an account. This data set is very comprehensive, it spans about 10 years (from 2007 – shortly after the firm's founding – to the present) and contains roughly 1 million usable observations for 150 variables.

Data Wrangling
I start by checking for any **duplicate** observations, but I did not find any.
Next, I set the issue date of the loan as the index, because this will allow us to more easily slice and group loans by date. Since oftentimes multiple loans were issued on the same date, I create a hierarchical index that also includes the loan ID.

I go on to **drop** any observations that are irrelevant because our target variable – whether someone defaulted on their loan or not – is yet indeterminate. These consist of observations where the loan is still current, and where the loan is late by up to 120 days.
Having gotten rid of unnecessary rows, we now drop unnecessary columns, starting with variables whose values were not yet available at the time the loan was issued . Doing so is important because using such information from the future would create the appearance of better predictive accuracy than the model actually possesses (endogeneity). At the same time, this step is also tricky, because the data documentation is often lacking; as a result, a few such variables are only discovered later on. Furthermore, I delete a few variables which are not relevant, as revealed by the data dictionary.

Next, I address the topic of **missing values**. This is a problem for roughly a third of our variables, because over time Lending Club started collecting additional groups of variables. I decided to discard all variables that have at least 30% missing values (though any threshold between 15% and 65% would have lead to the same results, because no variables had a proportion of missing values falling within this interval).
I made sure not to drop variables where missingness stands for "not applicable" rather than "not collected." Examples are variables such as time since last delinquency, because not all people in the sample have already experienced this event. I identified those variables by the fact that for those cases, missingness is distributed seemingly randomly across time, rather than confined to certain sample periods (usually the beginning of the sample).
Since all variables for which this problem occurred referred to the time that has passed since a specific event, I chose the following **transformation** for them: First, since for some observations the event occurred zero months ago, I added 1 to each value to make all the counts positive. Then, I raised each variable to the power of -0.5. Finally, I set the value to zero for all observations for which the underlying event did not occurr. (The fact that the event did not occur was indicated either by the value being missing or – for earlier observations – zero.) The result is that observations for which the result occurred more recently received a higher score; and the longer ago the event occurred, the closer the score moves to zero. For borrowers for which the event did not occur at all, the score equals exactly zero, which models that this is equivalent to the event occurring an infinite time ago.

I then go on to check whether variables are of the right **data type**. Originally, all data are imported either as floats or objects. For floats, I identify variables that are in fact integers, and also don't have any missing values (which would require storing them as floats anyway). I then convert them to integers.

Next, I manually inspect all variables of type object. This identifies two variables that should be datetime objects, as well as three variables that should be numeric, but were imported as objects because the unit was appended to the value. I convert all these to the proper type. In addition, I change the datetime variables from absolute date to relative date (time since the loan was issued).

The remaining variables are in fact categorical. After the EDA, I will transform them using one-hot-encoding, after dropping variables with too many unique categories.

Exploratory Data Analysis
The main purpose of the explanatory data analysis (EDA) I carried out was to, firstly, assist in data cleaning by identifying potential problems in the data, and secondly to assist in feature engineering.

Another common use of EDA is to discover interesting relationships between variables (e.g., a non-linear effect that we may try to model by including a variable's square term). Let me justify why I did not make much use of this: Usually, we are interested in the *partial* effect of a variable, i.e. holding other variables constant. If we use visualizations, it is generally only practical to plot the effects of two or three variables at a time. This tends to give a misleading picture of partial effects (e.g., correlated predictors will give rise to spurious correlations). While one might argue that it can't hurt to try finding something interesting that way, we need to take the opportunity cost into account, so I think time is better invested elsewhere. For instance, if we use a flexible model such as gradient boosting, we don't have to worry about trying to visually detect all nonlinear effects and interactions between predictors.

A better use of our time is to use EDA to **find problems with the data**. The reason is this often draws on our domain knowledge, as well as general human knowledge about the world. Both of these are hard to encode into a model, such as the fact that we know that certain variables has to fall within a particular range. Another common problem is that some missing values may have been encoded as zeros or missing values standing for "not applicable" rather than that the true value is unknown.

In fact, I detected both of these problems with Lending Club's data through data visualization. The first problem was revealed while I was deciding which variables have a high enough ratio of valid to missing values to impute the missing values (rather than dropping the whole variable). In making this decision, I not only considered the proportion of valid observations, but also visualized how the missing values are distributed over time. This showed that – unsurprisingly – a high fraction of missingness was usually caused by the fact that the variable was only collected for a subset of the sample period. However, it was suspicious that for some variables the missing values were distributed seemingly randomly across time. (About half the values were missing, so it is unlikely that such a high proportion simply failed to be properly recorded). This prompted me to take a closer look, which revealed that missingness seemed to denote not "unknown", but rather "not applicable": For example, the variable "time since last delinquency" was not applicable to borrowers who never experienced a delinquency. Unfortunately, we don't have two different types of missing values available to encode each. I solved this problem by setting the "time-since" variable to negative infinity for each observation where the event count (e.g., the number of delinquencies) was zero. (Setting it to negative infinity works because I later transformed the "time-since" variables by raising them to the power of -0.5, which gives a higher

score to applicants to whom the event occurred more recently, and converges to zero as the time-since approaches infinity).

In the process of making sense of these inconsistencies, data visualization also uncovered a related problem, namely that "not applicable" was encoded not always as missing, but in some cases instead as zero. It turned out that this practice seems to have changed at some point in the middle of the sample period. Since the data dictionary did not address this, I had to use a number of different plots to figure out what was going on. For example, I looked only at the subset of borrowers who had a zero for the number of months since the last delinquency, and then plotted the monthly average for the number of delinquencies. This revealed that the average monthly number of delinquencies was zero until 2010, but from 2013 on hovered around 2 (there were no observations for the years in between). This suggested that the practice of encoding missing values as zeros stopped somewhere between 2010 and 2013, but this interpretation raises some other inconsistencies, which further plotting showed to be due to a change in Lending Club's lending behavior. (This explained, for example, why there were no zeros at all during the middle of the sample period: By that time, "not applicable" was denoted by "missing", and the lending behavior was still so cautious as to not accept any applicants who had just experienced a delinquency zero months ago.)

A different use of data visualization was to assist in **feature engineering**. The goal was to create features that are less skewed, in particular by taking the logarithm where necessary. I eventually automated this task by quantifying skewness, but in the process it was still helpful to use data visualization in order to come up with good measures and to make sure no errors were introduced. For example, the measure of skewness I settled on looked at how much closer/further the median was from the upper quartile compared to the lower quartile. Thus, this measure did not work for variables that were so skewed that all three quartiles fell onto the same value, so I had to visually decide what was going on with these distributions. Furthermore, plotting some of the extremely skewed variables showed that for many variables, skewness was caused by an inflated number of zeros (located at the minimum) or ones (located at the maximum). For example, for most applicants the number of derogatory public records was zero. This prompted me to take a second step to remediate skewness, namely to create another binary feature for these extremely skewed variables, telling us which values are equal to this inflated minimum or maximum.

Overall, thus, exploratory data analysis through data visualization was able to both detect some problems with the data, and also helped create better features. While I tried to automate these manual tasks as much as possible (e.g., by writing functions that try if taking the logarithm decreases skewness), some of these tasks are (still) too hard or time-consuming to automate.

Summary

The machine learning part uses a number of different classifiers – logistic regression, SVMs with RBF and polynomial kernels, random forests, and gradient boosting. After analyzing the results

of each individual model, I combine all models into an ensemble using a hard voting classifier. Since some of these models became computationally expensive (could not be trained within 1-2 days on my laptop), I pursued several strategies to make estimating these models feasible: Firstly, I used Bayesian optimization to tune the hyperparameters, which takes less iterations than a brute-force grid- or randomized search. Secondly, I ran the more computationally intensive models on AWS EC2 to take advantage of more powerful hardware. Thirdly, I used a single validation set rather than cross-validation where necessary. If this still wasn't enough, I reduced the dimensionality of the features space using principal components analysis, and used only a subset of observations.

The challenge of how to properly split the data

The first challenge was to decide how to split the data into training and test set. While the data do have a time component, I chose to ignore the time dimension for two primary reasons: Firstly, time-series models are substantially more complex, so I leave this to further iterations of the project. Ignoring the time-component should not impact performance too much because we are not dealing with proper time-series (observations on the *same* individual at different points in time), and as a result the correlation of the error terms is likely not as strong as to overwhelm the influence of the predictors (as may be the case for autoregressive processes). While the fact that we are not able to take into account the default rates of recent loans as additional predictors will lower predictive accuracy somewhat, our model will still give us a good prediction of default *averaged* over all future economic scenarios. Note that recent default rates may not be very predictive of the default rates of new loans anyway, because our loans are scheduled to be paid back over a timeframe of 3 or 6 years.

The second reason why I ignore the time component is that it is not even clear if a proper validation strategy could be devised. Since we treat the data as independent, we can simply split them randomly. But if we would model the time-series nature, this yields dependent data, so we would have to decide between two different options for splitting the data: One strategy is to train the models on the earliest portion of the data, and then split the later portion randomly into training and validation set. Alternatively, we could split the data into random chunks (each containing, say, 6 months of data), and then randomly assign some of these chunks to the training set, and finally randomly split the other chunks into test and validation set. (The purpose of still creating test and validation set by random splits is to make sure that test and validation set have the same distribution, which makes model tuning easier.)

Since I do not model the time-component of the data – as explained above – I opt for splitting randomly. It may be tempting to instead train the model on the older data and measure predictive accuracy on the newer data in order to get a better estimate of how our model will perform on future data. The main argument for doing so is that the predictive accuracy computed this way would incorporate the additional uncertainty when making predictions on new loans due to factors such as changing macroeconomic conditions, as well as potential changes over time in the relationships between the predictors and the target.

However, there are other countervailing factors that make splitting based on date potentially less generalizable. The main problem is that since our whole data set only spans approximately one business cycle, our validation and test sets would not contain observations drawn from across the manic cycle (they would come from a boom period). Since loan default rates are highly cyclical, the test accuracy computed this way might thus generalize *less* well to future data if predicting default rates during a boom period is systematically easier or harder.

Similarly, the relative performance of different learners, as well as the optimal hyperparameters, may vary across the business cycle. This is another reason that we want to have our training, test, validation set drawn from across the business cycle in order to choose hyperparameters that train models which generalize best into the future.

For all these reasons, I ignore the time-component of the data, and leave it to potential future versions of this project to explore the feasibility of incorporating the dependence of the data.

Hyperparameter optimization
I initially started out using grid search and randomized search, but these brute-force approaches turned out to be too computationally expensive for the size of my data. Thus, I turned to **Bayesian hyperparameters optimization**, which performs an intelligent search: We start by specifying a prior distribution for the hyperparameters. This formalizes our beliefs about which hyperparameters make sense *before we have seen the data.* Like randomized search but unlike grid search, this gives us the chance to not only specify the minimum and maximum values between which to confine our search, but it also allows us to concentrate probability mass around the most likely values in the center (e.g., through a (log)normal distribution), so that more plausible values are tried more frequently than possible but unlikely values.

By contrast to a randomized search, however, we learn from the results from previous iterations: We start with the prior distribution that we defined above, and update our confidence about where the optimal hyperparameters lie after each trial. The distribution after each update is called the posterior distribution (it is a conditional probability, given the prior as well as the evidence observed so far). If the last values we tried gave us a good result on the validation set, it is likely that we have been moving in the right direction (and vice versa), and we update the probability distribution accordingly. The rules about how to update our confidence about which hyperparameters are optimal are defined by Bayes's Rule. In the beginning, when we have gathered few actual data, our posterior distribution will mainly be determined by the prior distribution we started with. But as we gather more evidence by trying additional hyperparameter combinations, the prior distribution will be swamped by the evidence from these trials and will concentrate more and more probability mass around the values that worked well.

Thus, for the first few iterations, the results will be similar to randomized search that draws hyperparameters from the same distributions that we use for our prior distributions. However, as we try more and more hyperparameters combinations, Bayesian optimization will increasingly

concentrate our efforts on the subset of the search space that worked best previously. This is particularly important for models where we have many hyperparameters to tune (e.g., XGBoost). While randomized search is preferable to grid search, it still runs into the curse of dimensionality, because it becomes increasingly unlikely that a good combination of hyperparameters is selected randomly as the number of hyperparameters increases. Overall, thus, Bayesian optimization allows us to get a better accuracy for a given computational cost, or it allows us to attain a given accuracy at a (often much) lower computational cost.

For the computationally less expensive models (logistic regression, random forests, and XGBoost), the performance of a given set of hyperparameters will be evaluated using 5-fold **cross-validation** to make the most efficient use of the available data. However, since I already have to reduce the sample size considerably for the SVMs with RBF and polynomial kernel, I use a **single validation set** instead. Compared to 5-fold cross-validation, this cuts computational costs by a factor of 5.

The metric I optimize is the **average precision score** on the (cross-)validation set. This measure is similar to the commonly used metric of the area under the ROC curve, but it is better suited to unbalanced classification problems such as ours. The precise meaning of the average precision score is less straightforward, but it is roughly equivalent to the area under the precision-recall curve. Another common metric when using precision and recall is the F1-score (which corresponds to the harmonic mean between precision and recall). Its disadvantage is that it is based on a particular point on the precision-recall curve (namely the point we get with scikit-learn's default probability threshold of 0.5). By contrast, the average precision is based on the entire precision-recall curve, and thus take into account a classifier's performance for different trade-offs between precision and recall.

Predictive modeling

Note that the primary goal of this project is accurate **prediction**. Thus, we do not care as much about identifying which variables are most important, or even about the precise effect of particular variables (which falls under the domain of statistical *inference*).

There are two reasons why prediction and inference can sometimes be in conflict: Firstly, models with the most predictive accuracy usually are more complex, and as a result the effect of one variable often depends on the values of other variables. While we are usually still able to see which features are most important overall, we often cannot get an estimate of how big the effect of these variables is, or even whether it is positive or negative (or both, depending on the value of other features). The second reason why prediction and inference can be in conflict is that in order to maximize predictive accuracy, it is usually beneficial to keep multiple features in the data set even though they may measure the same underlying causational factor. The downside is that this can bias our measures of feature importance for some learning algorithms. Most importantly, for logistic regression – which is unique in giving us estimates of the effect of different variables – these effects are often strongly inflated in absolute value for highly correlated variables. For these reasons, I only briefly describe feature importance for each

model. Note also that while identifying the most important features may seem "interesting", it is not an important business problem: What we want to know is what the default risk of *specific* applicants is, given the *combination* of their unique attributes. Even though a black-box model that does not return a specific reason justifying its prediction may feel less convincing to humans, it is not worth sacrificing predictive accuracy for this.

Let's now talk about some of the specifics. My strategy is to train an instance of each of the major classification models, unless there were any contrary indications. For example, I did not use deep learning because it is complex to train, so it would have been a whole project on its own. The models I used to include logistic regression with elastic net regularization, SVMs with RBF and polynomial kernel, random forests, and XGBoost. The latter achieves by far the best performance (average precision on the test set of 0.41). All the other models achieved a score around 0.39.

The starting point is a linear model, namely **logistic regression** with elastic net regularization. Because the training set has about 800,000 observations and 130 features, I used stochastic gradient descent to substantially cut down the computational cost.

The next model is a **random forests** classifier. Its strength is that it is able to model nonlinear relationships and interactions between predictors. For both models, I get a similar average precision score of 0.39.

I continue with another tree-based ensemble, **XGBoost.** a particularly good implementation of gradient boosting. It should not be too surprising that it achieves by far the best performance of all models (0.41).

The remaining two models **– SVMs** with RBF and polynomial kernel, were by far the most were computationally expensive, so I trained them in a separate notebook run on AWS EC2. As mentioned in the beginning, I still had to take additional steps to make the computation feasible. In particular, I reduced the dimensionality of the features space from 130 to 30 by extracting principal components. Since the first 30 principal components explain about 90% of the variance in the features, this seemed like a promising strategy. I still had to reduce the training size to 80,000 observations. Unfortunately, the model performs slightly worse as logistic regression and random forests. I thus tried using the original features instead, which forced me to further cut the training size to 40,000. This yielded a similar but slightly better performance than all other models except XGBoost. Finally, I trained a SVM with polynomial kernel, which performed similar to logistic regression and random forests. Here, I only tried the original features, since these had performed better than the principal components for the RBF kernel.

The final step is to combine all these different models into an **ensemble** in order to take advantage of the insights from the different learners. Ensembles have been shown to be a good way of decreasing both bias and variance of predictions without overfitting. There are different

ways of creating ensembles. The one I choose is a **hard voting classifier**. This strategy simply takes a majority vote of the hard predictions (class labels) from the individual models.
The disadvantage of this strategy is that we lose some information by converting probabilistic predictions to binary class labels before the vote. It would thus be better if we could simply average the probabilities (soft voting classifier). However, the problem is that SVMs return the distance from the separating hyperplane rather than probabilities. While it is possible to estimate probabilities from this, this is computationally too expensive .

One important decision we need to make is how to convert the soft predictions into hard predictions. By default, scikit-learn will generate hard predictions by using a threshold of 0.5 for probabilities and 0 for distance to the seperating hyperplane. In other words, if a borrower's default risk is greater than 50%, we'll predict that they will default. However, it may make sense to deviate from this threshold if the cost of false positives and false negatives is asymmetrical.

In our case, we probably want to err on the side of caution, because the cost of one default is much higher than the income lost from making one fewer loan. How to precisely weigh these two different costs is a much more complicated issue than we can solve here, in part because we do not of knowledge about all the costs associated with each. This decision would also be easier if we were able to model the net present value of each loan rather than a simple binary prediction of default or no default, because the cost of default depends in large part on how much a borrower pays back before the default. Unfortunately though, as explained in more detail in the proposal, we did not have the necessary data to compute this more appropriate target variable. Thus, a justifiable choice is a recall of 0.5, which allows us to attain a precision of about 0.35. This means that we are willing to turn away (a little under) two applicants who would not default in order to successfully identify one applicant who will default. For models returning probabilities, this corresponded to turning away applicants whose default risk exceeded between 0.5 and 0.65, depending on the model.
After computing hard predictions on the test set for all models, I go ahead and compute the average. I use a simple average, except that I weight XGBoost twice as much as the other models. This is justified because it not only performs much better, but it's predictions are also slightly less correlated to the other models. In addition, this avoids the possibility of ties (as we effectively go from averaging 6 to 7 models).

While one could develop an even more sophisticated weighting, deciding on how to compute the exact weights is challenging because these need to take into account not only a model's relative performance, but also the correlation of its predictions with other models. Thus, I will go with the simpler solution describe above for now.

Scikit-learn implements a voting classifier, but it wasn't possible to use it here. The reason is that I relied on XGBoost's native API rather than the Scikit-learn API (since this allowed us to use a more optimized data format, amongst other). Since scikit-learn's voting classifier takes instances of the individual models as its arguments, I would not be possible to include XGBoost in the ensemble.

However, implementing a hard voting classifier is straightforward. I start by taking the (weighted) average of the predictions for each test set observation. As explained above, we will double the weight of XGBoost.

We need to decide again how to compare the performance of this ensemble with the performance of the best model, XGBoost. Note that it does make sense to compute the average precision of a hard voting classifier (unless it averaged over a lot more models and thus gives a more continuous output), since doing so requires soft predictions. Instead, I look at precision and recall, as well as the F1-score. The latter is now a good summary of our model's performance: Since we already calibrated our class predictions to achieve a trade-off between precision and recall suitable to our business problem, we no longer need to take the classifier's performance on the whole precision-recall curve into account, but only care about its performance at the trade-off we chose.

Results

I find that the ensemble performs very similar to XGBoost. Both achieve the same F1-scores for identifying defaults and non-defaults. When we take a more detailed look, we see that XGBoost performs slightly better on some measures (recall on non-default). Therefore, we would probably go with its predictions for making loan decisions for now.

Nevertheless, the results do show the power of ensembles: Even though the other models - which performed considerably worse - constituted 5/7 = 71% of the ensemble, the resulting accuracy is almost identical to XGBoost. By adding more different models, and by weighting the predictions in a more sophisticated way, we should be able to create an ensemble that surpasses the performance of the best individual model.