Predicting loan defaults is of the major problems banks are facing, and achieving high accuracy has an immediate effect on their bottom line. The predictions are 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.

While gaining the last bit of additional accuracy gets increasingly hard in any machine learning application, it also yields disproportionate results for an application area 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 relative 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 differently.

The primary goal of this project was thus accurate *prediction*. As a result, 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).*

The first step to maximizing our predictive accuracy was to make sure that our data are clean, because this gets rid of noise that would otherwise increase the estimator's variance. This process also helped with getting to know the data better, which in turn allowed leveraging our domain knowledge to engineer better features. In particular, I normalized all monetary variables by dividing them by each applicant's income. While we don't know how much of a boost in accuracy each of these steps added, we do know that the new features I computed disproportionately ended up among the most important features.

However, an important lesson was that doing a good job cleaning the data takes a lot of time, so we have to ask beforehand whether we are willing to pay that extra cost in order to get a small boost in accuracy. The best solution may be to start out doing only a minimum amount of cleaning, and then jump into predictive modeling to get results as quickly as possible. We can then decide whether we are willing to invest extra time in order to get incremental improvements.

The same is true for feature engineering. It is worth pointing out that more sophisticated models such as XGBoost and random forests make less demands on the distribution of the data. For example, they are not affected by skewed distributions that are far from Gaussian. Since XGBoost ended up being the best performing model anyway, a quick first approach could limit feature engineering to a bare minimum to only address problems that these models are not able to deal with on their own (e.g., dividing the borrower's loan amount by their annual income, which drew on human domain knowledge).

During the predictive modeling stage, it again took a lot of extra effort in order to get the last increase in accuracy. Firstly, I needed to resort to Bayesian hyperparameter optimization in order to be able to optimize more than two or three hyperparameters. This more advanced technique was thus not strictly necessary for logistic regression and SVMs, but it proved vital for XGBoost, which had a lot of hyperparameters to optimize. The best alternative probably would have been to perform a lazy search, i.e. to sequentially tune one or two hyperparameters at a time. Needless to say, this would have caused us to take a performance hit on our best-performing model.
A second step I took during the predictive modeling stage was to run some of the models on AWS, most importantly in order to train an SVM with RBF kernel. While this algorithm turned out to yield the second best performing model, it still did not come close to XGBoost.
Finally, I combined all models into an ensemble using a hard voting classifier (weighting the predictions of XGBoost by a factor of two due to its superior performance). While this ensemble performed better than the (weighted) average of their individual scores, its performance came very close but did not reach the performance of XGBoost. Thus, we would want to go with XGBoost rather than the ensemble for actual predictions.

Overall, thus, a lot of extra effort was necessary in order to squeeze out the last bit of accuracy. On the other hand, as mentioned above, this is likely worth it for a problem such as this. In fact, a lending institution that is only slightly better at predicting its applicants’ default risk may derive such disproportionate results from this that it may be worth to keep fine-tuning the model even more.