Ben Lengerich
Matt Rozak
Bryan Dickens

Prediction Addiction

In this project, we attempted to use 769 unlabeled features to predict the amount a bank loan would default. To accomplish this, we broke the problem into two sub-problems: first classifying loans as either defaulting or non-defaulting loans, and then subsequently fitting a regression to the defaulting loans to predict their amounts of default. This project was complicated in several ways, including synthetic features, duplicated features, and an extremely high proportion of loans that did not default. While we were able to overcome many of these challenges, we did not anticipate the amount of feature engineering that would be required, and so future improvements and increased computational resources may improve our models.

1. Feature Engineering

Within the loan default prediction Kaggle competition, there were 769 unique features per each account and thousands of accounts. From the beginning we knew we needed to find an ideal subset of a few key features to best build models around and get reliable results, as having 700+ features will almost guarantee us over fitting to the data.

Our first thing to address was the NA values within the dataset, as there are thousands of them scattered about, and cannot be simply ignored. The first attempt was to just replace all cells with 0, but after the second milestone we readdressed this and tried three different attempts of replacement based off of the mean, median, or mode of the specific feature the cell was NA in. Out of the three attempts, the best replacement strategy was the mean, and we fit the data with an Imputer and transformed the dataset.

Next for feature engineering which was consistent from the beginning, was change all the labels to a Boolean 0/1 based on whether the loan defaulted or not. All the best performers in this competition did a similar strategy, and it makes the most sense to simplify the problem down into a classifier problem at the feature selection stage. With data scaling, we initially scaled the data immediately, but as will be discussed later in the feature construction section, we decided to hold off on the data scaling until after the features have been narrowed.

Covered in previous milestones was the earlier feature engineering attempts that we will cover briefly again here. First was the decision tree which uses Shannon Entropy to find the best feature to split on and then continue down the tree. The results were a relative failure due to excessively long runtime and huge changes in trees on small variations in features. However, learning what not to do helped narrow us to what we should do. Next was SelectKBest method, which was a sklearn package with a classification scoring function that selects the k best scores. This method was fast and resulted in moderate success. It was best coupled with

ensemble bagging and cross validation folds, in which we sampled 1000 bootstrap replicates each selecting the best 20 features out of 1000 randomly selected data points in 10 different fold. Then having each replicate vote on which 20 were the best features. The aggregate results from this experiment are included below:

```
Randomly selecting 1000 chunks of 1000 data points and
return most popular.

Overall best 20 features are [402, 757, 403, 320, 758, 279,
756, 321, 667, 404, 759, 312, 620, 280, 374, 322, 621, 375,
189, 261]

And then random selecting again at k = 50

Randomly selecting 1000 chunks of 1000 data points and
return most popular.

So overall best 50 features are [757, 402, 320, 279, 403,
758, 756, 620, 667, 759, 404, 321, 621, 312, 280, 375, 400,
374, 666, 23, 398, 322, 281, 511, 24, 422, 441, 189, 313,
29, 510, 261, 440, 251, 11, 668, 221, 314, 179, 191, 226,
66, 421, 395, 140, 256, 190, 53, 296, 662]
```

However, passing these narrowed best features did not improve our classifiers or regression predictions much, so it was back to the drawing board for the next best feature narrowing technique. Inspired by the forum chat called "Golden Features", there was a participant who found high correlations between two features when he ran feature construction using the difference operation to combine them. An example is f100-f200, then treating that as a new feature and measuring the correlation between that and the results column. We decided to extrapolate that to all basic operations (+-*/) and investigate it much further.

We began by initially measuring feature correlations directly without any feature construction. For runtime simplicity we initially limited to the best 20 feature correlations for each combination below. The ranking we chose was a linear correlation coefficient with the formula:

$$r_{12} = \frac{[\sum_i (y_{i1} - \widehat{y_1}) * (y_{i2} - \widehat{y_2})]}{\sqrt{[\sum_i (y_{i1} - \widehat{y_1})^2 * \sum_i (y_{i2} - \widehat{y_2})^2]}}$$

We applied that with each feature and then sorted based on the highest correlation. The pseudocode below is used for these correlations:

```
1.for i in range(features_count):
2.   correl = data[:,i],results_column #this is r₁₂
3.   feature = (i, abs(correl[0]))
4.   feature_correl_list.append(feature)
5.sort(feature_correl_list) on max correl
6.return sorted feature correl list
```

The results of just the initial feature correlations were:

Base Initial feature correlations

feature	**f322**	**f323**	**f314**	**f376**	**f377**	**f25**	**f324**	**f31**	**f26**	**f258**
correlation	0.123837712	0.111112751	0.110451	0.103534119	0.103519596	0.101329536	0.100613615	0.099761545	0.099245816	0.088416
	f315	**f191**	**f263**	**f253**	**f223**	**f228**	**f514**	**f193**	**f316**	**f181**
	0.099010707	0.097377	0.093318	0.093118	0.090319	0.090246	0.089583	0.088987	0.088962	0.088946

Then we measured each of the basic operations, and all that was changed in the pseudocode was another for loop of j in the same range through the features, and the correlation now was:

```
2. correl = data[:,i] (+-*/) data[:,j],results_column
```

The results here were as follows on the next page:

It can be observed that the difference feature correlations are by far the most useful, with the top performer of scoring 0.20 on the correlation f528-f274. Followed by a close second of f528-f527 with a correlation of 0.19. All other combinations show a decent correlation usually ranging above 0.1 but nothing as significant as these two.

Adding Features Correlations

feature combo	**f263+f322**	**f253+f322**	**f124+f322**	**f228+f322**	**f258+f322**	**f114+f322**	**f223+f322**	**f89+f322**	**f251+f322**	**f268+f322**
correlation	0.1355777	0.1341873	0.1334459	0.133265	0.1328677	0.1324499	0.1309782	0.1308356	0.1308296	0.1308205

Subtracting Features Correlations

feature combo	**f528-f274**	**f528-f527**	**f263-f766**	**f263-f404**	**f253-f766**	**f253-f404**	**f228-f766**	**f228-f404**	**f124-f766**	**f258-f766**
correlation	**0.2042486**	**0.189592**	0.1468195	0.1467686	0.1462188	0.1461654	0.1453182	0.1452655	0.1447405	0.1447262
	f124-f404	**f258-f404**	**f114-f766**	**f114-f404**	**f223-f766**	**f223-f404**	**f268-f766**	**f268-f404**	**f89-f766**	**f89-f404**
	0.1446976	0.1446714	0.144497	0.1444515	0.1435274	0.1434745	0.1425235	0.1424714	0.1424431	0.1423988

Multiplication feature combo	**f766*f768**	**f766*f766**	**f471*f322**	**f322*f4**	**f768*f767**	**f322*f228**	**f322*f258**	**f322*f124**	**f322*f89**	**f322*f263**
	0.1322689	0.132232	0.1314957	0.1299152	0.1299033	0.1294497	0.1288635	0.128651	0.1284793	0.1284033

Division feature combo	f377/f13	f376/f13	f322/f230	f322/f260	f322/f121	f322/f213	f322/240	f314/f13	f322/f190	f322/f211
correlation	0.1326399	0.132566	0.12849	0.1284836	0.128124	0.127136	0.127136	0.127063	0.126657	0.126113
	f322/f250	f322/f170	f322/f111	f322/160	f263/f13	f322/f131	f119/f404	f268/f404	f193/f13	f119/f291
	0.1255513	0.125045	0.12497	0.124494	0.12402	0.122736	0.120238	0.120145	0.119947	0.117427

If allotted more time we would explore combinations of three or more features, i.e. (f100+f200)/f300. We also would explore more unique operations such as exponential and logarithmic functions. These are all computationally and time expensive as just the simple 2-combo basic operations runtime is 4*769*769 different correlation computations.

After the competition had been completed, we reflected on the feature engineering and one disappointing aspect of this competition was that many of the features were revealed to be synthetic. Some of the most powerful features were discovered by visual inspection, not computational methods. For instance, the winner of the Kaggle competition revealed that features f67 and f597 was a sort of group identifier, and within each group there was only one default. We didn't invest much time in visually inspecting the data like this, and we didn't approach the problem of feature group identifying as we really had never heard of such a thing before. However I personally had never done feature construction either, and thought that it is a neat approach to *add* features in solving the problem of feature *narrowing.* The experience of this Kaggle project has without a doubt been a fantastic learning experience and I hope to further pursue future competitions.

2. Regularization Strategies

In order to ensure that our functions extended well to unknown data, we tried both L1 and L2 regularization. Because most of the features were junk data, the sparser solutions produced by L1 regularization were able to more easily match the targets.

3. Variance of the models

In order to decrease the variance of the regression, we used ensemble methods to combine several weak models into a stronger model. These included gradient boosting, extra trees, and random forest regression models, the performance of which can be seen in the table below. We chose to pursue gradient boosting regression, as it seemed the most promising.

Regression Type	R^2
Gradient Boosting	.388
Extra Trees	.309
Random Forest	.303
Stochastic Gradient Descent	.033

To measure the variance of the regression model, we measured its performance on a test data after training on various amounts of data. Notably, the performance peaks at ~9000 training records; more than that leads to overfitting. For our models, we used 6554 records to train.

Performance of Regressor as a function of training size		
Proportion of Data for Training	**Number of Records in Training Set**	**Performance (R^2)**
.1	978	.200
.33	3326	.338
.5	4891	.353
.66	6554	.409
.85	8315	.374
.9	8804	.439

.95	9293	.395
.99	9685	.363

4. Other Improvements

Since we broke the project into a classifier and a regression, this section will first explore improvements made to the classifier and then follow with improvements to the regression.

Classifier

Our approach to classification evolved over the course of the project. Each attempted approach is documented below. Failures are documented due to their success of teaching us how to not go about things.

Broad Classifier Discovery

Initially, we weren't sure of the proper approaches to classifying the loan data. In order to get a rough idea of the right algorithm to move forward with, we tested scikit-learn's SGDClassifier, LinearSVC, and Gaussian Naive Bayes methods. Unfortunately, this testing was done in a poor manner. We trained each approach with the entire training data set and tested with the entire training data set. We did no feature selection and instead used every feature. Null values in the data were replaced with zeros or ignored. As could be expected, this approach led to unsatisfactory results. None of the methods outperformed the all zeroes baseline, and most of them performed much worse.

SGD

Out of all of the methods tested in the initial stage, scikit-learn's Stochastic Gradient Descent methods achieved the best results. Thus, we decided to attempt to pursue the SGD approach. Unfortunately, we continued using no cross validation or feature selection. Due to a lack of knowledge at the time, we attempted to vary the SGD parameters with the hope that a decent classifier would somehow be generated. Of course, this approach led to more underperforming results.

Logistic Regression Classifier

After attending a few more weeks of class, reading a few helpful discussions on the Kaggle forums, and keeping up with several machine learning blogs, our first successful classifier was born. In this stage of the project, we realized how important feature selection, data preparation, and cross validation were. We also learned about scoring metrics (AUC, F1) that measure the accuracy of a classifier with an unbalanced dataset more accurately than raw correct percentage.

After narrowing the data to 2-3 "golden" features, imputing missing values with the mean, scaling the data, and using k folds cross validation with scikit's Logistic Regression method, we achieved an AUC of ~.91. Now that we had a satisfactory approach, we needed to tune it. In addition to a handful of "golden" features praised on the Kaggle forums, we also generated the top 50 "best" features. We ran the classifier trained with all ~4000 combinations of the golden features and the top ten "best" features and dumped the scoring metrics including AUC, F1, percentage correct, true positives,

false positives, true negatives, and false negatives to a CSV file. After analyzing the results, we found several combinations of features that achieved an AUC of ~.95. With this knowledge of the best performing feature subsets, we were then able to tune the parameters of the Logistic Regression Classifier method. We varied the parameters of the Logistic Regression Classifier and generated scoring metrics for about 1600 of these variations. After dumping the results to CSV, we found a variation that achieved an AUC of .9717!

We were never able to tweak the Logistic Regression Classifier to outperform this result. We tried generating our own features by adding, subtracting, multiplying, and dividing features and feeding these to the classifier as extra information, but this approach was doomed due to the exorbitant amount of possible features and the lack of computer time and RAM needed to test them. One Kaggle forum member who successfully attempted this approach mentioned that 96GB of RAM was needed to run his solution. We came to the conclusion that the only way to achieve better results was to attempt a different approach.

Decision Trees

We spent a minimal amount of time attempting to get a decision tree model to work for us. Feeding the decision tree methods with our best feature subsets and varying the parameters never achieved an AUC of greater than ~.6. We abandoned this approach rather quickly, so it may have been possible to coax better results out of it given more knowledge about this method.

Gradient Boosting Classifier

It seemed that virtually every top Kaggler used a Gradient Boosting Classifier in their solution. Thus, we decided to try this approach as well. We quickly determined that not having a fast computer dedicated to machine learning was a huge disadvantage with this method. Running the Gradient Boosting Classifier with a limited number of features and the default (low-intensity) parameters resulted in a classifier that ran in a reasonable amount of time but gave unsatisfactory predictions. The top Kaggle submissions decreased the learning rate, increased the number of features to several hundred, increased the number of estimators to thousands, increased the max depth, and changed other parameters to much more intensive values. We never got the chance to experience the results of the Gradient Boosting Classifier with similar parameters because it could never complete by the time that the laptop running it needed to be closed and moved. We also tested this approach with more moderate parameters and a smaller subset of features, but this only managed to achieve an AUC of ~.7. With more time and computing power, we would have loved to see where this approach would have taken us.

Classifier Conclusion

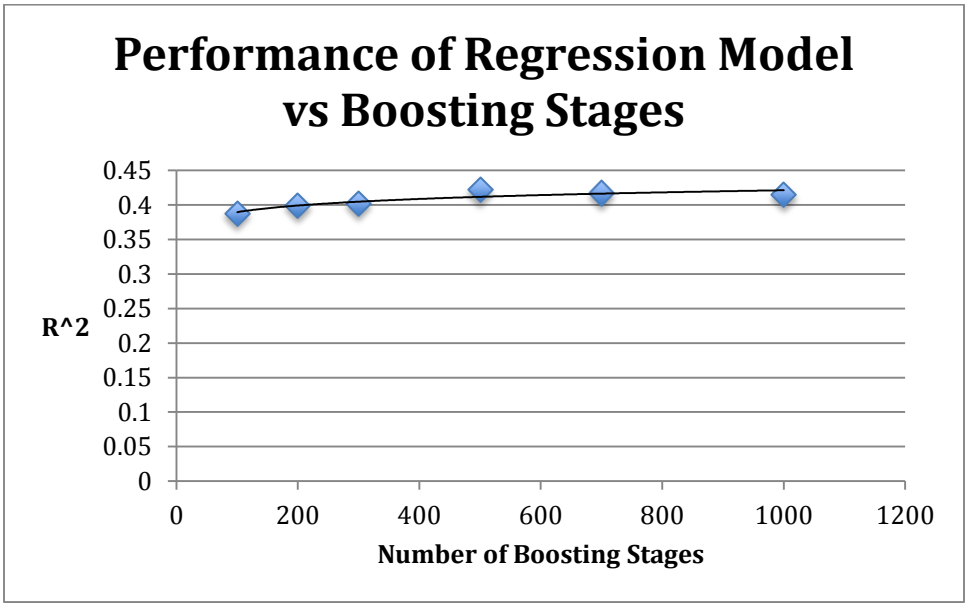
In conclusion, we see our classifier results as decently favorable. In future competitions, we would definitely place more emphasis on feature generation and ensembles much earlier in the process - it seems that many of the successful Kaggle members had the luxury of crunching numbers for many hours or days at a time.

Regression

To improve the regression model, many factors of the Gradient Boosting Regression model were tested. We settled on a gradient boosting model as it greatly outperformed all other ensemble regression models (previous failures have been documented in prior reports), though care had to be taken because it could easily become computationally prohibitive. Shown below are a number of tests used to tune the regression model.

Boosting Stages

The effect of the number of boosting stages was explored to see if a model that was faster to train could approximate the accuracy of a computationally prohibitive model. From this data, we realized we could get a good approximation of optimal model behavior from just 100 stages, and above 700 overfitting is a serious concern.



Inference Methods

We also experimented with various inference methods to replace missing values. These included replacing the unknown values with either 0 or the mean, median, or mode of the feature column. Surprisingly, replacing values with 0 proved to be the best strategy, perhaps because many of the synthetic features were irrelevant. Additionally, we tried randomly sampling other record's features for many simulations to get an average value, but this quickly became computationally prohibitive. We also attempted to model the joint distribution of the features, but we quickly abandoned this idea as it was even more intractable.

Method	**R^2**
0	.401
Column mean	.388
Column median	.392
Column mode	.400

5. Conclusion

After much tinkering with the individual models, we tested variations of combinations of the classifiers and regressions. The best performances are shown in the table below.

Model	**MAE**
Classifier+Regressor	**0.81061554**

Overall, we experienced decent success with this project. With more computational resources, we would improve our models, using decreasing learning rates to ensure global optimizations rather than local and exploring more memory intensive solutions. It was very frustrating that many (if not most) of the given features were worse than useless, but we learned the immense value of feature engineering in circumstances like these. Furthermore, we learned a great deal about various modeling techniques and error measurements, and with a bit of perseverance we were able to create a model that beat the all-zeroes benchmark.

Number of Boosting Stages	**R^2**
100	.388
200	.399
300	.402
500	.422
700	.417
1000	.410