Jeromie Clark

Capstone Report

Project Overview

Kaggle, a data science competition site, is currently hosting a competition on Mortgage Default Risk[1] targeting profiles of individuals with little to no credit history and individuals whom fell victim to "untrustworthy lenders" and as a result, have poor credit histories with traditional credit bureaus.

Instead of relying exclusively on traditional markers like credit score and payment history, the company uses "alternative data, like telco and transactional information" to determine their customers' creditworthiness.

The contest sponsor has provided a comprehensive, anonymized dataset that includes any previous credit bureau history and open loan balances, details on any outstanding commercial credit and cash loans, credit card balances, previous applications for loans from the same individual, and a history of all payments made on the loans issued. The application dataset is comprised of several hundred features, and hundreds of thousands of records, comprised of a mix of continuous, binary and categorical data.

It's also worth noting that the contest provides both a labeled training set and unlabeled test set. We'll use the training set exclusively for model selection and evaluation using k-fold cross evaluations for benchmarking, and submit predictions to Kaggle based on the provided test set in order to retrieve actual contest scores.

Problem Statement

Using the dataset provided, which includes both metadata about each borrower as well as the payment history of each loan, the goal is to successfully create a model that provides an accurate prediction of whether or not a borrower will repay a given loan. Our task is to use the labeled dataset to build a model with which can accurately predict whether or not future loan application will be repaid as agreed.

Candidate Models

Using the scikit-learn flowchart[2] for choosing an algorithm as a guide, I ultimately selected three algorithms to compare based on the characteristics of our data, Stochastic Gradient Descent (SGD), Random Forest and AdaBoost. The latter use ensembles of Decision Tree Classifiers as weak learners.

[1] https://www.kaggle.com/kailex/tidy-xgb-0-777/data
[2] http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

Our dataset is large and feature-rich, with hundreds of features and hundreds of thousands of records. The size of our dataset rules out Support Vector Machine and Naïve Bayes algorithms, both of which are too computationally expensive to be practical with datasets of this size.

Stochastic Gradient Descent is an optimized "on-line" variant of a gradient descent algorithm, which evaluates the loss and updates the learning model as the algorithm progresses using a tunable learning rate. SGD balances speed and accuracy when fitting the dataset, which makes it feasible for use on a large dataset. The SGDClassifier implementation that we're using uses SGD for fitting an underlying Support Vector Machine[3]. Stochastic Gradient Descent was first proposed by Robbins and Monro in 1951[4], and although the algorithm has long been established, it is still widely used in modern machine learning.

Random Forests and AdaBoost are ensemble methods, which use multiple instances of estimators (known as weak learners) to produce meta-results that are more accurate than individual instances of the underlying estimators themselves. The Random Forests algorithm uses a number of Decision Tree classifiers on subsets of the training data to improve the predictive accuracy of the algorithm and control the tendency of a single Decision Tree to overfit the training data[5]. The first random decision forest algorithm was introduced by Tin Kam Ho in 1995, and later refined and trademarked by Breiman and Cutler in the early 2000s[6].

AdaBoost, short for Adaptive Boosting, was introduced in 2003 by Fruend and Schapire[7]. It's similar to Random Forests in that it uses the aggregate output of an ensemble of weak learners (again Decision Tress in our application), but AdaBoost increases the weight of incorrect answers on each iteration, which causes the algorithm to weigh misclassified data points, focusing an increasing amount effort on the cases that are more difficult to classify.

Metrics

Our contest sponsor cares primarily about avoiding borrowers that are likely to have performance problems in the future. To that end, we are using accuracy as the primary performance metric for this project. Accuracy is defined as the ratio between the number of correct predictions to the total number of predictions.

As a secondary concern, erroneously identifying borrowers as potentially problematic is detrimental to our lender's business, as it unnecessarily reduces their pool of potential customers. The fewer loans our lender makes, the less opportunity they have to profit from issuing a loan.

Initially, I used Accuracy and F-Beta Scores as metrics; however, because this is an imbalanced problem (only 8% of borrowers actually default), accuracy in and of itself is fairly useless. F-Beta scores give us a good sense of performance against both accuracy and recall; however, the

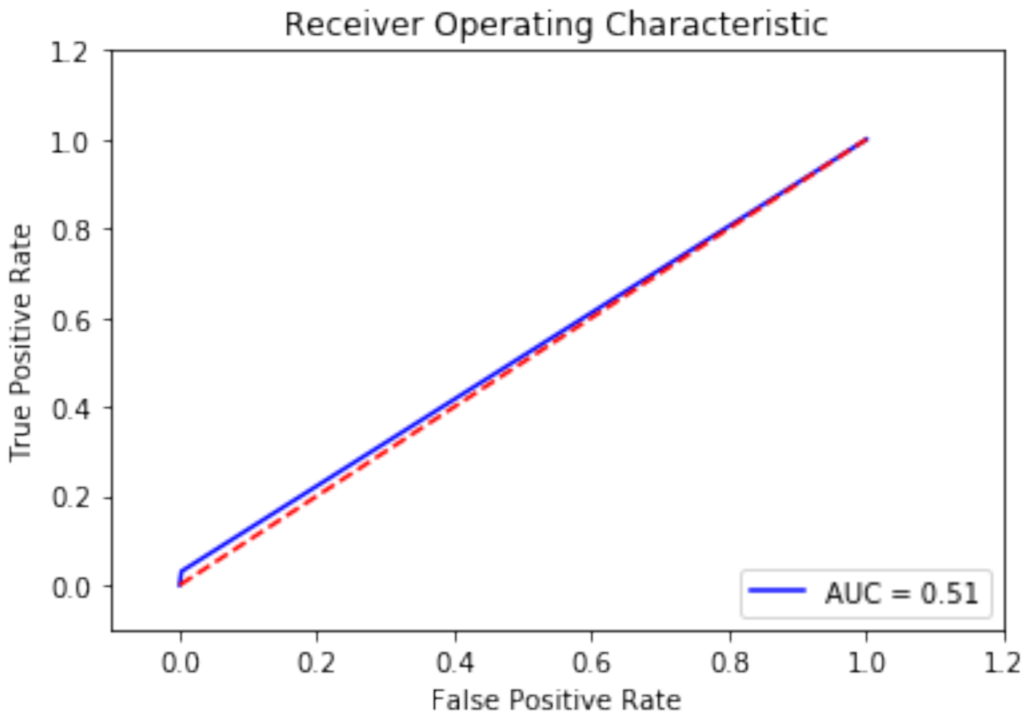
---

[3] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
[4] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[5] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
[6] https://en.wikipedia.org/wiki/Random_forest#cite_note-breiman2001-7
[7] https://en.wikipedia.org/wiki/AdaBoost

Kaggle competition uses another metric, Receiver Operating Characteristic, Area under Curve (ROC AUC) for scoring. For consistency, I ultimately adopted ROC AUC as the performance metric for this project.

The Receiver Operating Characteric Curve is a plot of a given model's True Positive rate or recall (the ratio of positive classifications that were correct identified vs. the actual correct total) on the Y-axis, vs the False Positive Rate (the ratio of positive classifications that were incorrect vs. the actual correct total) on the X-axis at various thresholds.

The ROC AUC score by extension is expressed as the percentage of the total area under the ROC curve.



In this instance, the red dotted line represents the performance of our naïve predictor, a coin-toss performance of 0.50. Our Receiver Operating Characteristic Curve is the blue line, which is a very small improvement on the naïve predictor. The area under the curve (AUC) is the area under the blue line.

Benchmarks

Because only approximately 8% of borrowers have payment problems, the dataset is skewed in terms of the distribution of problematic and successful borrowers. Approximately 92% of our loans as not problematic, which means that it's trivial to achieve a high (92%) accuracy rate, while in actuality, failing to detect any problematic loans. This naïve predictor was built as a benchmark for comparing the performance of various machine learning algorithms.

Because accuracy alone can be a misleading metric, we must also consider algorithmic performance in terms of precision and recall. These metrics are typically combined to create what is known as the F1 score, defined as the harmonic mean of precision and recall scores. More specifically, we use the F-beta score with a beta value of 0.5 to emphasize precision. The result ranges from 0 to 1, with one being the best possible score.

Our naïve predictor achieves an accuracy score of 0.9190 (~92%), and an F-Beta score of 0.9341. When looking at the ROC AUC score, the evaluation returns 0.5, which equates to the performance of random chance.

Our goal for success is to exceed the naïve predictor benchmark with an ROC AUC score > 0.5. Representing an improved ability to correctly identify borrowers that might default on their loans.

Data Exploration

The dataset provided for this contest is broken into multiple files. The primary source of information is application_train.csv, which contains both the bulk of the information about the applicant and metadata about the loan application itself.

This project begins with a significant amount feature exploration and data preprocessing, model comparison and tuning, and iteration over various hyperparameters and preprocessing steps. I ultimately augment the dataset with additional features from the bureau and bureau balance datasets, examining performance along the way to understand the impact of those new features. A detailed exploration can be found in the supplemental Jupyter notebooks.

Much of the data outside the application_train.csv file requires some feature engineering, as there are one-to-many relationships between the applicant and data points like individual credit card accounts, open balances, previous defaults, etc.

The application_train dataset itself is comprised of 122 individual features. Each application has a unique ID, which we can use as a key to join in data from the other data files. For basic analysis and sanity, I bucketed each feature into one of four categories and wrote a utility function to pull basic statistics about the data sufficient to validate basic sanity and identify any necessary preprocessing steps.

The initial dataset had 61 non-numeric (categorical) features, defined as features comprised of two or more possible strings, which reflect a particular categorization. As an example, the CODE_GENDER feature (Figure 1) was comprised of ‘F’, M’, and XNA’ values. These representations were analyzed for sanity, and then tracked for One-Hot Encoding during the preprocessing phase.

```
Value: Civil marriage Count: 29775 Percentage: 9.682580
Value: Married Count: 196432 Percentage: 63.87804013514
Value: Separated Count: 19770 Percentage: 6.42903831082
Value: Single / not married Count: 45444 Percentage: 14
Value: Unknown Count: 2 Percentage: 0.00065038323832318
Value: Widow Count: 16088 Percentage: 5.231682769071676
```

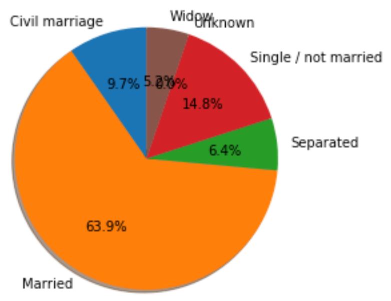


Figure 1 - NAME_FAMILY_STATUS Feature

```
Value: F Count: 202448 Percentage: 65.83439291602576
Value: M Count: 105059 Percentage: 34.164306317497584
Value: XNA Count: 4 Percentage: 0.0013007664766463637
```

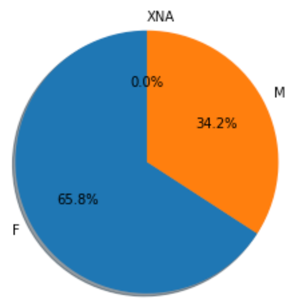


Figure 2 - COUNT_GENDER Feature

In addition, there was a subset of 6 categorical features which were actually binary in nature (Figure 3), typically characterized by “Yes/No” or “Y/N” fields. We classified those features as “String to Bool”, and ultimately converted those fields to binary (1|0) representations. The meaning is identical, but the binary representation is more effectively processed by standard machine learning algorithms.

```
Value: N Count: 202924 Percentage: 65.98918412 Value: 0 Count: 202924 Percentage: 65.98918412
Value: Y Count: 104587 Percentage: 34.01081587 Value: 1 Count: 104587 Percentage: 34.01081587
```

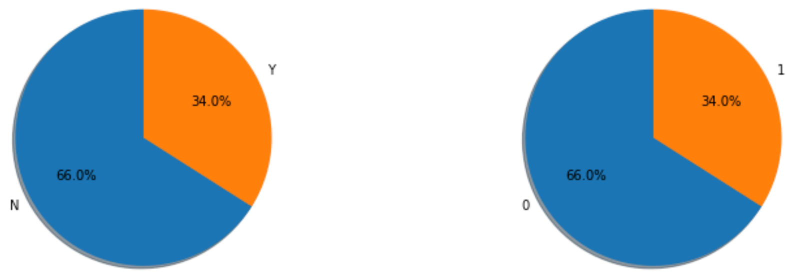


Figure 3 - Conversion of Y/N to Boolean Values

Machine learning algorithms tend to perform best under conditions when continuous fields represent a Gaussian distribution.

This graph shows the normal distribution of applicant’s ages, represented as a negative integer, relative to the application date. The data is distributed along a reasonable curve and is well-centered.

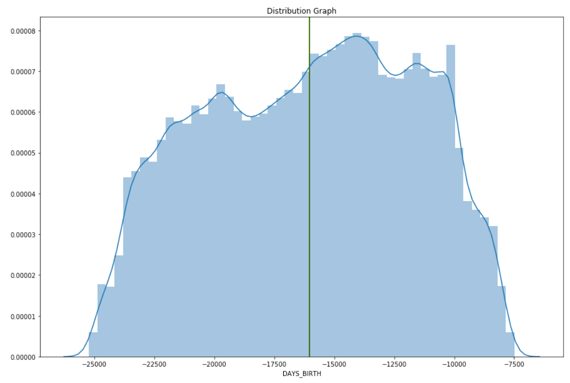


Figure 4 - Centered, reasonably normal distribution

In some features; however, we see heavily skewed datasets. This is relatively common, particularly when working with financial data, where you tend to see high counts near zero, and low counts of large values at the opposite end of the spectrum.

In the example below, we examine the applicant's total income. The range of applicant incomes is huge, with a minimum value of 25650, and a maximum value of 117,000,000. The average and mean values are 168797, with a standard deviation of 237122. As a result, you see that the graph is heavily skewed, with a small bump towards the left end of the range (the vertical bar is the median), and a long tail of continuous values extending out to the right of the range (Figure 4).

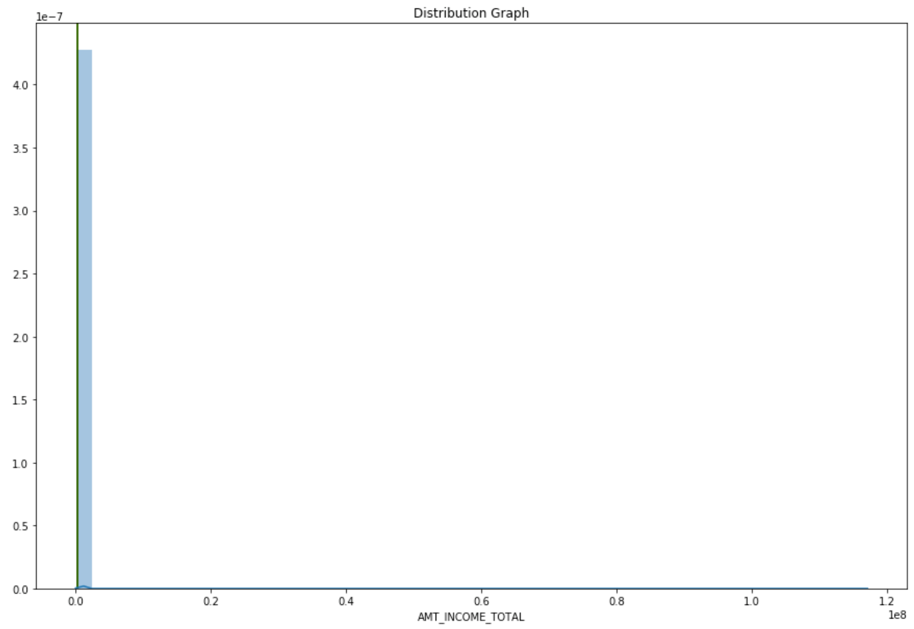


Figure 5 - AMT_INCOME_TOTAL Unprocessed

Financial data like this tends to be lognormally distributed, and by taking the log of the values, we can typically restore symmetry to our data. After processing, we now see a symmetrical curve, nicely centered around the median. In addition to the 61 continuous numeric fields, we identify another 11 fields for log transformation during the pre-processing phase.

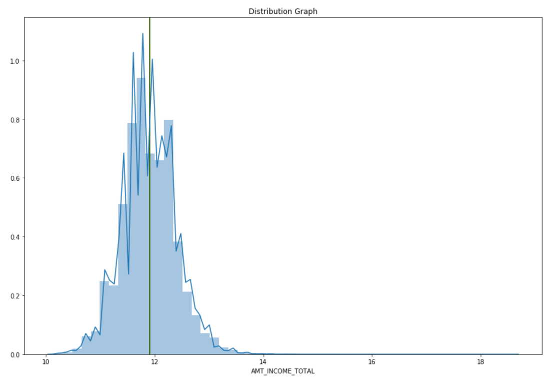


Figure 6 - AMT_INCOME_TOTAL - Log Transformation

Anomalous Data

This data exploration exercise revealed a handful of features that contained datapoints that were problematic.

DAYS_EMPLOYED is represented as a negative integer, relative to the date the local application was submitted. The field included values that indicated that some people had worked for 365243 days, or approximately 1000 years.
This seemed like they were probably a result of a bug, and in order to avoid penalizing the applicants for a software issue, the erroneous value was replaced with the Inter-Quartile mean, in an effort to avoid shifting the center of the values in the reasonable range.
A number of normalized housing data fields like "HOUSETYPE_MODE" contained categorical String values, with the exception of NaN when a value was not specified. This doesn't work when one-hot encoding categorical values, so the string value "not specified" was substituted.

Methodology

Preprocessing

Converted Date Values to Positive Integers

For some reason, the Home Credit team represents dates as negative integers, relative to the date that a loan application was submitted. A 20 year-old applicant would be representated as -7300 in the DAYS_BIRTH column. While we know that older applicants tend to be better about repaying loans than younger applicants, the negative values presented DAYS_BIRTH as a strong indicator of risk for default, instead of considering increasing age as a strong negative indicator. When comparing feature correlations, having the borrower's income as a contraindicator for default and age as an indicator seemed counter-intuitive, and I applied an absolute value transformation to the date features to deal with this.

Converted Two-Category String Features to Boolean

A number of logically Boolean features (e.g. “Does the borrower own a car?”, “Does the borrower own a house?”) were represented as “Y” or “N” instead of binary values. To aid in efficient processing, I employed the sklearn LabelBinarizer feature to convert these to integer 1 and 0 values, respectively.

One-Hot Encoding for Categorical Features

In instances where a feature has more than two categories (i.e. Gender: M (Male), F (Female), XNA (Decline to State / Not Applicable), you can choose to represent those values through label encoding (M=1, F=2, XNA=3); however, lacking context, the machine learning algorithm may interpret the integer value of 3 as being more important than an integer value of 1.

An alternative approach would be to create new features for each category, and provide a binary indication of which category is chosen as a 1 in the corresponding category.
This technique is known as one-hot encoding. While this technique increases the number of features in the dataset, it does not arbitrarily signal information that might potentially be interpreted as weighting to the machine learning algorithm. This is the approach that used for features with more than one category.

Categorical Encoding Examples

Original Encoding

ID	Gender
1	M
2	F
3	XNA

Label Encoding

ID	Gender
1	1
2	2
3	3

One-Hot Encoding

ID	Gender_M	Gender_F	Gender_XNA
1	1	0	0
2	0	1	0
3	0	0	1

Removed non-numeric values from numeric features

In order to scale a group of continuous numeric fields to a comparable range, those fields must exclusively contain numerical data.

In a number of instances, fields where numeric data was not supplied were populated with NaN (Not A Number) values. To facilitate further processing, those values were converted to 0 using the NumPy library’s nan_to_num() function, which converts NaN to 0, -inf to Python’s MININT, and inf to Python’s MAXINT values.

In subsequent iterations, I explored the impact of Imputation on the data. Instead of simply substituting missing data, imputation replaces missing datapoints following various strategies, including using the mean, median and most common value for a particular feature. The impact of these options, in combination with various scalers and outlier removal strategies are explored in depth in the accompanying Jupyter notebook, Step 3.1 - Validating Preprocessing Steps with Optimized Model and the spreadsheet labeled Preprocessing Analysis.

Outlier Removal

In looking at our dataset, it was clear that some numeric fields included exceptional values that skewed the mean for the majority of records. This is very apparent in AMT_INCOME_TOTAL and AMT_CREDIT, where some individuals earn and borrow sums that are orders of magnitude larger than the average borrower. It's unclear if these are data entry records or legitimate transactions performed by very wealthy individuals, but they're not useful for making predictions about the typical borrower.

Taking a cue from earlier projects, we perform outlier removal by taking the Inter-Quartile mean of our data (i.e. the mean of data between the 25th and 75th percentiles of the range), so as to remove any skew from values at the extreme upper or lower bounds of the range. This is also known as Tukey's method for outlier removal. We then substitute those mean values for values that exceed values above +/- 1.5 * the inter-quartile mean.

The result is that we remove the outliers without moving the mean of the data for the majority of records.

Feature Scaling

When machine learning algorithms evaluate numeric features, they may apply weight to larger numerical values. In order to ensure that the algorithm used applies equal weight to values of independent features, we put them all in a relative scale from 0 to 1. This scaling approach maintains the shape of the value distributions in each feature, while solving the problem of artificially weighting various features.

Because some fields may have contained outliers, the scikit learn RobustScaler class was initially considered to mitigate the effect of any outliers in the set; however, when evaluating the performance of our candidate algorithms between data preprocessed with RobustScaler and StandardScaler, and MinMaxScaler, StandardScaler actually performed slightly better in practice.

As a further sanity check, we also compared performance using data with and without log transformations. Interestingly, log transformed data performed poorly when applied to both financial data (loan amount, income amount) and to other numeric data that was identified as being more normally distributed under a log transformation during feature exploration.

Implementation

Preprocessing Implementation

The sci-kit learn library was used extensively for authoritative implementations of scoring, feature scaling and machine learning algorithms.

During data exploration, I created and manually classified arrays of feature names in categories that were relevant for preprocessing. Numeric features were features represented by continuous values, like the applicant's income and age. These features would be candidates for scaling and outlier removal. A subset of numeric features were fields that were heavily skewed, and appeared more normally distributed after taking the log of those values. Those features were also subject to scaling and outlier removal, in additional to log transformation. Categorical features like Gender ("M","F","XNA") were stored in the "non-numeric" array. Those features would later be one-hot encoded.

I did not initially implement Scaling or Outlier Replacement correctly, accidentally modifying values in a nonsensical way. After careful manual examination of the transformed data, it became clear that care needed to be taken to isolate those transformations to only appropriate fields.

I also ran into issues passing dataFrames around between functions. I ran into some quirks where dropping dataFrames didn't always happen consistently, and array-style accessors into dataFrames were problematic. This forced me to learn to access and modify data using dataFrame.loc() and dataFrame.iloc(), which ultimately resulted in much simpler and faster code, particularly for outlier removal, where field updates were previously not working, or working consistently.

To fully understand the impact of the available choices, I added a workbook that fully explores the various combinations of preprocessing steps on prediction performance. See the accompanying Jupyter Notebook, "Step 3.1 – Validating Preprocessing Steps with Optimized Model" and spreadsheet "Preprocessing_Analysis" for detailed test results.

Optimal Preprocessing Configuration

The combination of preprocessing steps that resulted in the best prediction performance was:

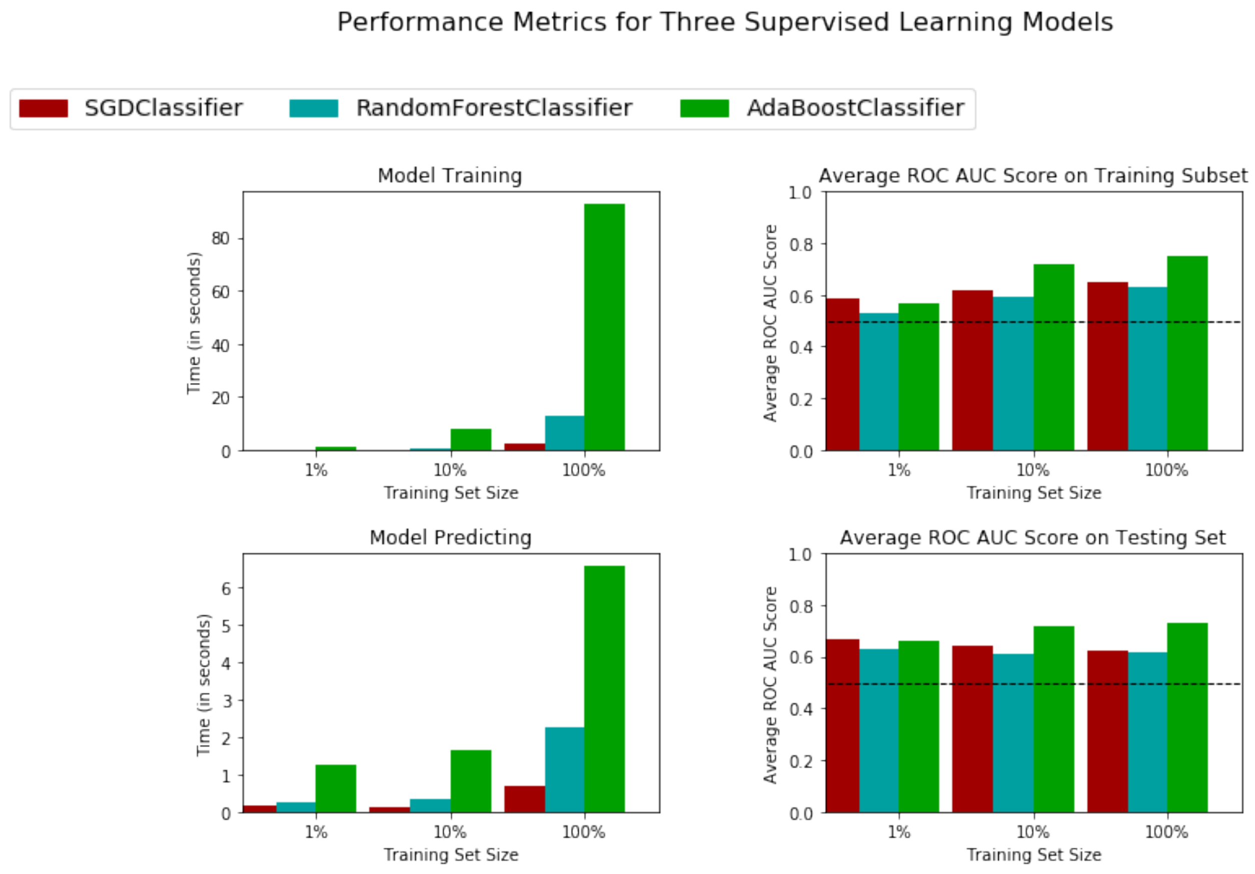
- Standard Scaler
- Outliers Removed
- Imputation (Mean)
- No Log transformations of any fields

Model Selection

Our evaluation showed that while AdaBoostClassifier is significantly slower than the other classifiers in our evaluation, it provides superior prediction performance when compared to the other algorithms used on our dataset. Because fast training and prediction speeds were not a key factor in our evaluation, AdaBoost was selected as the winning algorithm.

Each algorithm was evaluated against a testing and training set, generated from the labeled application_train set provided by Kaggle. For evaluation, we uses a 3-fold cross validation strategy, applied over the entire application_train set, with ‘roc auc” as the scoring engine.

Testing was conducted in three rounds, evaluating training and prediction performance when supplied with 1%, 10% and 100% of the application_train dataset. Evaluation time, training and prediction accuracy were logged, and scores were generated using the sklearn.



3-Fold Cross-Validated ROC AUC – Full Dataset		
Algorithm	Train	Test
SGDClassifier	0.6461246474007966	0.6254142444853207
RandomForestClassifier	0.6309923025120727	0.6141287602801629
AdaBoostClassifier	0.7472755600396243	0.7320643426988109

AdaBoost performed significantly better on predictive performance than SGDClassifier, and was selected as the algorithm for the project.

Goodness of Fit

In order to ensure that our algorithm is not either overfitting or underfitting the data, the ROC AUC score was compared against both the Training and Testing subsets (in this case, we used sklearn's test_train_split function to generate testing and training subsets of our labeled application_train.csv dataset), to rule out the possibility that we memorizing the training data and applying it poorly to test predictions.

As our visualization below indicates, each algorithm performs better than our naïve predictor (indicated by the dotted horizontal line), and we see reasonable performance against both the training and testing sets. We're not over-learning the training set, and our predictions exceed the performance of our naïve predictor.

Hyperparameter Tuning

In order to maximize performance of the AdaBoost algorithm, a number of hyperparameters need to be selected. In addition, since AdaBoost uses an ensemble of DecisionTreeClassifiers as its weak learner, hyperparameters for those algorithms also need to be tuned.
A cross-validating Grid Search was conducted over the AdaBoost hyperparameters for algorithm using the sklearn GridSearchCV class. The search compares the impact of each possible configuration on prediction performance as defined by the ROC AUC score.

Hyperparameters tuned included:

- For AdaBoost:
    - algorithm: [SAMME, vs SAMME.R]
    - n_estimators: [100,500]
- For Decision Tree Classifier:
    - Max Depth: [1,2,3,4,5]
    - Criterion: ["gini", "entropy"]
    - Max Features: ["auto", "log2"]

The SAMME algorithm is designed for multi-class boosting, and stands for Stagewise Additive Modeling using a Multi-class Experimental loss function[8]. The SAMME.R algorithm is similar, but typically converges faster than SAMME, with lower test error and fewer boosting iterations[9]. The n_estimators parameter defines the threshold at which boosting is terminated, and learning_rate is a value [0 .. 1], which shrinks the contribution of each individual classifer.

The n_estimators parameter determines the number of maximum number of DecisionTreeClassifier instances to invoke. The algorithm exits early in the event that an optional solutions is found before this threshold is reached.

[8] https://web.stanford.edu/~hastie/Papers/samme.pdf
[9] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

For DecisionTreeClassifier, max_depth determines the maximum depth of the underlying tree. The max_features parameter controls the number of features to consider when looking for the best way to split a node.

In order to minimize variance, we always seed random_state with a static value (42) whenever a method takes a random_state value, in order to ensure that our results are consistent between executions.

AdaBoost	ROC AUC Prediction Score – 3-Fold Cross Validation
Unoptimized	0.7320643426988109
Optimized	0.7487326537020242

Improving Performance

Once fit with training data, AdaBoost provides a list of the features it considers most important when making predictions via the feature_importances_ property. As a first step to improving performance, I took a look at the most important features to see if eliminating less important features might actually produce better predictions.

Dataset	Prediction Score (ROC AUC)
Most Important Features (Top-40)	0.5121526035127204
All Features	0.513152783064558

Predictive Performance is better with the full dataset at this point, and there's no pressing need to reduce the dataset for execution time, so the full dataset will continue to be used.

PCA Analysis

Following the thread of feature importance, I used the sklearn PCA class to conduct Primary Component Analysis over the most important features to understand what combinations of features factor into the decision making process. The graph is quite large, and can be found the Jupyter Notebook "Step 4 – Feature Selection – Application Data".

A number of these dimensions offer interesting insight:

- Dimension One appears to be a scenario where all indicators point to an applicant that is likely to default on the loan; however, AMT_CREDIT (the amount of the requested loan), AMT_ANNUITY (the annual loan payment) and AMT_GOODS_PRICE (the price of the item being purchased) dominate the composition of features.
- Dimension Two includes a large number of contra-indicators (possibly indicating a well-qualified applicant), with AMT_CREDIT, AMT_ANNUITY and AMT_GOODS_PRICE again being the large indicators of potential default.
- Dimension Three is dominated by the proprietary score EXT_SOURCE_2, DAYS_LAST_PHONE_CHANGE and DAYS_EMPLOYED as indicators of default.

- Dimension Four is dominated by DAYS_ID_PUBLISH and DAYS_EMPLOYED as likely indicators of default, when EXT_SOURCE_2 and EXT_SOURCE_3 offer strong contra-indications.

In general, we see all of these dimensions dominated by a handful of features, and a sense that the relationship between the amount of the loan, the amount of the loan payment and the price of the purchase send really strong signals as independent features.

Polynomial Feature Interactions

Continuing in the vein of important features, sklearn provides a PolynomialFeatures class that examines polynomial interactions between features. I used this class to add new features representing all possible permutations of the most common features from our PCA analysis to our dataset. I then retrained AdaBoost and looked at the feature importances to see if any new features factored into the top 20 features.

This resulted in the addition of five new features:

- EXT_SOURCE_2_X_AMT_GOODS_PRICE_X_AMT_ANNUITY
- AMT_CREDIT_X_AMT_ANNUITY
- EXT_SOURCE_3_X_AMT_GOODS_PRICE
- AMT_ANNUITY_X_EXT_SOURCE_1
- EXT_SOURCE_2_X_AMT_CREDIT_X_AMT_GOODS_PRICE

To evaluate their impact on performance, the dataset was re-scaled, a new instance of AdaBoost was trained, and predictive performance was compared. I performed this comparison with both full and reduced datasets, but the full dataset continued to outperform.

Dataset	Prediction Score (ROC AUC)
Important Features	0.5121526035127204
Important Features + New Features	0.5129962120328915
Full Dataset	0.513152783064558
Full Dataset + New Features	0.5161028439797717

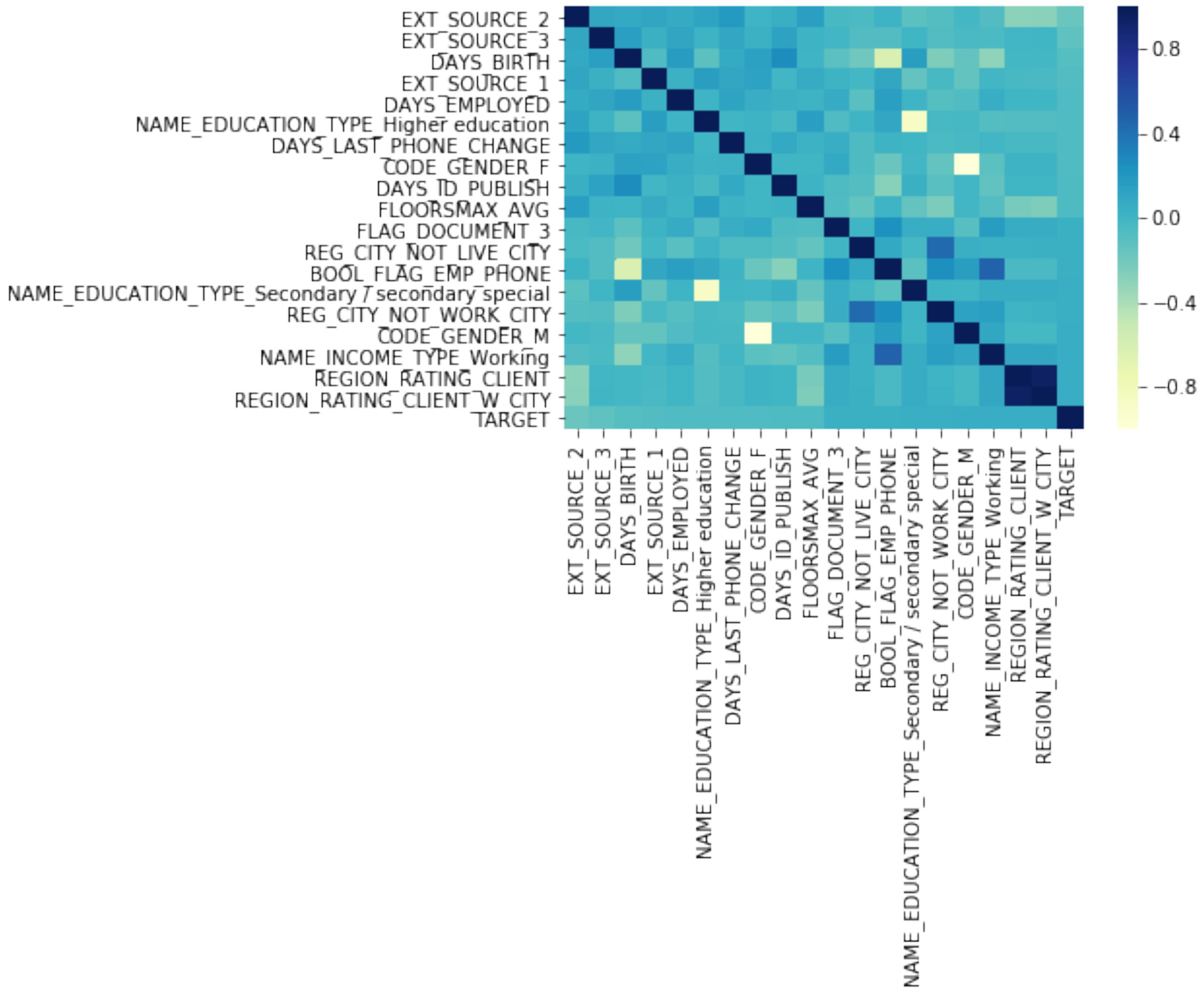
Examining Correlations

Having looked at important features, another interesting aspect to examine is which features are highly correlated. Using the DataFrame.corr() method, it's straightforward to calculate and visualize relationships between features in our dataset.

Using the Seaborn visualization library's heatmap feature, we can easily create a nice graphical illustration of those relationships.

Some interesting highlights include strong negative correlations (the white squares) between applicants who graduated high school and those that graduated college, men vs women, and the applicant's age, and whether or not they supplied a phone number for their employer (I presume that in practice, the employer's phone is an indicator of verifiable employment).

Strong positive correlations (dark squares) are shown between people "registered" in the city but don't work there, and people "registered" in the city that don't live there. In both instances, they have a commute to the city in common. There's a correlation between people that are employed as "working" and those that have an employer phone number, which intuitively makes sense. The age of a person's identification documents and their actual age is strongly correlated. My guess is that false identities probably have newer documents, and older applicants are more likely to pay loans in general, and would typically have well-established documentation.



These aspects are well-represented in our feature importance list already, but understanding the trends lends some confidence that I'm at least on the right track.

Feature Engineering

Many of the PCA dimensions explored earlier were dominated by the proprietary EXT_SOURCE_* features, and by various combinations of AMT_CREDIT, AMT_GOODS_PRICE and AMT_ANNUITY, appearing both in concert and opposition.

In thinking about what these signals might mean, it seemed like understanding the loan and loan payment as a percentage of a person's income might offer additional insight, as well as how much the person's down payment was on that commercial purchase (represented as the percentage of the purchase price borrowed).

When adding these features to the reduced dataset, PER_GOODS_CREDIT, PER_ANNUITY_INCOME and PER_CREDIT_INCOME all appear in the top 40 features.

Dataset	Prediction Score (ROC AUC)
Reduced Dataset + Percentage Features	0.5121526035127204
Full Dataset + Percentage Features	0.5166206159709185
Reduced Dataset (Baseline)	0.5104830686876065
Full Dataset (Baseline)	0.5161028439797717

The additional features provide a slight improvement to performance.

Incorporating Credit Bureau Data

To increase the score further, I looked at incorporating additional data about the customer into the model. Credit Bureau reports and balances from bureau.csv and bureau_balances.csv seemed like a potentially rich source for improvements.

Credit Bureau and Application data have a one-to-many relationship, and are associated by the shared key "SK_ID_CURR". Additionally, Credit Bureau and Credit Balances also have a one to many relationship.

In order to incorporate credit bureau data into our model for evaluation, I engineered a number of features that simply aggregate counts and balances. As an example, SUM_BUREAU_BALANCES is a total of all open account balances reported by the credit bureaus for a given customer. These features were generated, then merged into the corresponding application_* dataframes using the dataFrame.merge() method, using the shared key for association. New features were then modified using the same optimal preprocessing steps we identified earlier.

Dataset	Prediction Score (ROC AUC)
Final Application Dataset	0.5166206159709185
Application Dataset with Bureau Features	0.5161596689529163

Interestingly, adding the bureau features lowers our predictive performance.

Conclusion

In summary, this project attempts to determine whether a particular loan will be repaid by applying supervised learning to a historical set of loan data. Supervised Learning techniques are employed to classify loans as problematic by building an accurate predictive model based on the historical dataset.

The original dataset was examined, problematic records and outliers were normalized, and data was processed and structured in a manner suitable for machine learning. We then explored a number of potential learning algorithms, selecting the best one based on predictive performance and feasible execution time.

We ultimately employ the ensemble method AdaBoost to aggregate and weigh the results of multiple instances of simple learning algorithms to better tune and focus the model on difficult edge cases. Hyperparameters for the algorithm were tuned automatically, using Grid Search with cross-validated scoring to determine the best performance.

Once a model was selected and tuned, the individual features in the dataset were weighted for importance, and important features were examined through PCA analysis to see how features worked together to explain classifications. Correlations between datapoints were also examined to understand relationships between dependent features. Insights into those relationships led to the composition of new features that were relevant to the problem domain, which improved predictive accuracy of our model.

Finally, additional datasets with many-to-one relationships to the original application data were aggregated and added to our dataset to improve the predictive accuracy of our model.

While my model is not competitive in real-world scenarios at this point, the exercise was educational. I was able to explore new scoring methods (ROC AUC) and model selection techniques (cross-validated grid search). The project also required new skills for joining and merging multiple dataframes, as well as engineering and validating new features. A significant amount of troubleshooting and double-checking results led to a much deeper understanding of the Pandas, Numpy and Sci-Kit Learn libraries.

I intend to continue to refine this project in the future, with the goal of getting a competitive score in the Kaggle competition. The inclusion of additional features, deeper exploration of feature engineering (particularly around domain-specific insights), and the employment of automated feature selection techniques all seem like promising next steps.

There are still a number of additional datasets that can be incorporated into our model, which represent cash balances, credit card balances and the outcomes of previous loans issued to an individual by the lender. A full exploration of those datasets may reveal additional signals that could improve our score.

I’m also interested in using Deep Feature Synthesis. The Python FeatureTools library is of particular interest, and is primarily concerned with stacking and aggregating multiple features in a complex dataset. It explicitly addresses instances where the dataset is comprised of multiple tables with one-to-one and one-to-many relationships.

It’s also possible that other models may offer better performance. In particular, Gradient Boosting models like XGBoost (Extreme Gradient Boosting) show up a lot in winning solutions for Kaggle competitions, and would be interesting to explore further.