Santander Customer Transaction Prediction

Shayne Biagi
Bradley Blount
Kara Probasco
Virginia Polytechnic Institute and State University
Falls Church, Virginia, United States

ABSTRACT

This paper presents a full, end-to-end, machine learning project for the Santander Customer Transaction Prediction data set competition on Kaggle. We employ advanced feature engineering, tailored pre-processing, and a range of machine learning algorithms to predict customer behavior. Some of the models included are perceptron feed-forward networks, boosted tree ensembles, and logistic regressions.

Our models undergo various evaluations throughout, considering traditional classification metrics like accuracy and precision, with a greater focus on improving recall scores. Within the domain of this problem task, we made the assumption that prioritizing accurate predictions on the class of those who would make a transaction in the future was more crucial in real-world applications. Missing out on prospective transactions is worse than wrongly suggesting that a transaction 'may' occur in the corporate world. We prioritize interpretability and discuss the implications of our approach, providing insights for improved decision-making at Santander Bank. We also explored synthetic data generation as a possible solution to offset large class imbalances found in most real-world data sets. Our work contributes to the understanding of customer predictions and sets the stage for future advancements in similar contexts.

CCS CONCEPTS

• **Information systems** → **Data analytics**; • **Computing methodologies** → *Machine learning algorithms*; • **Applied computing** → *Enterprise modeling*.

Virginia Tech '23, Falls Church, VA,

KEYWORDS

neural networks, ensembles, synthetic data generation, banking, finance

ACM Reference Format:
Shayne Biagi, Bradley Blount, Kara Probasco. 2023. Santander Customer Transaction Prediction. In *Proceedings of Machine Learning with Big Data (Virginia Tech '23).* , 11 pages.

1 INTRODUCTION

The Santander Group is a Spanish multinational financial services company which ranks as the 19th largest banking institution in the world. Being such a large banking institution, they are commonly faced with problems regarding the likes of predicting customer satisfaction or financial health. So, their data science team is continually developing data analytics and machine learning tools to ensure that they can best solve these problems.

2 PROJECT PROBLEM STATEMENT

In 2019, Santander released a data set resembling their real customer's data in order to challenge the Kaggle community to develop and perfect machine learning models for accurately predicting whether a customer will perform a specific transaction in the future, regardless of the transaction's total amount.

In the most basic terms, our problem is to determine if a customer will make a purchase based on data about the customer. We are tasked with applying state-of-the-art machine learning models to predict the binary outcome of the aforementioned problem. We will apply various machine learning techniques, conduct feature engineering, and evaluate our models to formulate the most accurate predictions possible, with a greater focus on reducing false negatives. Our approach aims to help companies target potential customers more accurately.

Although accuracy will still be taken into account (to ensure that we build relevant models), we will primarily be looking to get high scores for recall. With the problem inherently involving analyses of historical data and patterns, data analytics is a well-suited approach. Also, as financial institutions like Santander commonly deal with vast amounts of customer data, data analytics and machine learning are the perfect candidates to handle and process these large data sets effectively.

Continuous model improvement is crucial in the modern era. It is always necessary to adapt to evolving financial trends as well as current state-of-the-art machine learning methods. Regular updates to training data, feature selection, and model parameters are integral for sustained relevance as well. Our goal is to leverage modern machine learning techniques to build adaptive solutions that go beyond simplistic predictions.

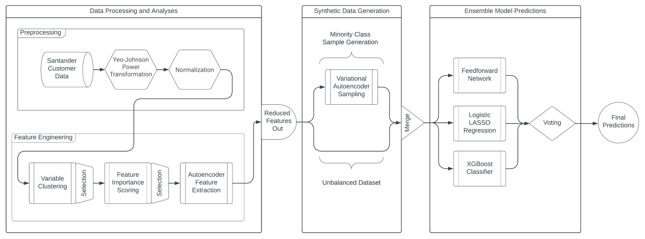
3 DATA SET

The data set provided by the Santander Group is a synthetic set which accurately depicts the structure of their real customer data for solving this task. For this reason, the data is anonymous and the features are unlabeled. This data set is accessible on Kaggle [4].

This data set consists of 200,000 instances of customer data, with each instance having a binary target column and 200 numerical features. The target column, which we are trying to predict, corresponds to whether the given customer will make a transaction in the future. The instances are named "train_X", where X falls between 0 and 199,999 inclusive. Features are named similarly as "var_Y", where Y falls between 0 and 199 also inclusive. We held out 10% of the original data to use as the test set during final evaluations. This data remained untouched until our final models were complete. Another 10% of the data was split off and used for validation of our methods, including tuning and feature engineering steps. This left 80% of the data left for training in those steps. We used 5-fold cross validation with the training set and validation set to evaluate out models during the training phase.

4 METHODS

4.1 Project Pipeline Diagram


Data Processing and Analyses
Preprocessing
Santander Customer Data
Yeo-Johnson Power Transformation
Normalization
Feature Engineering
Variable Clustering
Selection
Feature Importance Scoring
Selection
Autoencoder Feature Extraction
Reduced Features Out
Synthetic Data Generation
Minority Class Sample Generation
Variational Autoencoder Sampling
Unbalanced Dataset
Merge
Ensemble Model Predictions
Feedforward Network
Logistic LASSO Regression
XGBoost Classifier
Voting
Final Predictions

Pictured above is the complete data pipeline for this project. Our pipeline begins with data pre-processing, where the data is transformed and normalized for our models. Next, the data is pushed through the 3-step feature engineering section, where various dimensionality reduction techniques are employed to reduce complexity and possibly improve computational costs. Our reduced data set then enters the synthetic data generation stage, where minority class samples will be generated via sampling a learned distribution from a variational autoencoder to thwart the issues associated with class imbalance. Finally, our original data set along with the generated samples is then fed into an ensemble model to produce our final class predictions. The pipeline sections are described in more detail below.

4.2 Data Pre-Processing

During initial data exploration, many features in the data set were identified as having high variance measures. When features report high variances, machine learning models may be less able to identify meaningful patterns in the data set. In order to manage this, the Yeo-Johnson power transformation was applied to stabilize the variance across the data set. This allows our models to be more robust and more accurately capture the underlying relationships within the data.

Also, with the data set and its features being anonymized, we do not know the relative scales of the features. Varying scales may cause our models to be biased towards features with larger scales, possibly leading to incorrect predictions. After some investigation, we hypothesize that the data does include features with differing scales. Since we intend to use both gradient descent-based learning methods (logistic LASSO regression and neural networks) as well as distance-based algorithms (variable clustering) throughout our pipeline, we must account for this. To push our predictive models to learn from all features equally and without any biases, we have decided to apply normalization to the feature set via MinMax scaling. With this, all features will fall within the range [0,1].

4.3 Feature Engineering

4.3.1 *Variable Clustering.*

For the first part of our feature engineering process, we will utilize variable clustering for dimensionality reduction, leveraging the popular Python package VarClusHi. This package accomplishes variable clustering with a hierarchical structure. The algorithm automatically splits clusters by finding the first two principal components, performs an orthoblique rotation, and then reassigns features to the rotated principal component with the highest squared correlation with itself. The features are iteratively resigned until the algorithm has maximized the variance accounted for by the cluster components.

After the clustering algorithm has concluded, the features will be divided into clusters, from which the features with the lowest r-squared ratios will be selected to move on to the next step of selection. During this stage, we reduced the number of features from 200 to 71.

4.3.2 *Feature Importance.*

After variable clustering, we will use XGBoost, which uses an ensemble of boosted decision trees to estimate feature importance. Wherever the decision tree splits at a node, the algorithm computes gain by subtracting the summed node impurities of the child nodes from the impurity of the parent node. Then, this information gain is averaged over all trees which use this feature to split with. Finally, all features are ranked by their averaged gains, indicating to us the most important features for our given task.

With the feature importance scores, we can set a threshold to cut out any features below this given threshold. To know which threshold to use, we compute subsets of features by varying the threshold. These subsets were used to train and evaluate a simple Gaussian Naive Bayes classification model. With the set of evaluation metrics gained for each threshold, we choose the threshold which cuts out the most features without significantly reducing the models accuracy. This would cut out any unimportant features and further reduce the dimensionality of our feature set. Here, we reduced the feature set from 71 to 56 features.

4.3.3 *Feature Extraction.*

Here, an undercomplete autoencoder is constructed and trained to learn a compressed representation of the input data, or the latent space representation. Since our autoencoder is undercomplete, the latent space is a lower-dimensional form of the data, which in theory, captures all important characteristics of the original input data. The latent space is then served to the decoder, which is constructed as a slightly modified encoder in reverse. The decoder attempts to reconstruct the original data from the latent space. In training, we have used Mean Squared Error (MSE) as a loss function to measure the dissimilarity between the original and reconstructed forms of data. This reconstruction error, in our case, is the Mean Squared Error (MSE).

Our autoencoder is built according to three hyperparameters. These include: the input dimension, the latent dimension, and a reduction step ratio. The decoder is built very similarly to the encoder, but reversed. Thus, we will only detail the encoder build process here. Given these three hyperparameters, we compute the number of levels in the encoder required to get from the input dimension to the latent dimension by a constant percentage reduction. A discrete form of the compound interest formula is used. The equation to compute the number of hidden layers, L, given the input dimension and the latent dimension follows:

$$L = \left\lceil \frac{\log\left(latent\_dim/input\_dim\right)}{\log(1 - step\_ratio)} \right\rceil$$

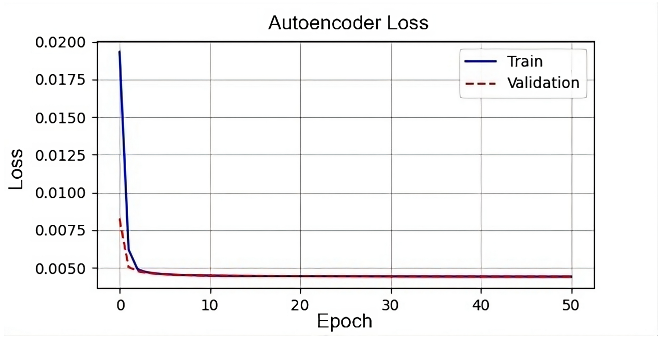
Additionally, if $L = 0$, L is set to 1 to place a lower bound on the number of hidden layers in the encoder network. That is, if the value of L is computed as 0, we will force the network to have a single hidden layer of dimensionality equal to the input layer. $input\_dim$ denotes the input dimension which dictates the size of the input layer in the encoder network. Finally, $latent\_dim$ denotes the size of the output layer and $step\_ratio$ represents the reduction step ratio. After the input layer, for each hidden layer, groups of Dense, BatchNormalization, and LeakyReLU layers are added with a constant reduction in size between levels until we reach the dimension of the latent space. Every subsequent level has a size of $step\_ratio\%$ less than the previous level. Finally, the network is capped off with a Dense output layer using a linear activation function.

In order to tune this architecture, a hypermodel is created to build and compile an autoencoder with hyperparameters in a defined range. The hyperparameter spaces are defined below, where HP_x is a hyperparameter with the subscript denoting the parameter name:

$$HP_{latent\_dim} \in \{x \in \mathbb{Z} \mid 1 \leq x \leq 56\}$$
$$HP_{step\_ratio} \in \{x \in \mathbb{R} \mid 0.05 \leq x \leq 0.9\}$$

As written above, the latent dimension can range from 1 to the number of features in the input data set. This maximum is set because at this point we do not know whether the feature set can be reduced any more without losing important information. The input dimension remains the same for all tuning trials. Also, the step ratio can range from a reduction of 5% each level, to a reduction of 90%. However, in order to maintain the constant downsizing of subsequent layers, if $step\_ratio > (1-(latent\_dim/input\_dim))$, the step ratio will be set to this same value. Finally, the hypermodel will also pick a learning rate within [1e-2, 5e-2, 1e-3, 5e-3, 1e-4, 5e-4] to compile the autoencoder model, using Adam as the optimizer. All hyperparameters are then tuned with HyperBand [3], setting max epochs to 350 and the reconstruction error (MSE) as the objective. This algorithm efficiently allocates resources to multiple configurations. It quickly identifies and promotes promising hyperparameter settings while eliminating poor performers with a successive halving strategy within brackets of different resource budgets. Hyperband leverages early stopping and focuses on a predefined loss metric for performance evaluation, making it particularly effective with limited computational resources.

After tuning, we can pull the optimal hyperparameters which minimize the reconstruction error. But, since our goal is dimensionality reduction, we then try to find the hyperparameters which minimize both the latent dimension and the loss. So, the validation loss and latent dimension are coupled in a weighted combination. Then, we take the hyperparameter set which minimizes this combination. The ideal hyperparameters were a latent dimension of 42, a step ratio of 0.534 and a learning rate of 0.0001. After the ideal hyperparameters were found, the autoencoder was retrained with these hyperparameters. The learning curve for the autoencoder training process follows:



Once this step reached completion, the dimensionality of the feature set was successfully reduced from 56 features down to a final 42. This means that our final set of features contained only 21% of the original 200, and retained most of its original structure. This also means that continuing from here, we will only have to consider 21% of the total size of our original data set, decreasing computational costs substantially.

4.3.4 *Synthetic Data Generation.*

The original data set has a hefty class imbalance of 90:10. 90% of the samples are examples of customers who did not perform a future transaction (the negative class), and 10% of the samples are examples of those who did (the positive class). Since we are looking to minimize the number of false negative predictions so that we do not miss out on the future customer purchases, we believed it would help to have more examples of that class.

In order to address this issue, we have constructed and tuned a variational autoencoder to generate synthetic samples for the minority class. Since variational autoencoders attempt to learn a latent distribution which generates the features themselves, we train a variational auto encoder on the minority class examples and then randomly sample this distribution to generate realistic examples for the minority class. These generated examples should be statistically similar to those in the original dataset. Thus, with these samples generated, we can train on more examples of the minority class and improve our model’s ability to accurately predict this class.

The structure of our variational autoencoder is mostly similar to a typical undercomplete autoencoder. The encoder compresses the original data into a compressed representation. This is where a traditional encoder would finish. However, since we are building a variational autoencoder, the compressed representation is then fed into two layers which represent the mean (μ) and variance (σ^2). This is so that the encoder can map the input data points to the parameters of the latent distribution. The mean and variance are then fed into a custom sampling layer to sample from this learned distribution. The sampling process can be written as:

$$x \sim \mathcal{N}(\mu, \sigma^2)$$

Here, x represents the sampled tensor and $\sim$ denotes the sampling process. x is sampled from the learned Gaussian distribution $\mathcal{N}$ with the mean μ and variance σ^2. The decoder is then built to decode this sample back into the original data.

For this to work, the variational autoencoder uses a loss function with two primary components: the reconstruction loss and the KL-Divergence loss. The reconstruction loss here is the same as the autoencoder above, using MSE as the error, capturing the difference between the original input data and the reconstructed data from the decoder. The KL-Divergence loss is added in order to enforce the latent distribution to follow a traditional Gaussian distribution. This is a regularizing term.

Our variational autoencoder, however, also includes a third loss term coined *pred_loss* or prediction loss. To compute this term, a loss model was created and trained on the entire encoded data set. To ensure a stable learning process, a deterministic LightGBM model was used here. It is important to note that this loss model is deterministic and does not introduce any randomness to the loss function. If randomness is introduced here, the training may become unstable, causing our final model to be imperfect at best and completely useless at worst. Like XGBoost, LightGBM is a variation of an ensemble of gradient-boosted decision trees. Then, when computing the loss, class predictions of the data from this model are compared with the class predictions of the reconstructed data from the decoder, using an accuracy score. Since this model was trained on only samples of the minority class, assuming a perfect loss model, this accuracy would be perfect if all of the decoded samples were also predicted as of the minority class. This would, in theory, ensure that our generated samples would highly resemble those in the original data set. And thus, also ensuring that we generate quality samples. This accuracy score is then converted into an error score (1 - accuracy) to serve as a loss. This component of the total loss function was added to push the variational autoencoder to generate samples which also resemble those from the original data set, improving the quality of our generated samples. It is important to note that the loss model was only able to achieve 0.903 in accuracy when training, meaning that at worst, 0.3 percent out of the 10% of positive class samples were caught by this model. This presents an issue, as we cannot be sure that our loss model fully understands what comprises a positive class sample, placing a hard ceiling on how well our model can generate realistic samples. With a perfect loss model, this would not be an issue, however, with a perfect loss model we also would not have to synthetically generate data. This problem is paradoxical. The issue is covered more in the future work section at the end of the paper.

The architecture of the encoder and decoder - apart from the mean, covariance, and sampling layers - were tuned similarly to the autoencoder above. For the encoder, hidden layers were constructed from the input layer to just before the mean and covariance layer, ensuring a constant step reduction in size between levels. The decoder, would then be constructed in the opposite manner, increasing in size by the same constant reduction until the original input dimension in reached. The latent dimension, step ratio, and learning rate were then all tuned using the Hyperband algorithm [3]. The optimal hyperparameters found were a latent dimension of 6, a step ratio of 0.150 and a learning rate of 0.0005 for Adam (the optimizer).

Once this model was tuned and trained, the latent distribution was sampled to produce a set of generated samples for the minority (positive) class. Then, some random Gaussian noise was applied to each sample to ensure robust models down the line. After the noise was added, the sampled data was included with our original data to train the final models. Originally, we had tried a 50-50 class distribution, meaning that we would be training our models on a very large amount of synthetic samples. To achieve this class balance, our data set’s positive class samples would be overwhelmingly comprised of synthetic samples. After some evaluations on a validation set, which showed a tendency for our models to over-predict the positive class now, we decided to decrease the amount of synthetic samples added to our data set. With further analysis, we were able to find that including 66315 synthetic samples in our data proved to yield the best results on our validation set. This meant that our final training set had 66.5% of samples for the negative class (customers who did not make a future transaction), and 33.5% of samples for the positive class (customers who made a future transaction). This rebalanced set was the set used to train and tune the final classifier models.

4.3.5 Ensemble Model.

With the synthetic data generation complete and the synthetic samples added to the original training set, we compiled all of our final tuned models into an ensemble. This ensemble includes a Multi-Layer Perceptron (feed-forward network) model, a Logistic LASSO Regression, and an XGBoost classifier. Using weighted soft voting, the positive class probabilities output by each of the individual models are used to compute a weighted average for the final class probability prediction. This probability is then rounded to get the final class prediction. In order to find the optimal weights to give each model in the ensemble, the additive inverse of the recall score for the validation set was minimized with Sequential Least Squares Programming (SLSQP). SLSQP is an iterative optimization algorithm used for constrained nonlinear optimization problems. The weights for our case are constrained, since they must all add up to one. This algorithm sequentially fits a quadratic model to the objective function and constraints in order to efficiently find a local minimum or maximum. Although we may miss a global maximum for recall, the efficiency enough was worthwhile for our team. By computing a weighted average of the class probabilities, we hoped that the ensemble could leverage the predictive powers from each individual model for an accurate and robust final prediction. Each model in the ensemble has unique strengths and weaknesses, such as the intermediary neural network and XGboost models being able to more accurately predict on the negative class while the Logistic LASSO Regression was better at predicting on the positive class. With an ensemble, we consider all of these perspectives. The varied characteristics of each model ensure a balanced and robust final prediction.

5 MODELS AND RESULTS

5.1 Baseline Classification Models

5.1.1 Models.

To log a starting baseline for this project, we implemented a couple of popular, but simple, classification models. These models were trained on the original, unprocessed data set. The models implemented for this include: Gaussian Naive Bayes, Logistic Regression, Decision Tree, and a Random Forest Classifier.

5.1.2 Results.

For all models, confusion matrices along with precision, recall, and f1 scores were computed. All evaluations were done using 5-fold cross validation. We used stratified folds to ensure that the validation sets are all representative of the class balance in the original data set. The reported metrics are the average across all folds and the confusion matrix cells are the sum over all folds. The baseline model evaluations are compiled in the tables below. Each row of the table represents one of the models, with columns for all cells in the summed confusion matrix or the averaged evaluation metrics. Also, note that, for the confusion matrices, the positive class denotes the customer's who will make a transaction in the future. All evaluation metrics are reported with precision up to three decimal places.

Summed Confusion Matrices for Baseline Models

Classifiers	TN	FP	FN	TP
Naive Bayes	159310	2622	11463	6625
Log Reg	159817	2095	13745	4343
Decision Tree	146796	15116	14252	3836
Random Forest	161912	0	18084	4

Evaluation Metrics for Baseline Models

Classifiers	Precision	Recall	F1
Naive Bayes	0.716	0.366	0.485
Log Reg	0.675	0.240	0.354
Decision Tree	0.202	0.212	0.207
Random Forest	0.600	0.000	0.000

These baseline models proved to yield poor performance on our classification task. Precision values were decent, but recall and f1 scores were quite low. After this, we were able to draw some conclusions and move on to implementing our data pre-processing and feature engineering steps. With the processed and engineered data set, we narrowed our focus to some more complex model types with the understanding gathered from our baseline implementations. We made the decision to increase our models' complexities in hopes of better performance.

5.2 Intermediary Baseline Models

5.2.1 *Models.*

Regarding the next set of models implemented, we decided on three, first of which was a regularized logistic regression. From our initial research and literature review, this model showed promise [2]. Also, from the initial baseline models, logistic regression was not entirely useless. We decided on a logistic regression model which uses L1 regularization in hopes that it would better learn more complex patterns in the data. Lasso (L1) regularization adds a penalty term to the logistic regression cost function and encourages some coefficients to become zero, effectively setting certain features to have no impact on the model, and, in some way, performing automatic feature selection. Additionally, as the decision tree alone performed better than the random forest model, we looked at XGBoost as an alternative ensemble model. Our thinking was that we may have chosen the wrong ensemble type. Random Forest builds multiple decision trees independently and combines their predictions (bagging), while XGBoost builds a sequence of trees, where each tree corrects the errors of the previous ones (boosting). This difference could allow us to harness the abilities of the single decision tree while creating a model which is more capable of complex feature correlations. Finally, building from our baseline logistic regression implementation again, we decided that multiple decision boundaries were something to explore. We chose to build from this and implement a single-layer perceptron classifier, with a hidden layer size equaling that of the input data and an output layer of size 1 with a sigmoid activation function. The network uses binary cross entropy as its loss and is optimized via Adam. Every perceptron can be seen as a single linear regression (similar to a logistic regression in shape), so having more in a network should provide us with the complexity which the data appears to require. In most machine learning tasks, a single layer neural network can be enough for good performance, so this was a good place to start.

5.2.2 *Results.*

What follows is the results we gained from our intermediary models. The evaluations are reported in the same manner as above, with one table detailing the confusion matrices of our models and another detailing their precision, recall, and f1 scores. Again, the matrices are reported as the sum over all folds, and the evaluation metrics are averaged across folds. Negative refers to the zero label (customers who did not make a future transaction) and positive refers to the one label (customers who did make a future transaction).

Summed Confusion Matrices for Intermediary Models				
Classifiers	TN	FP	FN	TP
SLP Classifier	161381	531	17305	783
Log LASSO	161735	177	17751	337
XGBoost	160654	1258	17110	978

Evaluation Metrics for Intermediary Models			
Classifiers	Precision	Recall	F1
SLP Classifier	0.608	0.043	0.080
Log LASSO	0.656	0.019	0.036
XGBoost	0.435	0.055	0.098

Here, we can see that our evaluation metrics did not improve across the board. In general, our precision scores lowered a bit, but our recall scores ended up taking a larger hit from our previous bests. Looking closer at the confusion matrices, our models did not mis-predict actual negative-class samples as of the positive-class. In fact, our intermediary models had less errors of that type (Type I) as compared to the baseline models. Instead, it was observed that, the intermediary models were failing to predict negative-class samples, and overly predicting them as the positive class. In simpler terms, as compared to the baselines, our intermediary models had less false positives but more false negatives. This was the root of our slightly lower precision scores, but much lower recall scores. Since our primary focus of this project was to achieve high recall, meaning that we would have to decrease the amount of mis-classified positive samples (Type II errors), we synthetically generated more examples of the positive class for our models to learn from (see section 4.3.4). In addition to this, we supposed that tuning the hyperparameters of our models after including the synthetically generated samples in the data would improve performance overall.

5.3 Final Models

Our intermediary models were predicting the negative (or 0) class, at worst, 99.2% correctly. So, our feature engineering and preprocessing improved our predictive power of the negative class, but, in turn, decreased our predictive power of the positive class. Thus, our issue seemed to lie in the fact that our data set simply did not have enough examples of the positive class to learn from. So, from here we applied synthetic data generation to re-balance our data set to have 66.5% positive samples and 33.5% negative samples, as aforementioned in section 4.3.4.

5.3.1 *Tuning.*

Next, we tuned our models with the new synthetically generated samples alongside the original data. Both the Logistic LASSO Regression and XGBoost Classifier models were tuned via grid search with cross validation. Grid search automatically computes different combinations within the entire specified hyperparameter space and evaluates them with K-fold (in our case 5-fold) cross validation. This is an exhaustive search technique which guarantees optimality within the defined hyperparameter spaces, but due to our limited computational resources exploration is limited in this method.

XGBoost has many parameters that can be optimized. However, due to our limited computation resources, we selected leaning rate, maximum depth, subsample, and miminum child weight as our optimization parameters. Subsample refers to the ratio of training instances that will be sampled prior to each boosting round. Minimum child weight sets a threshold for partitioning, requiring the sum of instance weight in a leaf node to surpass this threshold for further partitioning to occur.[1] The optimal parameters for our XGBoost model are 0.1 for learning rate, 11 as the max depth, 0.5 as the subsample, and a 5 for the minimum child weight.

For the Lasso regression, the regularization strength parameter ’C’ was explored in the grid search. The best hyperparameter obtained through grid search is ’C’: 0.001, indicating the optimal regularization strength. Lasso regression uses L1 regularization, which adds a penalty term equivalent to the absolute value of the magnitude of coefficients. A smaller ’C’ value indicates a stronger regularization, encouraging the model to use fewer features and prevent overfitting.

To tune the architecture of the perceptron classifier, Bayesian optimization was used along with a hypermodel which built the hidden layers according to various hyperparameters. Here, we explore the idea of a multi-layer neural network. Initially, the hypermodel chooses a parameter detailing the number of hidden layers in the network, which can range from 1 to 3. Then for each hidden layer it adds, the hypermodel will choose between batch or layer normalization techniques, and then choose between standard rectified linear unit (ReLU), leaky rectified linear unit (Leaky ReLU), or exponential linear unit (ELU) activation functions. After the activation function, a dropout layer is placed, and the dropout rate is also chosen. Finally, the network (with input and output layers added) is compiled with Adam, and the learning rate is chosen. Bayesian optimization was chosen for its ability to adapt to non-convex hyperparameter spaces. Bayesian optimization involves modeling the loss function with a Gaussian Process, incorporating observations, and iteratively optimizing the acquisition function to guide the search for the global optimum. Letting $f(x)$ be the loss function, where x is the hyperparameter configuration, the loss function can be modeled as a Gaussian Process (GP):

$$f(x) \sim \mathrm{GP}(\mu(x), \sigma^2(x))$$

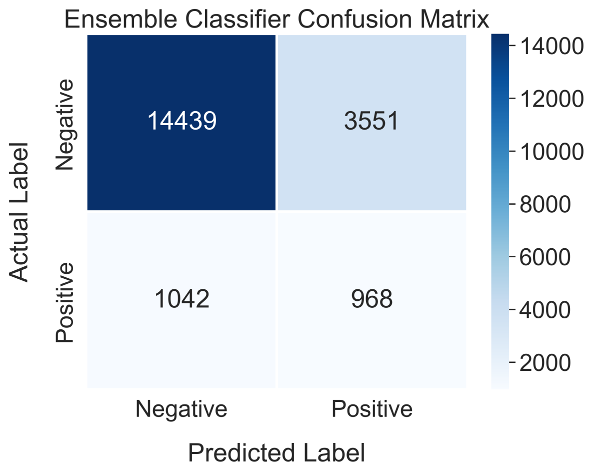
Here, $\mu(x)$ is the mean function and $\sigma^2(x)$ is the covariance function. Then, with the evaluated sets of hyperparameters, the algorithm uses its observations to update the Gaussian Process into a posterior distribution and iteratively optimize the acquisition function (upper confidence band in our case) to achieve optimal hyperparameters. In the end, the optimal architecture was found to be 3 hidden layers with sizes 80 > 96 > 48, no dropout, and batch normalization with LeakyReLU activation for each layer. The input and output layers remained the same from the single-layer perceptron implementation. The optimal number of layers found was the upper bound to the number of layers set beforehand, so we may not have reached the global optimum here. Due to the limited computational resources we had, we were forced to keep the hyperparameter search space small, but this is definitely something to consider. This issue is also covered a bit in the future work section.

After tuning, all models were trained with their optimal hyperparameters on the data from both the training and validation sets, including the support of the synthetically generated samples.

5.3.2 *Results.*

After the synthetic data generation and tuning was complete, an ensemble model was created, containing each of the tuned models. This process was detailed in the ensemble section above.

For the final evaluation, the ensemble model was evaluated on the 10% of the original data set held out at the beginning of the project. The data was processed through the pipeline, completing the same pre-processing and feature engineering steps as the training data. This set, however, was not infused with any synthetic samples to ensure an accurate evaluation. Below is the confusion matrix along with precision, recall, and f1 scores for the ensemble model on the holdout testing set.



Evaluation Metrics:

- Precision: 0.214
- Recall: 0.482
- F1: 0.297

Our final ensemble model returned the highest recall scores yet, predicting almost half of the positive class samples correctly. This is a significant boost from our previous highest recall score of 0.366. Unfortunately, this iteration still takes a sizeable hit in precision when compared to our baseline models and even intermediary models. This is the precision-recall trade-off. Improving our recall scores has costed us our precision scores, and therefore, predicting correctly on the positive class has caused our models to predict more poorly on the negative class. Despite this, it is provable that the synthetic data generation and tuning has boosted our models' abilities to correctly predict positive class samples and reduce Type II errors overall. As this was our primary focus in the project, this was an expected side effect. It is to our understanding that we could achieve higher performance in this regard with an improvement to our data generation techniques, allowing the models to more precisely differentiate the two classes.

In the context of our project, this trade-off between recall and precision proves valuable. In the real world, the choice between emphasizing recall or precision depends on the consequences associated with false positives and false negatives in the given application environment. With our task, we assumed that better recall would be more important. On the other hand, in scenarios where false positives result in significant costs or the like, prioritizing precision may be more beneficial. This evaluation should not only shed light on the ensemble's strengths and weaknesses, but also serve as a basis for informed decision-making in this field, allowing stakeholders to align model performance with specific objectives within the constraints of the real-world problem it addresses. Continual refinement is necessary to strike the optimal balance between recall and precision for any specific application.

6 FUTURE WORK

6.1 Improvements in Synthetic Sample Generation

In the synthetic data generation section, we described the issue associated with using an imperfect model to ensure sample quality from our data generation method. Unfortunately, due to this, our ability to generate a wide variety of realistic samples was limited. In order to assess this, we would have to explore other means of evaluating sample quality. One possibility is to simply use a different method for generating samples. With variational autoencoders, we are limited to the types of samples we can generate, and training on only minority class samples may not have best revealed what differentiates the positive class samples from the negative class samples. For future work, conditional variational autoencoders may prove to be more effective. These models are able to generate different types of data samples based on specific conditions or information. So, we could train the model on the full data set, grasping the features which best differentiate the classes, and conditionally generate samples for just the minority class. In theory, we could generate samples for any class we have, but generating samples for the positive class would not be useful for our limited computational budget. Additionally, we could utilize more complex generative models, such as diffusion models for this purpose.

Another possible improvement could arise from moving the synthetic data generation step to the beginning of our data pipeline. This would mean that we generate samples for the minority class before any data preprocessing or feature engineering takes place. With this approach, the generated data may better comprise of key information required to best identify the class label, circumventing any biases present in our data preprocessing or feature engineering strategies.

6.2 Hyperparameter Space Exploration

Due to our limited computational budget, we were also quite limited in the size of the hyperparameter spaces we explored in tuning. Our personal machines were simply unable to handle the workload necessary to search larger spaces. Steps were taken in order to try and limit computational costs, such as using the hyperband tuning algorithm [3], but we still ran into many issues due to our large data set. We used cloud computes from Azure for most of the larger tuning runs, but that quickly became expensive. In the future, it may prove extremely valuable to do more research in low-cost tuning methods which correctly balance exploration and exploitation of our hyperparameter spaces and possibly return better tuned models.

7 CONCLUSION

This project serves as an exploration of the modern machine learning landscape and its applications. Throughout our work, we faced a variety of unforseen issues, all of which required in-depth assessments and significant overhauls. Conclusively, the implementation of a soft-voting ensemble on the Santander Customer Transaction Prediction data set, along with its carefully selected member models, has proven to be a powerful approach for enhancing predictive modeling accuracy and robustness. The iterative approach to model building and tuning taken here has also contributed heavily to the search for predictions which balance accuracy while retaining an important focus on recall. In the corporate atmosphere, nobody wants to miss out on the chance to identify prospective customer transactions. Shifting our focus to best formulate a solution which does not exactly align with achieving high overall accuracy showcases the innate flexibility of machine learning. Furthermore, we experimented with leveraging generative models tailored specifically to our needs, addressing an extremely common issue faced by modern model developers, class imbalance.

The results of this experimentation highlighted the effectiveness of various feature engineering techniques, ensemble methods, and generative AI in facing the challenges posed by complex and imbalanced data sets. While our approach showcased promising results, there is always room for further exploration and refinement. Future endeavors may involve experimenting with different ensemble techniques, exploring more feature engineering strategies, and fine tuning our approach to synthetic data generation.

Our implementations not only reveal new avenues of perspective on common machine learning tasks, but also hold practical implications for businesses seeking to optimize predictive modeling for their own customer satisfaction needs. As machine learning technologies continue to evolve, the insights gained here contribute to the ongoing pursuit of more adaptable and robust machine learning models in the domain of finance and beyond.

8 INDIVIDUAL CONTRIBUTIONS

8.1 Shayne Biagi

- Constructed and tuned Autoencoder for Feature Extraction
- Constructed and tuned Variational Autoencoder for Synthetic Data Generation
- Constructed, evaluated, and tuned Neural Network Classifiers
- Constructed, evaluated, and found optimal weighting for the final Ensemble Model
- Designed Pipeline Diagram
- Project Management
- Technical Report Writing for proposal, milestone, and final papers

8.2 Bradley Blount

- Implemented Feature Engineering via VarClusHi and KMeans w/ Silhouette Scores
- Constructed and evaluated Logistic Regression and Naive-Bayes baseline models

- Constructed, evaluated, and tuned LASSO Regression classifier
- Constructed and evaluated the soft voting Ensemble Model

8.3 Kara Probasco

- Implemented data preprocessing steps (Yeo-Johnson and Minmax Scaling)
- Implemented feature importance dimensionality reduction
- Constructed and evaluated Decision Tree and Random Forest baseline models
- Constructed, evaluated, and tuned XGBoost Classifier
- Technical Report Writing for proposal and milestone and proofreading/editing for final paper

REFERENCES

[1] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[2] Devendra Prakash Jaiswal, Srishti Kumar, and Partha Mukherjee. 2020. Customer transaction prediction system. *Procedia Computer Science* 168 (May 2020), 49–56. https://doi.org/10.1016/j.procs.2020.02.256

[3] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research* 18 (Mar 2016), 1–52.

[4] Banco Santande S.A. 2018. Santander Customer Transaction Prediction. https://www.kaggle.com/c/santander-customer-transaction-prediction/data