Santander Value Prediction: Kaggle 2018

Cheng-Hao Tai

September 6, 2018

Abstract

This report exhaustively details all my analysis, preprocessing, feature engineering, and predictive modeling efforts undertaken throughout the course of this Kaggle challenge. The sections below cover the fields of tree-based boosting methods (CatBoost, LightGBM, and XGBoost), neural-network architecture (stacked auto-encoder), covariate shift correction via the Kullback-Leibler Importance Estimation Procedure, and time-series feature mining from an anonymized dataset. Through a series of 10 successive models and several rounds of ensembling, my project partner and I finished in the top 6% of over 4,400 competitors to earn our first bronze medal.

1 Overview

The Santander Value Prediction Challenge is the Santander Group's 3rd Kaggle competition and offers the Kaggle community an opportunity to assist Santander in improving their customer service capability by predicting the value of transactions for each potential customer. The project was kicked-off on June 18, 2018 and wrapped-up on August 20, 2018. A total of around 4,000 teams participated in this challenge.

For the competition dataset, Santander provided an anonymized set of tabular training and test data with the following dimensions (note that the training set's extra column is the target variable):

- Training Set: 4459 Rows, 4992 Columns
- Test Set: 49342 Rows, 4991 Columns

The given datasets presented unique challenges (to be elaborated on in the following sections) that provided an excellent sandbox for honing my skills relating to analyzing, processing, and designing predictive models with respect to tabular data. Throughout the course of this project, I embarked on a total of 3 different modeling stages with accompanying substages. On a high level, these stages can be described as:

- Stage 0: Initial Exploration and Public Kernels
- Stage 1: Covariate Shift Correction
- Stage 2: Time-Series Unraveling
  - 1st Half: Inspired by Public Kernels
  - 2nd Half: Massive Breakthrough and Final Push

The remainder of this report will proceed in a chronological fashion — detailing the various stages and substages as they occurred within the timespan of the challenge. As you read through this report, please feel free to skip over sections as desired. As the stages are being detailed, I will also point out the locations of my Jupyter notebooks and Python scripts with respect to the main Github repository so that you may look over them or run them to replicate my results.

2 Stage 0: Initial Exploration and Public Kernels

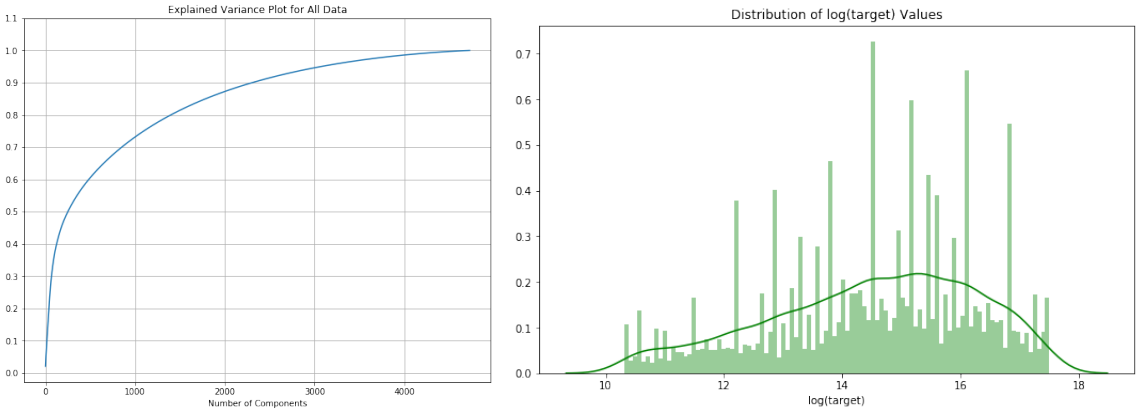
Stage 0 and its accompanying substages focused on establishing solid footing in this competition. To that end, all the models produced in this stage took inspiration from top-scoring public kernels so to efficiently identify the best strategies for dealing with the Santander datasets. All due acknowledgements for the authors of the public kernels are included within the referenced Jupyter notebook files.

2.1 Substage: Data Exploration and Vanilla Preprocessing

Notebook Location: `./scripts/DEPP_Vanilla_S0.ipynb`

This first substage aimed to yield an introductory understanding of the Santander Value Prediction dataset and to conduct preliminary preprocessing. The main steps undertaken in this substage are:

- Removal of train set's constant columns from train and test
- Removal of train set's duplicate columns from train and test
- A PCA analysis of the combined training and test dataset which found that approximately 3,000 features account for 95% of the dataset's variance (see Figure 1a)
- A visualization of the training label distribution (see Figure 1b)



(a) Percent Variance Explained

(b) Target Variable Distribution

Figure 1: Vanilla Preprocessing Visualizations

The Figure 1 visualizations generated in the vanilla preprocessing substage begin to reveal the inherent challenges of this dataset. First, the Explained Variance plot does not reveal a sharp elbow indicating the presence of a few critical low-dimensional features. Rather, the plot smoothly curves towards a plateau. What this means is that dimensionality reduction cannot be undertaken without a non-negligible loss of information. Furthermore, the visualization of the target variable shows a fairly even distribution of $log(target)$ values which echos the concern expressed by the Explained Variance plot. The relatively uniform nature of these target variables imply that there aren't natural clusters within the dataset that can be leveraged for predictive value.

Dataset	Percentage of Nonzero Values
Training	3.1458
Test	1.4052

Table 1: Data Sparsity in Train and Test Set

The issues highlighted by the visualizations in Figure 1 are clarified when looking at the sparsity of the dataset (i.e. the percentage of nonzero values). A quick analysis reveals the sparsity figures shown in Table 1. Looking at these percentages, it is immediately obvious that an overwhelming majority of values in the train and test datasets have zero value. As such, it is little wonder that Figure 1a revealed that there was little underlying structure to be leveraged for dimensionality reduction.

Together, the sparsity and apparent lack of underlying structure in the Santander dataset are the two main challenges to be tackled in this Kaggle challenge. The following Stage 0 models all aimed to navigate around these issues.

2.2 Model 0v0: Tuned LightGBM

Notebook Location: `./scripts/Model_0v0_TunedLGBM.ipynb`

Model 0v0 utilizes the dataset generated from my vanilla preprocessing substage (Section 2.1) to train a LightGBM boosting algorithm to quickly obtain a baseline result to kick-off the competition. The main challenge presented in Model 0v0 was the selection of hyper-parameters for the LightGBM regressor, specifically the learning rate and the number of boosting iterations. The following strategy was used to find the best learning rate and obtain predictions:

1. For each learning rate in a range of learning rates (from large to small), train a LightGBM regressor in a cross-validation manner (using LightGBM's CV API) and record the following statistics for each regressor model:
    - *Optimal Boosting Rounds:* The number of iterations corresponding to the lowest RMSE value for the current regressor model
    - *Score:* The lowest RMSE value for the current regressor model
    - *Parameters:* The LightGBM hyper-parameters that defined the current regressor model
2. From the list of recorded statistics, choose the set of statistics corresponding to the lowest RMSE to obtain the:
    - Optimal Boosting Rounds
    - Best Set of Hyper-Parameters for LightGBM Regressor
3. Using the tuned hyper-parameters, train and make predictions using separate LightGBM Regressor models initialized using a range of random seed initializations to obtain separate sets of final predictions
4. Average over these final prediction sets to obtain a single well-generalized set of final predictions

2.3 Model 0v1: Auto-Encoder / Dimensionality Reduction with CatBoost

Source Code Locations:

- **Main Notebook:** `./scripts/Model_0v1_Encat.ipynb`
- **Auto-Encoder Notebook:** `./scripts/autoencoder/nb_autoencoder.ipynb`
- **Auto-Encoder Code:** `./scripts/autoencoder/autoencoder.py`

After getting a foothold on the public leaderboard with Model 0v0, I proceeded to begin directly tackling the challenges identified in Stage 0's exploratory substage. Model 0v1 simultaneously addresses both the high-sparsity and indeterminate structure problems through the usage of multiple dimensionality-reduction techniques. In total, 6 different methods of dimensionality reduction were used in Model 0v1:

- **Stacked Auto-Encoder (built using Keras):** Reduction down to 156 features
- **KMeans Clustering:** Reduction down to 128 features
- **PCA:** Reduction down to 128 features
- **Truncated SVD:** Reduction down to 128 features
- **Gaussian Random Projection:** Reduction down to 24 features
- **Sparse Random Projection:** Reduction down to 24 features

By using a variety of dimensionality reduction techniques, I was aiming to tackle the Santander dataset's lack of any clear low-dimensionality structure. Even though the PCA analysis conducted in Stage 0's exploratory substage was unable to obtain promising results, it was still highly possible that there remained non-linear structures that PCA was unable to identify. As such, by using this varied cocktail of dimensionality reduction strategies, I hoped to extract these non-linear artifacts and reduce the original datasets into a leaner form with more concentrated predictive capacity.

Following the dimensionality-reduction cocktail, a CatBoost regressor algorithm was used to obtain final predictions. As far as the mainstream boosting algorithms go, the CatBoost algorithm is an emerging option that is popular for its robust handling of categorical features, fast training times, and relatively low need for extensive hyper-parameter tuning.

2.3.1 Explanation of Stacked Auto-Encoder

Every dimensionality-reduction strategy was implemented via SKLearn's API with the sole exception of the stacked auto-encoder which I implemented from scratch using Keras. As such, this was the most powerful non-linear dimensionality reduction algorithm that I used in Model 0v1 that was also the most intuitive / transparent to me (and thus merits a short explanation).

A vanilla auto-encoder functions by training a dataset on itself in a 2-layer neural network structure. The first layer of an auto-encoder network implements the *encoding* step which reduces dimensionality while the second layer implements the **decoding** step which casts the first layer's output back into the original dataset's dimensionality. The usage of nonlinear activations in the encoding and decoding layers of an auto-encoder mean that an auto-encoder is capable of projecting data onto non-linear axes (as opposed to PCA which is limited to linear projections). Throughout each round of forward pass and back-propagation through the vanilla architecture, the neurons of the encoding layer converge to these low-dimensionality features (linear and non-linear) such that the information loss between the original and the encoding are minimized.

While a vanilla auto-encoder probably would have sufficed in capturing some non-linear low-dimensionality structures within the Santander dataset, I wanted to improve upon the robustness of the vanilla auto-encoder

by extending the vanilla structure into a stacked architecture. A stacked auto-encoder is essentially a cascaded series of vanilla auto-encoders where the output of each vanilla network's encoding layer is set as the input of the next vanilla network. As data passes through the stack, its dimensionality is steadily reduced all the way through the final auto-encoder of the stack. By separating the vanilla structure into a multi-tiered stack, I was aiming to decrease dimensionality in a gradual and controlled manner (rather than the steep descent that a vanilla architecture would've demanded) and thereby achieve a more robust final projection.

2.3.2 Model 0v1 Process Flow

1. Scales vanilla preprocessed datasets to zero mean and unit variance
2. Transforms combined train and test datasets via stacked auto-encoder
3. Performs dimensionality reduction using SKLearn cocktail
4. Trains CatBoost regressor model on training set with 5-fold cross-validation and make predictions

2.4 Model 0v2: XGBoost with Pipelined Feature Extraction

Source Code Locations:

- **Main Notebook:** `./scripts/Model_0v2_XGPipe.ipynb`
- **Python Code:** `./scripts/Model_0v2_XGPipe.py`

NOTE: *Model 0v2 is written in the style dictated by the SKLearn pipeline API. As such, it does not use the vanilla preprocessed dataset generated in Stage 0's exploratory substage (Section 2.1).*

Model 0v2 introduces a new strategy of feature preprocessing than the previous two models in Stage 0. So far, Model 0v0 established a baseline score using minimal feature engineering while Model 0v1 tested the utility of dimension reduction on the dataset. Deviating from these two models, Model 0v2 implements the following process flow in a completely pipelined manner (using SKLearn pipeline API):

1. Identifies train set columns that have constant values and removes these columns from the training set and the test set
2. Identifies duplicate columns within the train set and removes these columns from the training set and the test set
3. Transforms dataset through the following algorithms:
   (a) Fit a PCA dimensionality reduction model on the train data down to 100 dimensions and transform both the train and the test datasets *(Note: even though this operation cuts off a significant amount of information, the reduction result can still have utility as metadata features)*
   (b) Fit a custom predictive classifier model for transforming a dataset into binned target values (classes) along with class probabilities on the train dataset and transform both the train and test datasets (detailed explanation below)
   (c) Define a pre-set cocktail of statistical functions and transform both the train and test datasets using these pre-defined functions *(Note: These functions operate on rows, not columns)*:
      - Length

- Minimum
- Maximum
- Median
- Standard Deviation
- Skew
- Kurtosis
- 19 Percentile Values (5 to 95%)

(d) Aggregate the above transformations into a single train set and a single test set

4. Train a XGBoost regressor on the training set and make predictions on the test set

Model 0v2's claim-to-fame is expressed in item (3) in the above process flow - specifically item (3b) and (3c). Item (3c) is straightforward enough to not warrant a detailed explanation. Simply put, each row of the training and test sets have row-wise statistical values calculated. Subsequently, these row-wise statistical values are used as metadata features when training and predicting using the XGBoost regressor.

On the other hand, item (3b)'s algorithm is less transparent. Essentially, (3b)'s algorithm is manifested in a custom Python class that utilizes a Random Forest Classifier to generate metadata features from a vanilla dataset. On a high level, the inclusion of item (3b) in the process flow has the effect of creating a more robust model by chaining machine learning algorithms together.

On a more granular level, item (3b)'s algorithm can be explained as follows:

1. Receive **the number of desired classes (or bins)** as input
2. Create a new vector of targets that have been transformed into bin values / number of desired classes *(i.e. if 2 classes / bins are desired, then the target values would be transformed to either 0 or 1 if they are below or above the median target value)*
3. Fit a Random Forest Classifier with 5-fold cross validation on the training set and the transformed target vector
4. Transform the training set and the test set by taking in a dataset and using the trained random forest classifier to obtain new metadata features:
    - 1st Feature: Class predictions (integer values)
    - 2nd Feature / Set of Features: Class probabilities (float values in the range $[0, 1]$ that denote the likelihood of a certain class *(Note: This is actually a set of features ranging anywhere between 2 to however many desired classes / bins are specified)*

This algorithm can be used to obtain metadata features from the train and test sets for any number of desired classes / bins. For the purposes of Model 0v2, I called upon this algorithm four different times for 2, 3, 4, and 5 desired classes / bins.

2.5 Stage 0 Ensembling

Notebook Location: `./scripts/Blender.ipynb`

At the conclusion of Stage 0, my three models yielded public leaderboard scores as shown in Table 2.

Model	Public Leaderboard Score
0v0	1.44
0v1	1.40
0v2	1.39
Ensemble	1.38

Table 2: Stage 0 Model Performances

In an effort to produce better results, I decided to ensemble the three models I developed in Stage 0 by averaging their prediction scores. This ensemble-aggregation procedure has the effect of reducing the influence of outlier / deviant model behavior through the combination of independent models. The result was a well-generalized result that outperformed any individual model.

3 Stage 1: Covariate Shift Correction

When I finished Stage 0, I had reached the limit of inspiration that the Kaggle community could provide (at the time). As such, Stage 1 marks a departure from models derived from public kernels and into completely uncharted and original territory.

One of the early observations I made in Stage 0 was that my training score would consistently be higher than my actual public leaderboard score — despite careful hyper-parameter tuning (see Figure 2).

```
Learning rate: 0.012
[200]   cv_agg's rmse: 1.4344 + 0.0334958
Optimal Round: 284
Optimal Score: 1.42870101453 + 0.034950291968
Learning rate: 0.008
[200]   cv_agg's rmse: 1.45785 + 0.0314299
[400]   cv_agg's rmse: 1.42848 + 0.0341142
Optimal Round: 445
Optimal Score: 1.42693548377 + 0.034321978767
Learning rate: 0.016
[200]   cv_agg's rmse: 1.42889 + 0.0343656
Optimal Round: 195
Optimal Score: 1.42871922782 + 0.0341465784269
```

Figure 2: Sample LightGBM Training Log Output from Model 0v0

Referencing Table 2, we can see that the public leaderboard was 1.44 while — for varying learning rates — Model 0v0 consistently outperforms the public leaderboard score when predicting on the validation sets. This behavior can be observed in every model created in Stage 0 and made me wonder whether or not this discrepancy between validation and test scores is a result of a systemic disparity between the training and test set features.

Investigating this trend, I discovered the concept of **Covariate Shift Correction**: an algorithm that is designed to compensate for feature incongruence between the train and the test sets. The key intuition behind covariate shift correction can be understood as:

$$P(x_{test}) \neq P(x_{train})$$
$$P(Y \mid x_{test}) = P(Y \mid x_{train})$$

That is, even though the train and the test features deviate from each other, they have the same predictive function / behavior.

Covariate shift correction is an active area of research and there are numerous proposed algorithms / optimizations to implement. However, I chose to implement the **Kullback-Leibler Importance Estimation Procedure (KLIEP)** algorithm detailed in the research paper *Direct Importance Estimation with Model Selection and Its Application to Covariate Shift Adaptation* authored by Sugiyama, Nakajima, Kashima, von Bunau, and Kawanabe. Two of the main reasons why I chose to implement this particular algorithm are because (1) of Sugiyama and Kawanabe's (two of the most recognizable names in the field of covariate shift correction) involvement along with (2) the fact that the intricacies of this algorithm are very clearly detailed within the referenced paper.

3.1 A High-Level Intuition of KLIEP

Fundamentally, the KLIEP algorithm asks the question: *How similar or dissimilar is any training sample to the test set distribution?* The similarity metric is expressed in terms of a weighting term that is used to scale an individual training sample's loss contribution.

The core of KLIEP relies upon pre-defined basis functions that can describe (through distance values) how likely a training sample is to have come from the test distribution. In KLIEP, this is implemented through the selection of an appropriate kernel function and the subsequent careful tuning of the kernel function's hyper-parameters. For my implementation of KLIEP, I chose to use the Gaussian Kernel and tuned two hyper-parameters: (1) the number of kernels used and (2) the Gaussian Width.

Figure 3 is a visualization of the Gaussian Width hyper-parameter on loss-scaling weights.

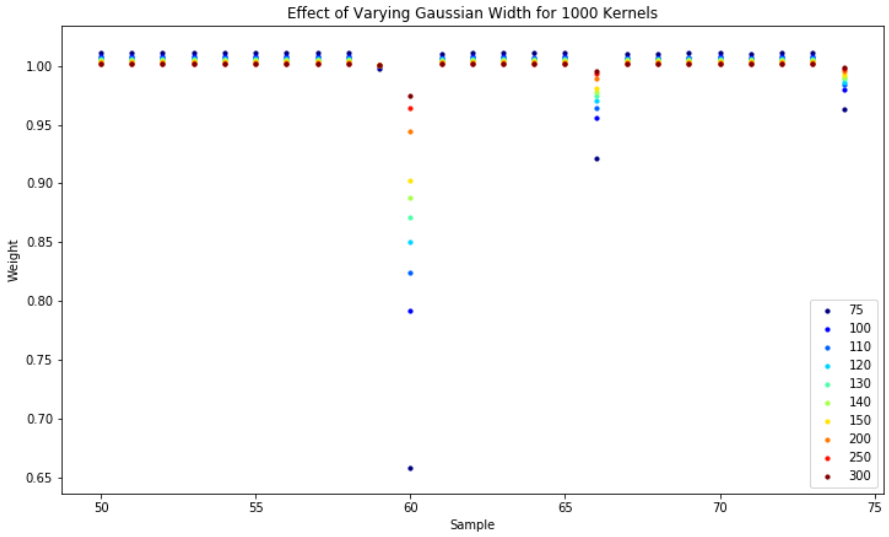


Figure 3: Effect of Varying Gaussian Width with 1000 Kernels

From the spread of weight values for each training sample shown in Figure 3, it can be seen that increasing Gaussian Width has the effect of decreasing the spread of weight values. Intuitively, this makes sense. Higher values of Gaussian Width traditionally correspond to the idea of higher variance (or, a more dispersed

distribution). In the context of KLIEP, higher Gaussian Width values correspond to a looser criteria for proximity. A smaller Gaussian Width value has the opposite implication.

Therefore, using a high Gaussian Width has the undesired effect of decreasing the KLIEP model's discriminatory ability between training and test distributions. However, using an excessively low Gaussian Width has the catastrophic effect of round-to-zero errors since the division by a small number increases the negative magnitude of the Gaussian Kernel's exponentiated term. As such, a critical aspect of constructing a KLIEP model is finding the optimal, balanced Gaussian Width value.

A great attribute of KLIEP is that it has a concave objective function (its derivation is detailed in the research paper I've referenced above). This optimization objective (denoted by J) is calculated using only the test samples but must obey constraints set by training samples and therefore takes both train and test distributions into account. Throughout the hyper-parameter tuning process, the best-performing model is chosen as the one that maximizes this optimization objective. Figure 4 illustrates how the hyper-parameter tuning process influences J.

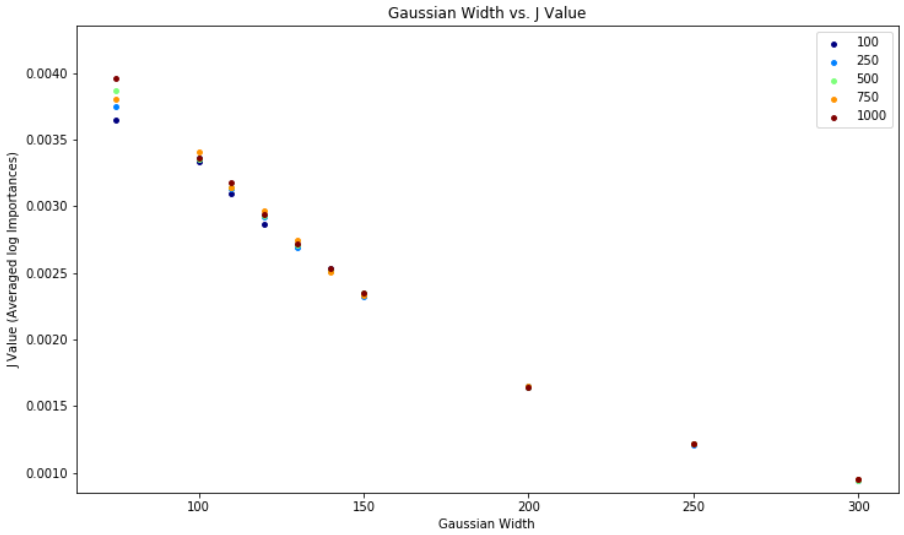


Figure 4: Effect of Gaussian Width on J Value

Looking at Figure 4, it can be seen that the optimization objective is tightly clustered for larger values of Gaussian Width but becomes well-separated as the Gaussian Width decreases in value. At the lower range of Gaussian Widths in Figure 4, we see that there exists a correlation between higher numbers of kernels and the objective score. This result is well-aligned with the intuition established in Figure 3 where a lower Gaussian Width was determined to correspond with a stricter notion of proximity (which in turn, improved the model's discriminatory ability).

As a final note, my KLIEP implementation was based on pseudocode provided within the referenced research paper. A snapshot of this pseudocode is shown in Figure 5.

3.2 Model 1v0: XGBoost with Covariate Shift Correction

Source Code Locations:

Input: $m = \{\varphi_\ell(\boldsymbol{x})\}_{\ell=1}^{b}$, $\{\boldsymbol{x}_i^{\mathrm{tr}}\}_{i=1}^{n_{\mathrm{tr}}}$, and $\{\boldsymbol{x}_j^{\mathrm{te}}\}_{j=1}^{n_{\mathrm{te}}}$
Output: $\widehat{w}(\boldsymbol{x})$

$A_{j,\ell} \longleftarrow \varphi_\ell(\boldsymbol{x}_j^{\mathrm{te}})$;
$b_\ell \longleftarrow \frac{1}{n_{\mathrm{tr}}} \sum_{i=1}^{n_{\mathrm{tr}}} \varphi_\ell(\boldsymbol{x}_i^{\mathrm{tr}})$;
Initialize $\boldsymbol{\alpha}$ $(> \mathbf{0})$ and ε $(0 < \varepsilon \ll 1)$;
Repeat until convergence
  $\boldsymbol{\alpha} \longleftarrow \boldsymbol{\alpha} + \varepsilon \boldsymbol{A}^\top (\mathbf{1}./\boldsymbol{A}\boldsymbol{\alpha})$;
  $\boldsymbol{\alpha} \longleftarrow \boldsymbol{\alpha} + (1 - \boldsymbol{b}^\top \boldsymbol{\alpha})\boldsymbol{b}/(\boldsymbol{b}^\top \boldsymbol{b})$;
  $\boldsymbol{\alpha} \longleftarrow \max(\mathbf{0}, \boldsymbol{\alpha})$;
  $\boldsymbol{\alpha} \longleftarrow \boldsymbol{\alpha}/(\boldsymbol{b}^\top \boldsymbol{\alpha})$;
end
$\widehat{w}(\boldsymbol{x}) \longleftarrow \sum_{\ell=1}^{b} \alpha_\ell \varphi_\ell(\boldsymbol{x})$;

(a) KLIEP main code

Input: $\mathcal{M} = \{m_k \mid m_k = \{\varphi_\ell^{(k)}(\boldsymbol{x})\}_{\ell=1}^{b^{(k)}}\}$,
$\{\boldsymbol{x}_i^{\mathrm{tr}}\}_{i=1}^{n_{\mathrm{tr}}}$, and $\{\boldsymbol{x}_j^{\mathrm{te}}\}_{j=1}^{n_{\mathrm{te}}}$
Output: $\widehat{w}(\boldsymbol{x})$

Split $\{\boldsymbol{x}_j^{\mathrm{te}}\}_{j=1}^{n_{\mathrm{te}}}$ into R disjoint subsets $\{\mathcal{X}_r^{\mathrm{te}}\}_{r=1}^{R}$;
for each model $m \in \mathcal{M}$
  for each split $r = 1, \ldots, R$
    $\widehat{w}_r(\boldsymbol{x}) \longleftarrow \mathrm{KLIEP}(m, \{\boldsymbol{x}_i^{\mathrm{tr}}\}_{i=1}^{n_{\mathrm{tr}}}, \{\mathcal{X}_j^{\mathrm{te}}\}_{j \neq r})$;
    $\widehat{J}_r(m) \longleftarrow \frac{1}{|\mathcal{X}_r^{\mathrm{te}}|} \sum_{\boldsymbol{x} \in \mathcal{X}_r^{\mathrm{te}}} \log \widehat{w}_r(\boldsymbol{x})$;
  end
  $\widehat{J}(m) \longleftarrow \frac{1}{R} \sum_{r=1}^{R} \widehat{J}_r(m)$;
end
$\widehat{m} \longleftarrow \mathrm{argmax}_{m \in \mathcal{M}} \widehat{J}(m)$;
$\widehat{w}(\boldsymbol{x}) \longleftarrow \mathrm{KLIEP}(\widehat{m}, \{\boldsymbol{x}_i^{\mathrm{tr}}\}_{i=1}^{n_{\mathrm{tr}}}, \{\boldsymbol{x}_j^{\mathrm{te}}\}_{j=1}^{n_{\mathrm{te}}})$;

(b) KLIEP with model selection

Figure 5: KLIEP Pseudocode as Taken from Sugiyama

- **Main Notebook:** `./scripts/Model_1v0_XGB_CS.ipynb`
- **Python Code:** `./scripts/Model_1v0_XGB_CS.py`
- **Covariate Shift Weight Generation Notebook:** `./scripts/covariate_shift/covariate_correct.ipynb`
- **Covariate Shift Weight Generation Python Code:** `./scripts/covariate_shift/covariate_correct.py`
- **Covariate Shift Weight Analysis Notebook:** `./scripts/covariate_shift/cs_analysis.ipynb`

Model 1v0 took the results of the KLIEP weight generation procedure and applied the tuned weights to the loss-update procedure integrated in a customized XGBoost regressor model. Since the KLIEP-generated weights are meant to be used to scale training loss from one boosting iteration to another, I implemented a custom XGBoost loss (aka objective) function that scaled each training sample's loss by its corresponding KLIEP weight.

3.2.1 Model 1v0 Process Flow:

Model 1v0 utilizes the same pipeline structure, data preprocessing methods, and feature engineering strategies as Model 0v2. The only major difference between these two models is that Model 1v0's implements a custom XGBoost regressor. The process flow for Model 1v0 is as follows:

1. Remove duplicate and zero-variance features in the same manner as Model 0v2
2. Apply Model 0v2's feature engineering to the data (PCA, Random Forest Classifier transformation, etc...)
3. Train a custom XGBoost model using a custom loss function
4. Make predictions on the test set using the trained XGBoost model

3.3 Stage 1 Results and Discussion

Model	Public Leaderboard Score
1v0	1.45

Table 3: Stage 1 Model Performance

3.3.1 Discussion of Results:

Unfortunately, the KLIEP algorithm in conjunction with XGBoost did not yield stellar results — falling short of all the models generated in Stage 0. However, there are several possible reasons explaining this disappointing result:

- As was revealed in Stage 0's exploratory substage (Section 2.1), a major challenge of the Santander datasets is their extreme sparsity and overall lack of low-dimensional structure
  - Since KLIEP works by matching training features' distributions to test features, a lack of structure to guide the matching process along with severe sparsity means that any matches learned by KLIEP may not be reliable
- The KLIEP weights used in Model 1v0 was most likely not optimized, despite the tuning that went into generating those weights
  - KLIEP weights were generated and optimized using the vanilla dataset from Stage 0's exploratory substage (which saw minimal preprocessing)
  - Model 1v0's data is preprocessed in an entirely different way (with a pipeline of PCA, random forest classifiers, etc...) which means that at best, those two datasets were loosely correlated
  - The fact that Model 1v0 ended up performing only slightly worse than Model 0v0 implies that the KLIEP weights used Model 1v0 were unsuitable and that nontrivial performance gains might be expected by re-training the KLIEP weights on a version of the dataset with proper preprocessing
- As will be further explained in Section 4 below, the Santander training and test datasets have features that are actually scrambled time-series data
  - KLIEP's distribution-matching process involves making distance calculations along feature dimensions and works best when features are largely independent from each other
  - With time series data (even in its scrambled form), the high interdependence of features means that KLIEP may have greatly struggled in discriminating between train and test samples
- After the time-series nature of the datasets came to light, I decided to stop pursuing covariate shift correction and instead focus on reconstructing the time-series
- While KLIEP was fundamentally not suitable for the Santander datasets, I'm confident that there are myriad other applications that will benefit from my KLIEP implementation and I look forward to testing my covariate shift correction algorithm again in the near future

4 Stage 2: Time-Series Unraveling

Around the middle of July (2018), there was a massive upset in the Santander competition leaderboard where it was revealed that the datasets provided by Santander were actually column-wise scrambled time-series data. From that point onwards, a majority of the public Kaggle kernels focused on locating features in the training and test datasets that could help in the reconstruction of this time-series. Proposed feature sets and feature orderings have been made public-knowledge on these forums. Figure 6 depicts a snapshot of a public kernel that demonstrates this phenomenon. Looking at Figure 6, it can be seen that the **target** column of the table is simply the first feature column shifted by two time steps (henceforth referred to as a **two-step-offset**). As such, by reconstructing the time-series such that the trend identified in Figure 6 is revealed, it becomes possible to obtain excellent prediction results even without applying machine learning.

ID	target	f190486d6	58e2e02e6	eeb9cd3aa	9fd594eec	6eef030c1	15ace8c9f	fb0f5dbfe	58e056e12	20a
7862786dc	3513333.3	0	1477600	1586889	75000	3147200	466461.5	1600000.0	0.0	44(
c95732596	160000.0	310000	0	1477600	1586889	75000	3147200.0	466461.5	1600000.0	0.0
16a02e67a	2352551.7	3513333	310000	0	1477600	1586889	75000.0	3147200.0	466461.5	16(
ad960f947	280000.0	160000	3513333	310000	0	1477600	1586888.9	75000.0	3147200.0	46(
8adafbb52	5450500.0	2352552	160000	3513333	310000	0	1477600.0	1586888.9	75000.0	31(
fd0c7cfc2	1359000.0	280000	2352552	160000	3513333	310000	0.0	1477600.0	1586888.9	75(
a36b78ff7	60000.0	5450500	280000	2352552	160000	3513333	310000.0	0.0	1477600.0	15(
e42aae1b8	12000000.0	1359000	5450500	280000	2352552	160000	3513333.3	310000.0	0.0	14(
0b132f2c6	500000.0	60000	1359000	5450500	280000	2352552	160000.0	3513333.3	310000.0	0.0
448efbb28	1878571.4	12000000	60000	1359000	5450500	280000	2352551.7	160000.0	3513333.3	31(
ca98b17ca	814800.0	500000	12000000	60000	1359000	5450500	280000.0	2352551.7	160000.0	35(
2e57ec99f	307000.0	1878571	500000	12000000	60000	1359000	5450500.0	280000.0	2352551.7	16(
fef33cb02	528666.7	814800	1878571	500000	12000000	60000	1359000.0	5450500.0	280000.0	23(

Figure 6: Time-Series Reconstruction by the Kaggle Community

An important point to take note of is that the only way to obtain the exact time-series reconstruction shown in Figure 6 is to have a precise selection and order of features and rows. Figure 7 shows the features, rows, and their orderings that will yield the tabular data shown in Figure 6.

```
Row indexes(36)
 [1757, 3809, 511, 3798, 625, 3303, 4095, 1283, 4209, 1696, 3511, 816, 245, 1383, 2071, 3492,
378, 2971, 2366, 4414, 2790, 3979, 193, 1189, 3516, 810, 4443, 3697, 235, 1382, 4384, 3418, 43
96, 921, 3176, 650]
Column indexes(40)
 ['f190486d6', '58e2e02e6', 'eeb9cd3aa', '9fd594eec', '6eef030c1', '15ace8c9f', 'fb0f5dbfe',
'58e056e12', '20aa07010', '024c577b9', 'd6bb78916', 'b43a7cfd5', '58232a6fb', '1702b5bf0', '32
4921c7b', '62e59a501', '2ec5b290f', '241f0f867', 'fb49e4212', '66ace2992', 'f74e8f13d', '5c648
7af1', '963a49cdc', '26fc93eb7', '1931ccfdd', '703885424', '70feb1494', '491b9ee45', '23310aa6
f', 'e176a204a', '6619d81fc', '1db387535', 'fc99f9426', '91f701ba2', '0572565c2', '190db8488',
'adb64ff71', 'c47340d97', 'c5a231d81', '0ff32eb98']
```

Figure 7: Rows and Columns to Generate Figure 6 Time Series

4.1 Substage: Kaggle Result Validation

Source Code Locations:

- **Feature Identity Validation Notebook:** `./scripts/time_series/similarity_with_target.ipynb`
- **Block Extension Notebook:** `./scripts/time_series/feature_extender.ipynb`

Before fully-trusting the Kaggle community's findings shown in Figures 6 and 7, I decided to conduct my own independent investigation to see whether or not I could arrive at the same result. Using the newfound knowledge that the dataset is a scrambled time-series, I tested an intuition that the most important features must be ones that share the most common values with the target variable. The result of this analysis is visualized in Figure 8.

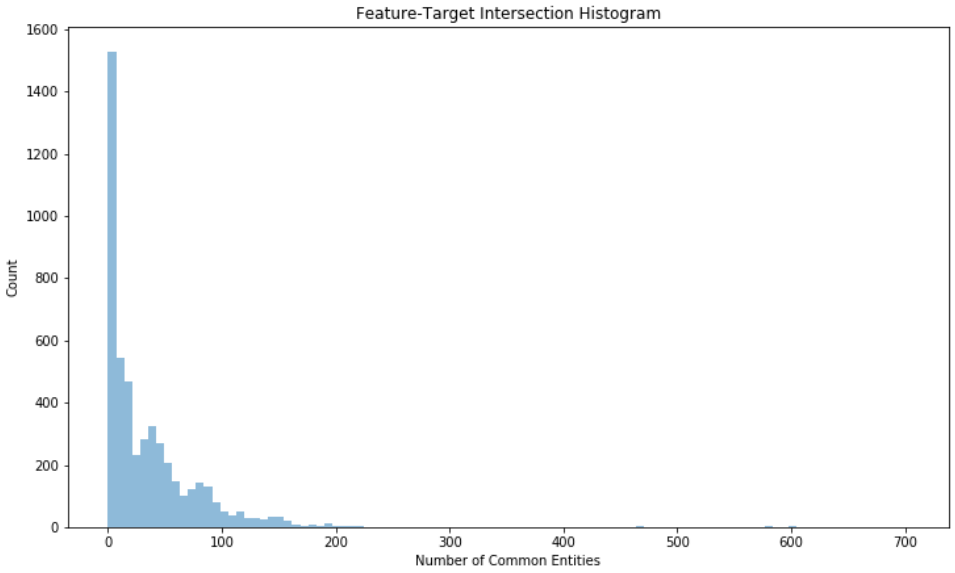


Figure 8: Histogram of Number of Common Values Between Features and the Target

Figure 8 shows a long-tail distribution implying that most features do not share many common values with the target variable. However, the tail-end of this long-tail distribution suggests that there are a couple of features that share a large percentage of their values with the target variable. Looking into this further, I find that when I filter for features that share **more than 300 features** with the target variable, I get a list of features (see Figure 9) that are identical to those shown in Figure 7.

Comparing Figure 7 to Figure 9 will reveal that while both lists contain the same features, Figure 7's ordering differs from Figure 9's. The disparity in ordering is due to the fact that Figure 9's feature sequencing is taken straight from the original Santander dataset whereas Figure 7's feature order has been reshuffled to a perfect diagonally-aligned time-series block. Yet despite their difference in ordering, the fact that the lists of features in Figure 7 and Figure 9 contain identical elements (despite them having been mined using different strategies) provides a partial validation to the Kaggle community's analysis.

```
array(['20aa07010', '963a49cdc', '26fc93eb7', '0572565c2', '66ace2992',
       'fb49e4212', '6619d81fc', '6eef030c1', 'fc99f9426', '1db387535',
       'b43a7cfd5', '024c577b9', '2ec5b290f', '0ff32eb98', '58e056e12',
       '241f0f867', '1931ccfdd', '58e2e02e6', '9fd594eec', 'fb0f5dbfe',
       '91f701ba2', '703885424', 'eeb9cd3aa', '324921c7b', '58232a6fb',
       '491b9ee45', 'd6bb78916', '70feb1494', 'adb64ff71', '62e59a501',
       '15ace8c9f', '5c6487af1', 'f190486d6', 'f74e8f13d', 'c5a231d81',
       'e176a204a', '1702b5bf0', '190db8488', 'c47340d97', '23310aa6f'],
      dtype=object)
```

Figure 9: Histogram of Number of Common Values Between Features and the Target

Beyond validating the identity of Figure 7's features, I also wanted more transparency into the process with which the Kaggle community derived their ordering. To this end, I replicated the efforts of a public Kaggle kernel that extended (both row and column-wise) a perfect diagonally-aligned time-series block. Not only did this deepen my understanding of the time-series nature of the Santander data, replicating the Kaggle community's work gave me a framework for growing any future time-series blocks that I might obtain (through my own independent efforts).

After having conducted these series of analysis, I could proceed further into Stage 2 whilst heavily relying upon the feature set specified in Figure 7.

4.2 Substage: Time Series Reconstruction

Notebook Location: `./scripts/time_series/time_construct.ipynb`

Though the features in Figure 7 are capable of creating a perfectly matched time-series (like the one shown in Figure 6), this perfect reconstruction is only valid for the row indexes that are also listed in Figure 7. If we extend our view to the entire Santander training and test sets, we would see that the perfect, diagonally-aligned time-series block breaks down. That being said, it does not negate the fact that Figure 7's feature set is useful. Rather, it implies Figure 7's feature set is only *complete* for the specified set of row indexes in Figure 7. For the remainder of the training samples, Figure 7's features are just as important (based on the intuition that they share the most values with the target variable) but don't depict complete time sequences.

So while Figure 7's 40 features still have predictive value, selecting a suitable value for the target variable won't be as easy as simply selecting and shifting (by 2 time intervals) the first feature column (see Figure 6). Instead, we must take into consideration that the 40 features may or may not constitute a complete time-series. This is accomplished using the following algorithm:

1. Assume that the 2-step-offset time interval lag seen in the Figure 6 holds true regardless of whether or not the 40 features represent a complete time-series
2. Check whether or not any one data sample belongs in a sequence:
    - Take one sample and cut off the first X values of the sample (default X is 2 from the 2-step-offset mentioned above) and make the remaining sequence into a tuple
    - Take all samples (including the one that just had its first X values lopped off) and now cut off the last X values for all the samples and make all the sequences into tuples (there should be as many tuples as there are samples)
    - If any tuples are repeated in the set of tuples, remove that tuple and all its duplicates entirely

- Match the one tuple to the duplicate-filtered set of tuples and if any tuple in the duplicate-filtered set matches the one tuple, then we can infer the one tuple's target variable by using the 2-step-offset rule to find the target value prediction within the one tuple's discarded front X values

3. Step 2 is repeated for all samples in a dataset and can be cascaded for a range of X values

    - By cascading a range of X values, you compensate for the fact that Figure 7's features often do not constitute complete time-sequences
    - If one X value yields a zero-value target prediction, perhaps a successive X value will succeed in finding a non-zero target value prediction
    - As soon as a X value produces a zero-value target prediction, we assume that to be the correct prediction and ignore results generated by successive X values

Using this outlined algorithm, it is possible to obtain target variable predictions for a portion of both the train and the test datasets.

These results can be easily formatted into a complete set of target variable predictions by setting all zero-value target variables (i.e. samples that do not conform to the time-series trend shown in Figure 6 and thus couldn't benefit from the outlined pair-matching algorithm) to the non-zero row-wise mean. Using this time-series reconstruction strategy to mine target variable predictions, I was able to easily outperform all my previous models.

4.3 Model 2v0: LightGBM with Data Leak

Notebook Location: `./scripts/Model_2v0_TS_LGB.ipynb`

As mentioned in Stage 2's time-series reconstruction substage (Section 4.2), excellent results can be gained simply by putting effort into reconstructing the time-series features within the Santander datasets to mine target values. In this sense, no machine learning is required to obtain top results in this competition. In actuality however, the fact that Santander anonymized every feature and also scrambled the time-series makes the task of time-series reconstruction a herculean challenge.

Even with the features that the Kaggle community found (see Figure 7), a majority of the target variables are forced to default to taking the row-wise mean of their respective samples. What this implies is that the features in Figure 7 do not constitute the only set that assumes the perfect diagonally-aligned order shown in Figure 6. To advance beyond the Kaggle community features, one would need to identify more sets and create entirely new time-sequences — something that is very difficult to accomplish.

As such, Model 2v0 attempts to bolster the time-series reconstruction's predictive power through the usage of LightGBM. Model 2v0's algorithm is detailed below:

1. Load the time-series reconstruction's target predictions for training and test sets (without filling in zero values with the row-wise non-zero mean)
2. Find the predictive value of each feature by training a XGBoost regressor on individual sets consisting of a single feature and the time-series reconstruction's training set target value predictions (leak)

    - Each XGBoost regressor trained on a feature-leak set is used to make predictions on the training target
    - The predictive value of any given feature can be quantified by the RMSE value generated from that feature's respective XGBoost regressor's prediction

3. Take the features with the best predictive values (i.e. lowest RMSE scores) and form a new training set and test set consisting of only these features
4. Add metadata features to these new training and test datasets by computing row-wise:
    - Log of mean
    - Mean of log
    - Log of median
    - Number of zero values
    - Sum
    - Standard deviation
    - Kurtosis
5. Add the time-series target predictions to the train and test sets (which essentially leverages the excellent predictive value of the time-series reconstructions from Section 4.2)
6. Train the LightGBM model on the preprocessed and feature-engineered data and make predictions on the test set
7. Merge the LightGBM predictions with the time-series reconstruction's test set target value predictions
    - Find the locations within the time-series reconstruction's test predictions that have zero-values
    - Replace these zero-values with the corresponding LightGBM predictions

The usage of LightGBM in conjunction with the time-series reconstruction prediction results was able to substantially improve upon the scores yielded from the time-series reconstruction substage.

4.4 Model 2v1: Modified LightGBM with Data Leak and Ensembling

Notebook Location: `./scripts/Model_2v1_TS_LGB.ipynb`

Like Model 2v0, Model 2v1 also applies machine learning in an effort to bolster the predictive value of the time-series reconstructions' target value predictions. However, it applies a slightly different preprocessing and feature engineering strategy along with the addition of a final ensembling step which performs a weighted combination of 4 different models: Model 2v0, Model 2v1, and two top-scoring results from the public Kaggle community. Model 2v1's algorithm is detailed as follows:

1. Perform feature scoring by filtering for features that satisfy the following criteria:
    - Shares at least 30 common values with the target variables
    - At least 3500 values in the feature are within a 5% range of their respective target variable values (which implies that the feature values are part of a sequence that includes the target variable)
2. Reformat the train and test sets to only include the important features that meet the criteria detailed above
3. Add Model 2v0's submission results as target feature in the test set
4. Calculate row-wise metadata features that include:
    - Non-zero mean
    - Non-zero maximum

- Non-zero minimum
- Number of zero values
- Mean
- Maximum
- Minimum

5. Add row-wise spatial location metadata (i.e. whether or not a row's index is cleanly divisible by 5) for values from 2 to 100 in an attempt to capture any potential structure that may exist in the row arrangements of samples
6. Create a new training dataset that consists of the concatenation of the old training and test set
7. Train a LightGBM model using this new training set and make predictions on the test set (note that 2v0 results are used as the ground truth for the portion of the new training set that was taken from the original test set)
8. Perform a weighted ensembling of various models' results

4.5 Stage 2 Results

Test Mean Notebook Location: `./scripts/time_series/test_mean.ipynb`

Table 4 shows the public leaderboard scores obtained from the time-series reconstruction substage, Model 2v0, and Model 2v1. As you can see, simply mining for time-series predictions was able to yield a two-fold increase in the public leaderboard score as compared to the best-performing model in both Stages 0 and 1.

In addition to the various experimental substages and models built in Stage 2, I decided to end this section with a quick experiment that aimed to yield insights on the test set. Using Section 4.2's time-series reconstruction algorithm, I wasn't able to obtain target value predictions for a majority of test set rows (the unknown majority's target values were simply assigned the row-wise mean value). Yet despite this shortcoming, the time-series reconstruction was able to outperform anything previously constructed in Stages 0 and 1. As such, I wanted to determine the predictive capacity using only row-wise means across the entire test set. The result (shown in Table 4) was the worst score I've obtained thus far in this challenge. This result demonstrates that though incomplete for a majority of entries within the test set, the time-series reconstruction's test set target value predictions carry substantial predictive value.

Model	**Public Leaderboard Score**
Time-Series Reconstruction	0.69
2v0	0.66
2v1	0.63
Test Mean	1.77

Table 4: Stage 2 Model Performances

5 Stage 2 Extension: The Breakthrough and Final Push

5.1 Breakthrough Intuitution

Source Code Locations:

- **Exploratory Notebook:** `./scripts/time_series/variety_analysis.ipynb`
- **Kaggle Probing Notebook:** `./scripts/time_series/set_viewer.ipynb`
- **Time-Series Reconstruction Notebook:** `./scripts/time_series/time_construct.ipynb`

The previous section, **Stage 2: Time-Series Unraveling**, worked exclusively with the features and feature ordering identified in Figure 7. A foundational premise of this section was that Figure 7's features (which, in Figure 6, demonstrate a curious behavior of the target variable having a 2-step-offset from the first feature) could be used to identify data samples that belong in a sequence. Simply having the knowledge that one data sample belongs in a sequence could enable me to make a target variable prediction for that sample by leveraging the 2-step-offset behavior.

As described in Section 4.2, the entire concept of target variable mining is focused on being able to obtain matching pairs of rows. These pair matches are made from evaluating whether or not 2 distinct rows are identical after one row has its front X elements removed and the other has its tail X elements removed. Since this matching algorithm is applied on a reduced-dataset that only utilizes the features in Figure 7, a successful pair match is a strong indicator that the 2-step-offset behavior can be leveraged to make a target prediction.

That being said, a "strong indicator" doesn't mean that matches can always be trusted. The X variable (which determines the depth of the front and tail cuts) starts from a value of 2 (to accommodate the 2-step-offset behavior) and increments up to the full number of features minus one. With the 40 features shown in Figure 7, this means that pair matching is conducted on rows ranging from 38 values down to 1 value. Of course, pair matches made on rows with longer lengths can be trusted much more than pair matches made on just 1 value. This simple observation is the main shortcoming of the pair-matching algorithm used in the Time Series Reconstruction substage (Section 4.2).

In short, the pair matching algorithm in Section 4.2 can't be trusted to be absolutely accurate. That is, even considering the significant performance gains that resulted from Section 4.2's Time Series Reconstruction, it must be assumed that a non-negligible number of the sequence matches were incorrect — thereby leading to performance losses.

This section begins by detailing a breakthrough discovery made by my project partner, Aaron Wu, that enabled me to substantially improve Section 4.2's pair-matching algorithm to achieve an exceptionally robust solution for target variable mining. Then, I will briefly introduce an algorithm I created in parallel with Aaron's efforts that attempted to uncover more time-series structure within the training set. Finally, I will introduce the various models and ensembling attempts that were constructed atop the newly mined target variables.

Before we begin, however, I must mention that a fair amount of credit for this breakthrough application must be given to the Kaggle community. In the *Kaggle Probing Notebook* referenced above, I verify a key assumption made by the community that enabled me to make a critical modification to the *Time-Series Reconstruction Notebook* which resulted in substantial performance gains.

5.2 Substage: Aaron's Breakthrough Time-Series Block Mining

Unfortunately, I don't have access to Aaron's mining algorithm so the best I can provide is a summary of how the process was accomplished. In this section, I'll outline Aaron's algorithm itself before explaining its incredible utility in an application inspired by the Kaggle community.

Aaron's breakthrough time-series mining relied heavily upon the features and feature orderings shown in Figures 6 and 7. Looking at Figure 6, it is immediately obvious that the sequence of rows and features clearly represent a time-series pattern. Using this established time-series pattern, it is possible to uncover further time-series patterns by:

1. Assuming that the row ordering which produced the time-series pattern shown in Figure 6 is correct and try to find additional sets and orderings of columns that create ordered time-series blocks
2. Assuming that the column ordering which produced the time-series pattern shown in Figure 6 is correct and try to find additional sets and orderings of rows that create ordered time-series blocks
3. Repeating the above 2 steps with each newly-generated time-series block

For both of the assumptions in item 1 and 2 above, new time-series blocks were grown from single columns (in the case of item 1) and single rows (in the case of item 2). These rows and columns weren't chosen at random. Rather, the rows or columns that had the highest non-sparsity percentages were given priority over all others in the mining process.

The growing process for the assumptions made in item 1 and in item 2 are very similar. In the hypothetical scenario where the assumption in item 1 is made (that is, Figure 6 and 7's row ordering is correct), a new time-series block might be generated as so:

1. Order the remaining columns (those not shown in Figure 7) in order of decreasing sparsity
    - Note that the degree of sparsity is evaluated over the rows given in Figure 7
    - By this criteria, many columns can be expected to be non-sparse at the beginning of the time-series block generation process
2. Of the ordered columns, select the one that has the least sparsity
3. Looking to the remainder of the ordered columns, select the next column that is at a one-step-offset from the first column
    - Note that columns can be grown towards the left or the right
4. Continue this growing process until columns that are at a one-step-offset can no longer be found

Once a new time-series block is generated, its features will be included in a **Do Not Consider** list along with the features already shown in Figure 7. In this manner, the search-space for Aaron's time-series mining algorithm is reduced with each newly-discovered block. Another interesting point to note is that while the algorithm's search-space is reduced with each new block, the number of low-sparsity candidate columns is also reduced. As a result, each successive block will have greater amounts of sparsity and therefore less predictive utility.

In total, Aaron's algorithm unearthed approximately **70 new time-series blocks** exhibiting the same diagonally-aligned behavior shown in Figure 6. Unlike the features and rows in Figure 6 however, it is important to note that Aaron's algorithm doesn't exhibit the two-step-offset behavior. As such, they cannot be used to directly make target variable predictions.

That being said, this doesn't mean that the newly-mined time-series blocks have no utility. On the contrary, these new sets are the keys that enabled me to finally overcome Section 4.2's pair matching algorithm's accuracy issue outlined in Section 5.1. The fatal shortcoming of this algorithm was the X term (which dictated the length of sequences considered when matching 2 rows) causing matches to be made on successively shorter sequences. This had the undesired effect of iteratively reducing the confidence with which each match is made.

5.2.1 Breakthrough Time-Series Reconstruction Algorithm

Armed with the 70 new time-series blocks from Aaron's mining efforts, I was able to circumvent the matching inaccuracy issue caused by increasing X values. The solution involved integrating these new time-series blocks into Section 4.2's pair matching algorithm. (Before reading further, please glance over the pair-matching's algorithm detailed in Section 4.2.) Aaron's time blocks were utilized in both Steps 1 and 2 of Section 4.2's algorithm. The changes are reflected in the below process-flow:

1. We will still assume that the 2-step-offset time interval lag shown in Figure 6 can be trusted as a fundamental behavior of the time-series data
    - Since Aaron's time-series blocks do not have predictive power by themselves (i.e. they don't share an immediate relationship with the target variable), this assumption is absolutely critical as it implies that Figure 6 and 7's features are still the only ones that can be trusted to contain viable candidates for making target value predictions
2. We will still carry out pair matching by checking whether or not any one data sample belongs in a sequence:
    - But now the front-end and tail-end value snipping is performed on data samples that retain the Figure 6 and 7 features **along with** features from a pre-specified number of Aaron's time-series blocks.
    - Due to the several-fold increase in the number of features, pair-matching becomes significantly more reliable
    - Previously, X would increment up until only single-value sequences are being matched together
    - Now, the inclusion of multiple time-series blocks means that even the single-value sequences are being concatenated together to form a sequence that is as long as 1 (the original set in Figures 6 and 7) plus however many of Aaron's blocks are being used
3. Besides the above changes, Section 4.2's algorithm remained the same

The inclusion of Aaron's 70 newly-mined time-series blocks in Section 4.2's pair-matching algorithm resulted in a substantial improvement upon any other results obtained thus far. (As in Section 4.2, targets that couldn't be mined were simply defaulted to the row-wise mean value.) As mentioned in the modified pair-matching algorithm above, an important hyper-parameter is the number of blocks to utilize for supplementing the vanilla Figure 7 features. Table 5 details the public leaderboard scores that were obtained from using varying numbers of mined blocks.

Number of Blocks	**Public Leaderboard Score**
10	0.59
55	0.55

Table 5: Target Value Predictions with Aaron's Mined Blocks

As Table 5 shows, it appears that using more blocks had the effect of improving performance. With each additional block integrated into the pair-matching algorithm, the criteria for what constitutes a valid match becomes stricter. Therefore, integrating more blocks increases the accuracy with which matches are made. Of course, this effect is valid only so long as the features that are contained in these blocks actually constitute a legitimate time-series. As mentioned in the above description of Aaron's block mining algorithm, each newly-generated time-series block had iteratively greater sparsity. What this means is that as you add more and more blocks to Section 4.2's matching algorithm, you'll eventually be doing more harm than good since Aaron's later time blocks cannot be trusted. After some experimentation on the training set, it was determined that 55 blocks yielded the optimal score on the public leaderboard.

5.3 Substage: Alternate (My Own) Time-Series Block Mining

Source Code Locations:

- **Main Notebook:** `./scripts/time_series/feature_mining.ipynb`
- **Python Code:** `./scripts/time_series/feature_mining.py`

Concurrently with Aaron's block mining algorithm, I worked on a separate algorithm that approached block mining using an altogether different approach than Aaron's. While Aaron's algorithm grew more blocks from the features and rows shown in Figures 6 and 7, I decided to perform my mining operation from-scratch. This idea was inspired by an analysis I conducted into the column-wise sparsity of the test dataset. Figure 10 depicts a histogram describing the distribution of sparse features within the test dataset. Interestingly, this histogram looks very similar to Figure 8 which shows a column-wise correlational behavior between feature values and the target variable in the training set.

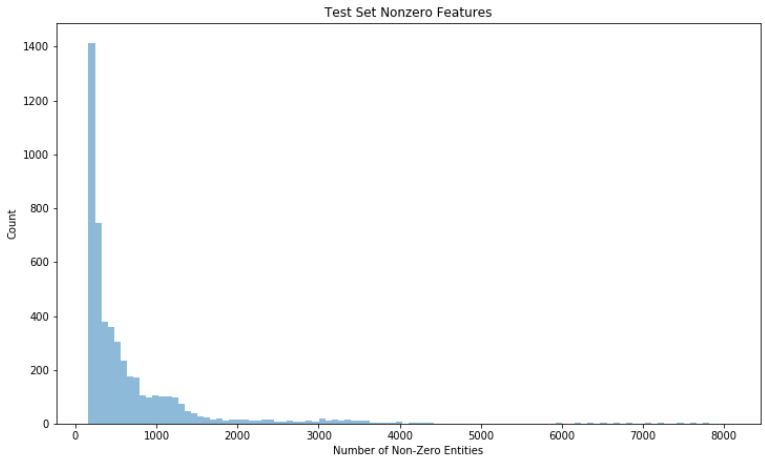


Figure 10: Histogram of Number of Nonzero Values in Test Features

The similarity between Figures 8 and 10 prompted me to examine the row-wise sparsity behavior of the test data as well. This investigation resulted in the histogram distribution shown in Figure 11.

Once again, a long-tail distribution similar to Figures 8 and 10 can be seen. The consistency between these three figures led me to hypothesize that there exists a strong relationship between high percentages of non-sparsity (either row or column-wise) and the capacity to organize into a time-series block. *(Of course,*

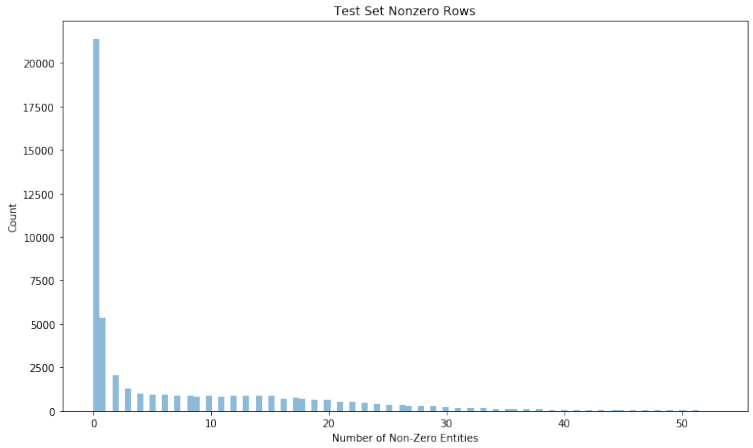


Figure 11: Histogram of Number of Nonzero Values in Test Features

given the fact that Figure 8 reflects training-set behavior while Figures 10 and 11 are specific to the test set, one of my primary assumptions is that the time-series behaviors of the training set are paralleled in the test set as well. Though it might seem like it, this assumption isn't actually overly-optimistic since Section 4.2's time-series reconstruction yielded excellent results for both training and test sets.) Using these newfound discoveries, I created a block-mining algorithm that synthesizes the intuitions of row-wise sparsity with time-series behavior. Unfortunately, my algorithm is heavily dependent upon knowledge of the target value and thus will only work on the training set.

A high-level process flow for my algorithm is as follows:

1. Separate the training data into batches:
   - Define the number of elements to be put into each batch
   - Define a minimum row-wise sparsity requirement (i.e. do not consider rows that are predominantly zeros)
   - Sort the training set rows in terms of increasing row sparsity
   - Divide the sorted training set rows into individual batches
2. For each batch obtained above:
   (a) Initialize 10 seed rows from which to grow a potential time-series block:
      - Looking at the target value of each row, give each row a ranking that quantifies how many times that row's target value shows up within a batch
      - Sort the rows within the batch by decreasing rank
      - Select the top 10 rows as seed rows
      - *This algorithm makes the assumption that rows with target values commonly-occurring within a batch are likely to belong in a time sequence*

   (b) Grow a candidate time-series block for each of the 10 seeds rows initialized above:
      i. Store the row index of the seed row
      ii. Create a subset dataset by filtering for rows in the current batch containing the seed row's target value
      iii. Assign probabilities to each row within the subset dataset that quantify the likelihood of any one row to be the next time-series entry (calculated using intersection divided by union between the seed row and the subset dataset's rows)

iv. Select the row from the subset dataset with the highest probability
v. For this newly-selected row, store the feature name containing the target value
vi. Assign this newly-selected row as the next seed row
vii. Repeat steps (i) through (vi) until there are no rows left to form subset datasets

(c) Save the row and column indexes for candidate time-series blocks for each batch (there should be 10 candidate blocks per batch)

The algorithm I outlined above was able to produce hundreds of candidate time-series blocks which required manual review. Obviously, this was an enormous amount of candidate blocks to review but fortunately, Aaron's block-mining effort was concluded before I had the chance to perform the manual labor. That being said, this algorithm was successful in mining time-series blocks and could've been used as a last-resort had Aaron not succeeded in his endeavor.

5.4 A New Baseline, a Standard Metadata Set, and Feature Scoring

Source Code Locations:

- **Train Feature Scoring Code:** `./scripts/feature_scorer.py`
- **Test Feature Scoring Code:** `./scripts/test_feature_scorer.py`

Taking a look at Table 4, it can be seen that Models 2v0 and 2v1 both improved upon the score posted by Section 4.2's time-series reconstruction. The reason behind these improvements is because both models utilized the target value predictions generated through the reconstruction process (Section 4.2). By including these predictions as new features, Models 2v0 and 2v1 were able to build upon and improve the baseline 0.69 score.

Following Aaron's block-mining efforts and a newly modified time-series reconstruction algorithm, a new baseline is established with a public leaderboard score of 0.55 (see Table 5). All models built in the latter half of Stage 2 builds and improves upon these new, more accurate target value predictions to obtain medal-qualifying performance. Like in the former half of Stage 2, the new models will incorporate reconstructed target variable predictions as metadata features (though the exact method that this is accomplished differs from model to model).

Speaking of metadata, it should also be noted that the models created in this section all incorporate row-wise statistical calculations (similar to models created in previous stages). However, for the sake of consistency, all models that were built in the latter half of Stage 2 utilize the same row-wise calculations (calculated across the full training and test datasets). These calculations include:

- Applied to Non-Zero Row Values:
  - Mean
  - Log-Mean-Exponential
  - Median
  - Standard Deviation
  - Kurtosis
  - Minimum
- Applied to All Row Values:

- Sum
- Maximum
- Number of Zeros
- Mean
- Log-Mean-Exponential
- Median
- Standard Deviation
- Kurtosis

In addition to the standardized set of metadata, many of the models used in the latter half of Stage 2 utilize features that have been vetted through a feature-scoring process identical to that used in Model 2v0 (see Section 4.3). The intuition behind this strategy is rooted in the curse of dimensionality. By pre-evaluating the predictive capacity of each individual feature, I am hoping to pass only the most-relevant features into my boosting algorithms. In this sense, the feature-scoring process is akin to a dimensionality-reduction algorithm. Also, based on different configurations used during the breakthrough time-series reconstruction algorithm (specifically, the X value), different sets of features will be generated.

I created two versions of the feature-scoring algorithm: one that is based on Model 2v0's and another that capitalizes on the perfect accuracy of the breakthrough time-series reconstruction algorithm. In this new algorithm (see the *Test Feature Scoring Code* referenced above), feature-scoring is performed on the test set instead of the train set. Of course, such an operation is only possible given that we have target variables. Fortunately, target value predictions made using the breakthrough reconstruction formula can be trusted with a near-absolute confidence, even on the test set. As such, it becomes possible to leverage out-of-sample structure in data preprocessing.

By using inter-changable sets of feature and metadata units between models, I am able to effectively evaluate and pinpoint deficiencies from one model to the next. Moreover, this high degree of modularity allows me to easily swap out various preprocessing strategies (often times, within the same model) and create a multitude of distinct results for robust ensembling.

5.5 Model 2v2: LightGBM with Robust Time-Series Reconstruction

Notebook Location: `./scripts/Model_2v2_TSTest_LGB.ipynb`

Model 2v2's purpose is to establish a new standard for model-based performance following the discovery of the breakthrough time-series reconstruction (Section 5.2.1). Model 2v2's process-flow is outlined as follows:

1. Load the training and test set reconstruction-based predictions (henceforth referred to as *leaks*)
2. Reformat the training and test sets by:
   (a) Adding the train and test leaks as new metadata features in the train and test sets
   (b) Loading the feature-scoring results for the desired time-series reconstruction configuration and integrating the highest-scoring features into the training and test sets
   (c) Loading and integrating the pre-calculated row-wise metadata into the train and test sets
   (d) Reduce the training and test sets to only include the newly-added features indicated in items (a) through (c)
   (e) *Note that item (b)'s action can easily be modified to include the entire feature set rather than features that have been pre-scored*

3. Scale the training and test sets to zero mean and unit variance with respect to the training set features
4. Tune a LightGBM regressor while monitoring the training and validation root mean squared error (RMSE)
5. Make predictions on the test set using the trained LightGBM regressor

5.6 Model 2v3: CatBoost with Robust Time-Series Reconstruction

Notebook Location: `./scripts/Model_2v3_TSTest_CBoost.ipynb`

Model 2v3 has all the same functionalities as Model 2v2 with the exception of the choice of boosting model. Obviously, Model 2v2 used LightGBM while Model 2v3 uses CatBoost. The usage of a different boosting algorithm required me to perform an entirely new hyper-parameter tuning procedure. Besides this small difference, Model 2v3 shares the extreme modularity of Model 2v2 that allows for rapid prototyping of various preprocessing schemes.

5.7 Model 2v4: LightGBM with Test-Only Data

Notebook Location: `./scripts/Model_2v4_TSTest_LGB.ipynb`

Model 2v4 was created so that a hypothesis could be tested and therefore its process flow deviates from 2v2 and 2v3. Recall Section 5.1's reference to the *Exploratory Notebook*. Figure 12 is taken from this notebook and demonstrates a curious behavior between the value overlap of two subsets of the test set: rows in which the breakthrough time-series reconstruction was able to successfully find target value predictions (designated as *test matched*) and rows in which that wasn't the case (*all test unmatched*).

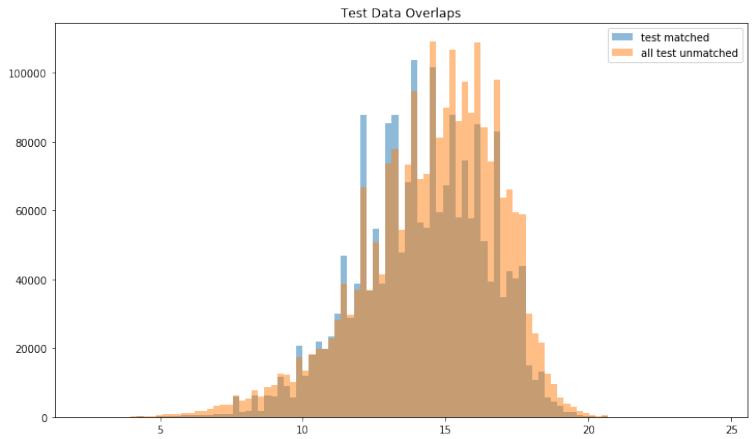


Figure 12: Histogram of Test Data Overlaps

Looking at Figure 12, it can be seen that the *all test unmatched* distribution appears to be slightly rightward-leaning from the *test matched* distribution. However, this behavior doesn't seem to be shared by the training set. Figure 13 depicts an equivalent plot for the training data.

Comparing Figure 12 to Figure 13, we can see that the data overlap between matched and unmatched training samples is much more consistent than that of the test set. What this observation might imply is that an unknown structure exists in the test set that can potentially aid in determining the identities of

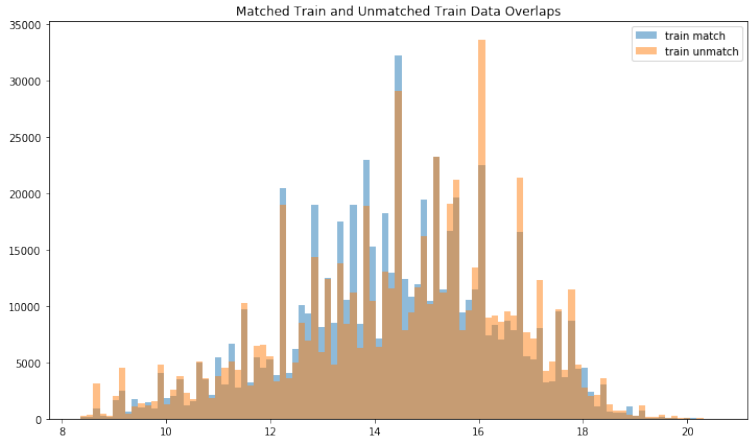


Figure 13: Histogram of Train Data Overlaps

the unmatched test samples (i.e. those that weren't able to benefit from the breakthrough reconstruction algorithm). Figure 14 lends additional credibility to this possibility.

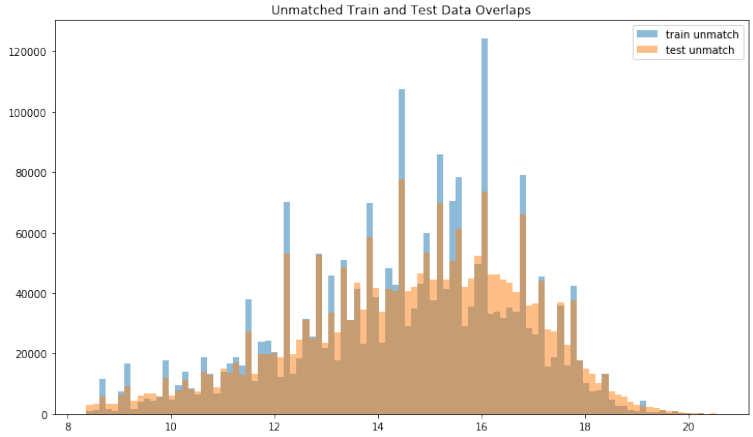


Figure 14: Histogram of Unmatched Train and Test Overlaps

Figure 14 overlays the distribution of unmatched training samples to unmatched test samples. Remember that the term *unmatched* refers to samples that couldn't benefit from the breakthrough time-series reconstruction's pair-matching process. These were rows in which I was forced to default to the row-wise mean value when making a submission on Kaggle. From Figure 14's overlaid histograms, we can see that the two distributions actually line up quite nicely despite the obvious differences between Figure 12 and Figure 13. Below, I lay out the facts of the situation:

- Test matched and test unmatched have **different** distributions
- Train matched and train unmatched have **similar** distributions
- Train unmatched and test unmatched have **similar** distributions

If the training set (which was used to train boosting algorithms in Model 2v2 and 2v3) have consistent data distributions between their matched and unmatched subsets, then how could it be expected to yield good performance on the test set where the matched and unmatched subsets are not well-aligned? It was using this line of logic that I created Model 2v4 with the hope that some structure might have existed in the

matched test samples that could be well-suited for resolving the disparity between matched and unmatched samples shown in Figure 12. Of course, just from looking at the 3 figures (12, 13, and 14), there isn't an obvious reason why I should've expected this to have helped — hence the need to test this hypothesis.

The process flow for Model 2v4 is as follows:

1. Load the test set
2. Format the test set by adding any combination of metadata, feature-scored features (see Section 5.4's *Test Feature Scoring Code*), leak information, or other preprocessed features
3. Scale the test set to zero mean and unit variance
4. Split the test set into two subsets:
    - Matched samples (used as a training set)
    - Unmatched samples (used as a test set)
5. Tune the hyper-parameters for a LightGBM regressor algorithm
6. Make predictions on the unmatched test samples
7. Aggregate the matched and unmatched test samples into a format suitable for Kaggle submission

As a final note, it is important to realize that the process-flow just outlined above is only possible due to the extremely high confidence that we can have in the matched test samples' predicted target values. Obviously, none of the matched test samples' target values were provided in this challenge's original problem specification. This observation speaks to the breakthrough time-series reconstruction algorithm's extraordinary utility.

5.8 Model 2v5: Match-Focused LightGBM

Notebook Location: `./scripts/Model_2v5_TSTest_Matched.ipynb`

From my (admittedly, limited) experience thus far in machine learning, I know that having more training data is always a good thing. Fortunately, with the breakthrough time-series reconstruction algorithm, I had the option of padding my training data with the matched test samples (those that benefited from the pair-matching approach for target prediction) which would more than doubled the number of training samples. Combined with the extreme modularity of the preprocessing pipeline shared by all latter Stage 2 models, the option of having an augmented training set could potentially have resulted in a very powerful model.

This was the motivation behind creating Model 2v5. The *Matched-Focused* part of Model 2v5's description refers to its usage of a training set comprised entirely of matched samples from both the training and test sets. In this way, new training and test sets were formed: the training set comprised entirely of matched samples and the test set containing unmatched samples.

The process flow for Model 2v5 is as follows:

1. Load both the training and the test sets
2. Restructure the training and test sets by:
    - Adding any combination of metadata, feature-scored features, leak information, or other preprocessed features

- Splitting the two sets into matched and unmatched subsets
- Recombining train and test matched and unmatched subsets into a new training set of matched samples (aggregate train) and a test set of unmatched samples (aggregate test)

3. Tune a LightGBM regressor model
4. Make predictions on the aggregate test set using the tuned LightGBM model trained on the aggregate train set
5. Reformat the aggregate train and test sets into a submission-ready format

5.9 Stage 2 Breakthrough Summary and Ensembling

Notebook Location: `./scripts/weighted_blender.ipynb`

In review, the latter half of Stage 2 saw breakthrough time-series reconstruction efforts and four unique models that leveraged the new discovery. Individually, some of these models produced excellent public leaderboard scores (at the time). Unfortunately however, due to daily submission limits and a fast-approaching competition deadline, I wasn't able to evaluate the public leaderboard scores for around half the models created in the latter half of Stage 2. Table 6 summarizes the chronological progression of submission results along with the effects of ensembling various models together.

Model	Children	Public Leaderboard Score
2v5 (Tuning 1)	N/A	0.54
2v2 (Tuning 1)	N/A	0.521
2v2 (Tuning 2)	N/A	0.522
2v3	N/A	No Score
2v4	N/A	No Score
2v5 (Tuning 2)	N/A	No Score
Ensemble 0	2v2 (T2), 2v3, 2v4, 2v5 (T1)	0.522
Ensemble 1	2v2 (T1), 2v3, 2v4, Ensemble 0	0.517
Ensemble 2	2v2 (T1), 2v3, 2v4, Ensemble 1	0.518

Table 6: Stage 0 Model Performances

As Table 6 shows, there were a total of 3 separate ensembles created during the latter half of Stage 2. (As opposed to the former half of Stage 2 however, all models incorporated in the ensembling procedure are completely original.) Once again, it could be seen that the ensembling process has the desirable effect of improving overall model performance. In the `weighted_blender.ipynb` file referenced at the beginning of this subsection, you can see that I utilized a weighted ensembling strategy that assigned different weights to different models based on their posted public leaderboard score. For models in which I didn't have the luxury of knowing their leaderboard scores, I made do with simple gut instinct (for better or worse).

My final standing in this challenge (243rd out of 4484) was the result of using Ensemble 2 for the purposes of final scoring. In hindsight, Ensemble 1 might have boosted me up a few places, but such is the nature of Kaggle challenges where a difference of a 0.001 has a non-negligible impact.

6 Miscellaneous

Throughout the course of this project, more than a couple of ideas didn't pan out for one reason or another (i.e. my own implementation of covariate shift correction and time-series block mining, for starters). The ones in which I put in a significant amount of work are detailed within the preceding pages. This section is dedicated for the low-effort ideas that were toyed with here-and-there. If anyone was ever interested in taking up this project, the following bite-sized concepts may be potential seeds from which to develop an investigation.

6.1 Lexicon-Based Post-processing

Source Code Locations:

- **Lexicon Generator Notebook:** `./scripts/lexicon_generator.ipynb`
- **Lexicon Implementation Example Notebook:** `./scripts/Model_2v3_TSTest_CBoost.ipynb`

Lexicon-Based Preprocessing takes inspiration for the time-series nature of the Santander dataset. The arrangement of rows and columns in the Santander dataset is neatly summarized by a time-series block sample shown in Figure 6. Looking at Figure 6, it is immediately obvious that there is a multitude of repeating values amongst the rows and columns. These repeating values can easily be formatted into a lexicon (or dictionary) of values.

The main utility behind this concept is that it represents a way of correcting regressor outputs. Due to the time-series reconstruction algorithm's (see Sections 4.2 and 5.2.1) robust target variable predictions, machine learning models only need to make forecasts on unmatched data samples. In such cases, it might be beneficial for an additional post-processing step to round the regressor output to the nearest lexicon entry. Depending on the quality of the regressor model, defaulting to lexicon values may result in slight performance gains.

Model 2v3 contains a toy implementation of this concept that can be found by scrolling to the bottom of the notebook.

7 Conclusion

The Santander dataset has been the most difficult tabular dataset that I've ever worked with to-date. The combination of sparsity, lack of low-dimensional structure, and a scrambled time-series has confounded me at every step of the project and proved to be an excellent avenue for exploring a wide breadth of machine learning topics from stacked auto-encoders and covariate shift correction to custom boosting implementations. With a combination of raw hard work, tenacity, a touch of ingenuity, and perhaps a bit of luck Aaron and I finished in the top 6% and earned our first bronze medal on Kaggle.

For anyone seeking to improve upon the results I've exhaustively detailed above, there are two potential paths to explore: further time-series exploration and more robust machine learning modeling. However of these two options, I remain convinced that there are more gains to be had in pursuing the time-series path. Despite the intense effort that Aaron and I poured into unraveling the time-series nature of the dataset, it remained troubling that a majority of the test set was unable to benefit from our time-series reconstruction algorithm. Surely, there remains substantial nuggets of structure in the test set that contains tremendous

predictive value but this would be a topic for someone else to explore. My involvement with the Santander dataset ends here.

At the end of the day, this project has been a massive learning experience for me and I hope that by reading through this report, some of the value I've gained has been transferred to you as well. Please feel free to reach out to me if you have any questions or comments on my processes. Thank you for taking the time to look over my project!

8 Acknowledgements

I am sincerely grateful to the incredible Kaggle community. Many of the models I built throughout this project took inspiration from the awesome work of brilliant Kagglers. Please look to my Github repo (specifically the Jupyter Notebook files) to find links to the public kernels that I referenced.