Santander Value Prediction: Kaggle 2018

Cheng-Hao Tai

August 12, 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 a scrambled feature set. Through a series of 6 successive models and heavy ensembling, I am projected to finish within the top 10% of over 4,000 competitors.

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 a following section) that provided an excellent sandbox for honing my skills relating to analyzing, processing, and writing predictive models on 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

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 or 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

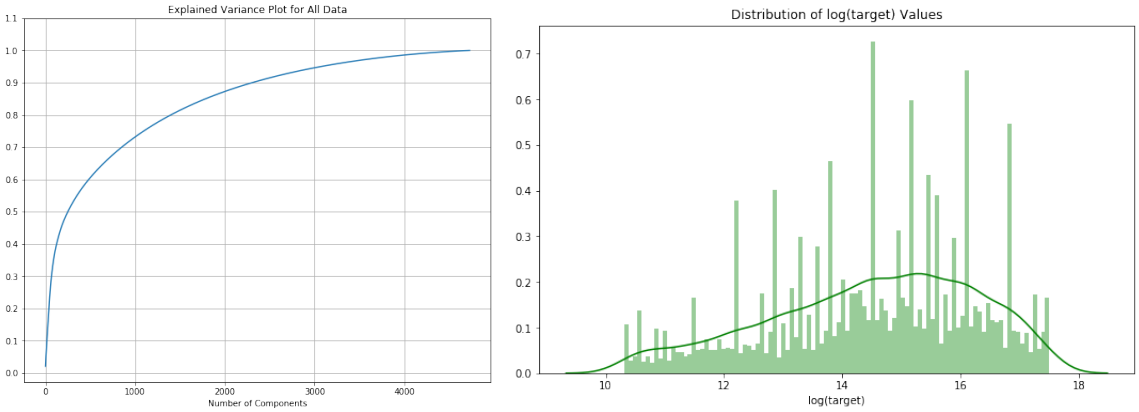
Stage 0 and its accompanying substages focused on establishing solid footing in this competition. To that end, all the models produced in this stage took inspiration from top-scoring public kernels so to efficiently identify the best strategies for dealing with the Santander datasets. All due acknowledgements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

From the visualizations generated in the vanilla preprocessing substage, the challenges of this tabular dataset begin to surface. 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 means 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 explained 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 the Explained Variance plot determined 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 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 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 Parameters for LightGBM Regressor
3. Using the tuned hyper-parameters, train and make predictions using separate LightGBM Regressor models over 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 value.

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 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 implied) 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: *Since 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.*

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 dataset into binned target values (classes) and class probabilities on the train dataset and transform both the train and test datasets
   (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 separate Random Forest Classifier to generate metadata features from a vanilla dataset. On a high level, the inclusion of item (3b) in the process flow creates a more robust model by chaining machine learning algorithms together.

On a more detailed 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 fitted random forest classifier to obtain new metadata features:
    - 1st Feature: Class predictions (integer values)
    - 2nd Feature (Set): 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). Referencing Table 2, we can see that the public leaderboard was 1.44 while - for varying learning rates -

```
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

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.

Following 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 is because of Sugiyama and Kawanabe's (two of the most recognizable names in the field of covariate shift correction) involvement along with the fact that the intricacies of this algorithm are very clearly detailed within the 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.

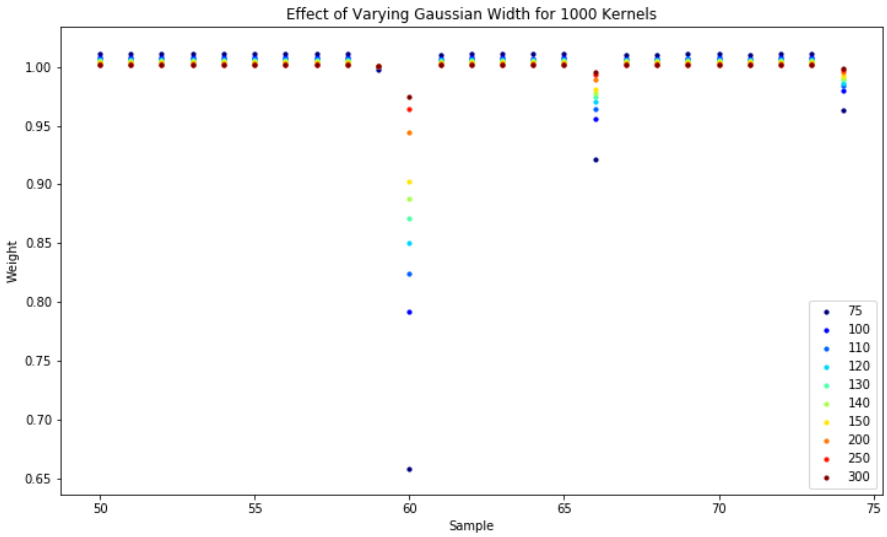


Figure 3: Effect of Varying Gaussian Width with 1000 Kernels

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 magnitude 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 efficacy 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 testing 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.

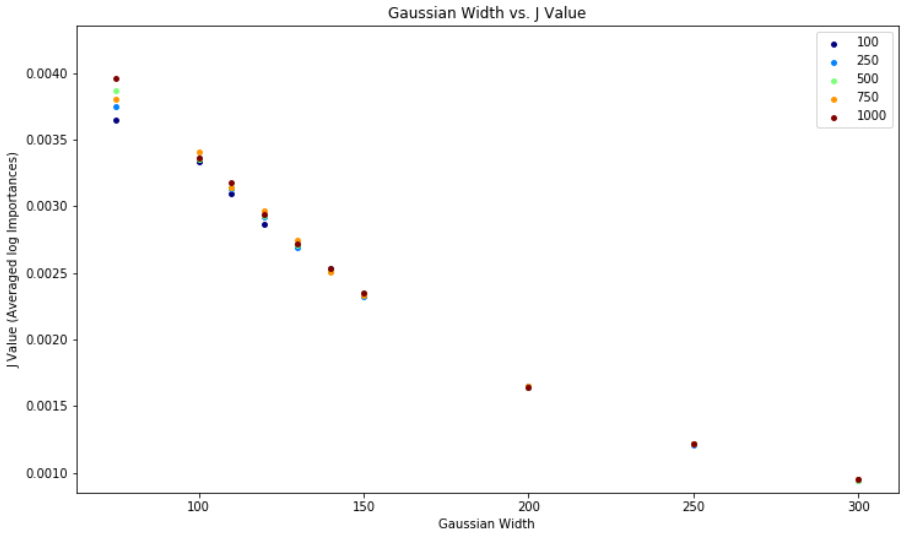


Figure 4: Effect of Gaussian Width on J Value

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:

- **Main Notebook:** `./scripts/Model_1v0_XGB_CS.ipynb`

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

- **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 using 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, 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 could be expected by retraining the KLIEP weights on 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 dimensions are largely independent of 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 of 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. 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 constructing machine learning models.

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 series 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

Notebook Location: `./scripts/time_series/similarity_with_target.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 the ones that share the most common values with the target variable. The result of this analysis is visualized in Figure 8.

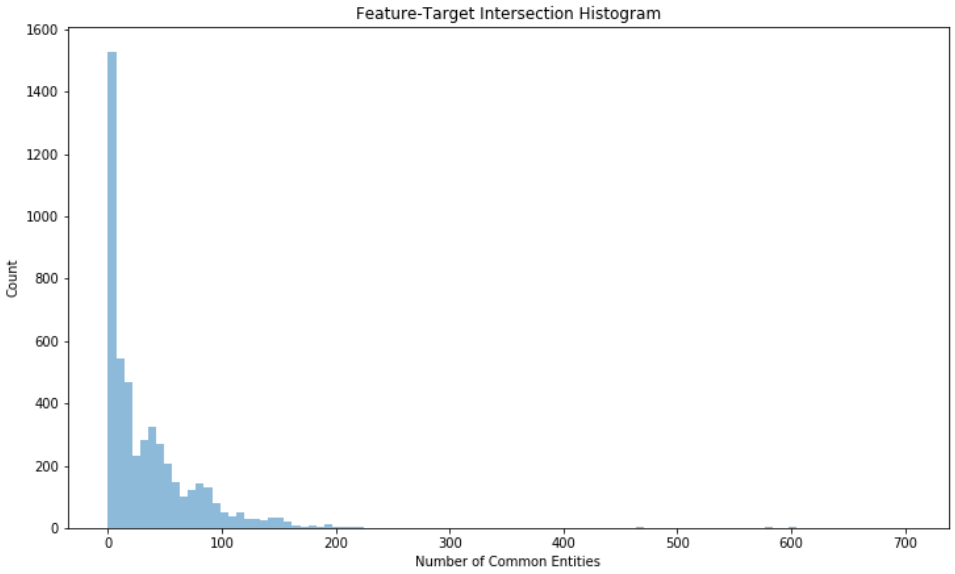


Figure 8: Histogram of Number of Common Values Between Features and the Target

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 find a list of features (see Figure 9) that are identical to those shown in Figure 7.

```
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

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 fine-tuned to perfectly recreate the time-series. Yet despite their difference in ordering, the fact that the lists of features in Figure 7 and Figure 9 perfectly match (despite them having been mined using differing strategies) provides a partial validation to the Kaggle community's analysis. This gives me peace-of-mind as I progress further into Stage 2 using 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 reconstruction 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 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 n values of the sample (default n is 2 from the 2-time-lag interval mentioned above) and make the remaining sequence into a tuple
    - Take all samples (including the one that just had its first n values lopped off) and now cut off the last n 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-time-shift rule to look at the one tuple's discarded front n values
3. Step 2 is repeated for all samples in a dataset and can be cascaded for a range of n values
    - By cascading a range of n values, you compensate for the fact that Figure 7's features often do not constitute complete time-sequences
    - If one n value yields a zero-value target prediction, perhaps a following n value will succeed in finding a non-zero prediction
    - As soon as a n value produces a zero-value target prediction, we assume that to be the correct prediction

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) to the non-zero mean across their respective samples. 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, excellent results can be gained simply through putting effort into reconstructing the time-series features within the Santander datasets to mine for target variable predictions. 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 task.

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 across-row mean of their respective samples. What this implies is that the features in Figure 7 do not constitute the only set that assumes the perfectly diagonal-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 training predictions
    - Each XGBoost regressor trained on a feature-prediction 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
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)
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 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 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. 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. Reconstruct the train and test sets to only include the important features that meet the criteria detailed above
3. Calculate row-wise metadata features that include:
    - Non-zero mean
    - Non-zero maximum
    - Non-zero minimum
    - Number of zero values
    - Mean
    - Maximum
    - Minimum
4. 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
5. Create a new training dataset that consists of the concatenation of the old training and test set
6. Train a LightGBM model using this new training set and make predictions on the test set
7. Perform a weighted ensembling of various models' results

4.5 Stage 2 Results

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.

Model	Public Leaderboard Score
Time-Series Reconstruction	0.69
2v0	0.66
2v1	0.63

Table 4: Stage 2 Model Performances

5 Conclusion

The Santander dataset has been the most difficult dataset I've ever worked with up to the point of this report's drafting. 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 to covariate shift correction. Barring any serious upsets in the final days of the competition, I expect to finish within the top 10% and (fingers crossed) maybe even earn my first 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 models. That being said, I personally feel that time-series exploration will yield more of a boost in predictive accuracy. While there's arguably always more to do in terms of hyper-parameter tuning and ensembling, I wouldn't expect these efforts to result in a score reduction down to the 0.48 range (which is where the top of the competition leaderboard is currently at).

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.

6 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.