Santander Bank Transaction Prediction

To evaluate different models for Binary classification

Ajeya Kempegowda
1st year MSDS
(Khoury college of Computer Sciences)
Northeastern University
kempegowda.a@husky.neu.edu

Deepanshu Parihar
1st year MSDS
(Khoury college of Computer Sciences)
Northeastern University
parihar.d@husky.neu.edu

Harish Ramani
1st year MSDS
(Khoury college of Computer Sciences)
Northeastern University
ramani.h@husky.neu.edu

I. INTRODUCTION

In this project, we are using data to help Santander bank identify whether customers will make a transaction in the future, irrespective of the amount of money transacted. The data provided has the same structure as the real data. We intend to build several traditional ML models that tackle the binary classification problem and also look at a new class of Neural Networks, and compare its efficiency with the rest of the models. We intend to use AUC (Area under the curve) as the metric to evaluate the models.

Our work chiefly involves in performing exploratory data analysis to observe relationships between the predictors and to develop machine learning models that can help us best predict future transactions with Santander Bank.

Further, we plan to explore the methodologies presented in Neural Ordinary Differential Equations, a research paper that won the best paper award in the renowned NeurIPS in 2018. This paper opens up a new avenue in the architecture of current neural networks, and we intend to compare the performance of these new class of neural networks to the aforementioned machine learning models.

II. METHODS

The transaction data set from Kaggle is available in '.csv format with close to 200k observations which can be readily loaded to Python environment. The data set consists of 200 features and are all numeric. We've trained all our machine learning models on the training data set and generated predictions on the test set provided by Kaggle. These predictions are saved in a '.csv' file and uploaded onto the Kaggle site to evaluate the accuracy of our predictions.

A. *Exploratory data analysis*

The initial EDA revealed that the data set is void of categorical variables and there are no missing values in the data. The histogram plots on the features were normally distributed, which implied that the data was pre-processed. Further, the count plot on the target column indicated a clear class imbalance issue (Figure 2.a.1), wherein close to 90% of our data has 0 label and the rest with 1. Finally, the correlation plot showed that none of the predictors were correlated as seen in Figure 2.a.2

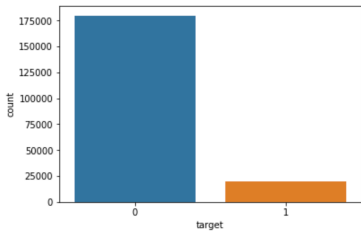

Figure 2.a.1

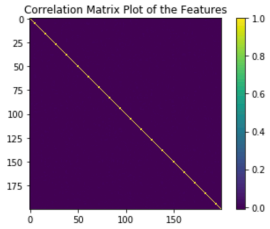

Figure 2.a.2

B. *Dealing with imbalanced data*

In order to achieve better classifier accuracy (in ROC space) it is essential to balance our data, either by down sampling the majority class or over sampling the minority class. We chose not to not Down sample the data as we would end up in dropping a major chunk of our observations. We chose the latter approach by employing Synthetic minority over-sampling technique (SMOTE) on training data. As seen in Fig 2.b.1

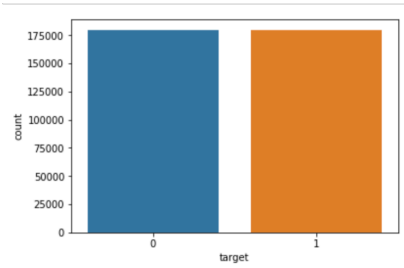

Figure 2.b.1

C. *Baseline models*

Evaluation is important in any machine learning process. A basis for comparison of results (a meaningful reference point to compare with). Once we obtain results from different machine learning algorithms, a baseline result can indicate the relative performance of these algorithms.

With this thought, we've chosen a simple Logistic regression as our baseline model as it's the simplest binary classifier.

Further, the initial EDA indicated that all the predictors are normally distributed and are also uncorrelated

between them. This leads to the inference that our predictors are independent. This is vital in our model development as we can employ Gaussian Naive Bayes classifier and expect to produce better results.

D. *Neural Networks*

Neural networks are well suited for classification problems, and they provide flexibility to existing models and in general tend to slightly increase the accuracy of the models. With this background we wanted to evaluate how Neural networks could improve our previous scores. We implemented a multi layer perceptron with 13 neuron each in the 3 hidden layers using Sci-kit learn library. Given, the complexity of the model, we employed ReLU activation to ensure faster convergence. Further, we trained our neural networks over 1000 epochs with a learning rate of 1e-4 to achieve better optimization.

E. *Ensemble models*

Ensemble modeling tries to achieve better accuracy by averaging out the predictions from multiple models than any of its constituent models alone. We performed ensemble modeling assembled with Naive Bayes algorithm as it explains our data better. We've incorporated a 5 fold stratified approach to cross validate the model. The Ensemble approach can be summarized as follows in Figure 2.e.1

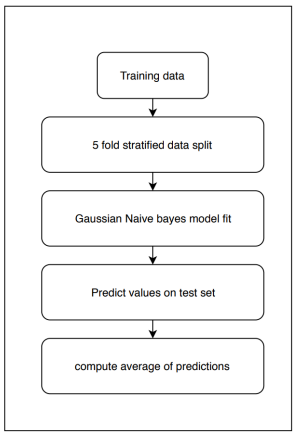

Figure 2.e.1

F. *Boosting models*

We wanted to try to improve our predictive score and through search we found two methods that are very popular in the data science community right now - XGBoost and LGBM. Both of these methods are gradient tree boosted methods. So talking about gradient boosting machine first.

Gradient Boosting Machines- GBM comes from a statistical framework called ARCing i.e. Adaptive Reweighing and Combining, where boosting is casted as a numerical optimization problem through minimizing the loss by adding weak learners using gradient descent. It later developed into gradient boosting machines as a machine learning technique for regression and classification problems. It produces a prediction model in the form of an ensemble of weak prediction models, typically, decision trees. Every iteration tries to reduce the errors of the previous iteration by sequentially fitting on the residuals. So, the intuition behind gradient boosting algorithm is to repetitively leverage the patterns in residuals and strengthen a model with weak predictions and make it better. Once we reach a stage that residuals do not have any pattern that could be modeled, we can stop modeling residuals, otherwise it might lead to over fitting. This makes the depth of the tree an important hyper parameter that we need to tune. The loss function that we minimize is binary log loss which is obtained using gradient descent.

Both the boosting methods we have used above are based on the GBM concept. The difference lies in the specifics of the optimizations. XGBoost follows level wise splitting whereas LGBM (Light Gradient Boosting Method) follows a leaf wise split.

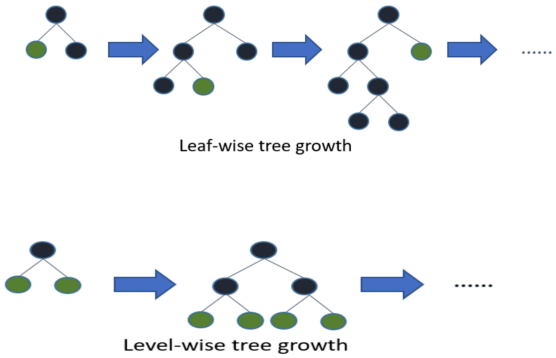

Figure 2.f.1

In contrast to level-wise growth, a tree grown with leaf-wise growth will be deeper when the number of leaves are the same. This means that the same 'max_depth' parameter can result in trees with vastly different levels of complexity depending on the growth strategy. This allows leaf wise growth to converge more easily but at the same time more prone to over-fitting and making 'max_depth' a very important hyper parameter.

XGBoost improves on GBM's by reducing the computations required. The most computationally expensive task is to decide on the choice of split at every node. Generally, this will have a time complexity of O(#data points * #features). But using newer methods like Histogram based splitting which take advantage of grouping features into a set of bins and perform splitting on the bins instead of the features, results in reducing complexity to O(#data points * #bins). Ignoring sparse input is another method which reduces this complexity even further.

A major reason for LGBM being widely used is that, it is light in terms of computation required as compared to XGBoost. It uses new methods called gradient based on side sampling(GOSS) and exclusive feature bundling to run even faster. In general what goss does is that it uses importance sampling to be

more efficient to concentrate on data points with larger gradients. Though there is another way to split the data called DART(Dropouts Meet Multiple Additive Regression Trees) which according to the documentation provides more accurate predictions by dropping splits that tend to make model sensitive to the contributions of the few data points.

G. *Neural ODE*

Neural ODE is a new class of neural network, that uses a ODE solver, thereby modifying the current architecture. In the current architecture, the forward propagation of the network is obtained by multiplying the discrete weights and the inputs whereas, Neural ODE provides a continuous depth model with respect to the hidden layers. With this architecture, it only takes constant memory w.r.t the number of hidden layers. Neural ODE has the advantage of adaptive computations for accuracy i.e, if a feature is easily learnable, then it requires fewer time steps to reach the ideal trajectory and if it is complex then more time steps are needed to find the trajectory.

1. ODE Solvers

ODE solvers are used to solve initial value problems, which is given an initial value of a process at time t_0, we need to compute the future state at time t_1. Evaluating the state requires integrating the derivative of the state through time and usually is computed via numerical solvers.
Euler solver is one of the most primitive ODE solver which takes a small step towards the gradient each time. We can represent the Euler's equation as,

$$Z(t+h) = z(t) + h * f(z,t)$$

Where, $Z(t+h)$ is the future state and $z(t)$ is the current state, $f(z,t)$ is the gradient of the state w.r.t time and h is the step size. The Euler method is generally considered to be a poor solver as it accumulates error at each step. However, there has been a lot of progress with ODE solvers over the years ,which makes adaptive step size changes depending upon the complexity of the trajectory and provides a very low error tolerance. We use adjoint method solver to compute the future state.

2. Representing the ODE Net

The Euler's equation is similar to residual networks or RNN where we take the input of the previous state and add it with the output of the current state to get the future state.
First, we take a look at a resnet block(Figure 2.g.2.1) which is a small neural network which defines the update to the weights of the network.

Now in our ODE net, we change the architecture slightly by varying as show in Figure 2.g.2.2

```
def f(z,t,θ):
    return nnet(z, θ[t])

def resnet(z):
    for t in [1:T]:
        z = z + f(z,t,θ)
    return z
```

Figure 2.g.2.1

```
def f(z,t,θ):
    return nnet([z,t], θ)

def resnet(z):
    return ODESolve(f,z,0,1,θ)
```

Figure 2.g.2.2

We basically just changed the resnet function to use an adaptive solver like adjoint method. In the function f, we are feeding the current depth of the network to compute the future state. This is how, the trajectory converges easily for some easily learnable features faster with fewer steps or hidden layers in our case instead of using all the hidden layers. This is how, we are parameterizing the hidden layers.

3. Training the ODE NET

We should not back-propagate the gradients in the network the usual way, although this doesn't result in any error, it takes a longer time to converge. We just need to compute the gradients backwards in time and then update the weights using any standard optimizer algorithm like 'adam' optimizer or stochastic gradient. The algorithm is given below,

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L/\partial \mathbf{z}(t_1)$

$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial \mathbf{z}(t_1)}^\intercal f(\mathbf{z}(t_1), t_1, \theta)$ ▷ Compute gradient w.r.t. t_1

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}, -\frac{\partial L}{\partial t_1}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), -, -], t, \theta$): ▷ Define dynamics on augmented state

  return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \theta}, -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial t}]$ ▷ Concatenate time-derivatives

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}, \frac{\partial L}{\partial t_1}$ ▷ Return all gradients

Figure 2.g.3.1

We can see how the gradients change over the depths with both the resnet(Figure 2.g.3.2) and the ODENet(Figure 2.g.3.3) in the following figure.

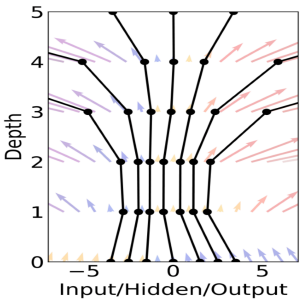

Figure 2.g.3.2

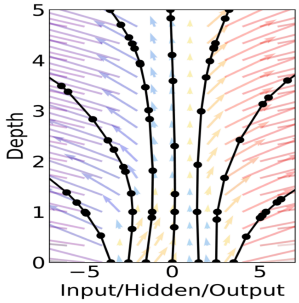

Figure 2.g.3.3

III. RESULTS

LGBM, XGBoost and Naive Bayes models give a ROC score of greater than 0.88 hence, these models tend to do a better job in predicting future values.

Since, this is a financial dataset, we can assume that it usually contains categorical features but, are masked as continuous variables in our case. Therefore, we can infer that tree based methods relatively work well here. Also, in exploratory analysis, we found that the features are independent. Hence, it provides a valid reasoning as to why Gaussian Naive Bayes worked well here.

Further, if the dataset is significantly large enough we suggest to make use of LGBM technique which gives a better score of 0.922 and also it handles overfitting very well. On the contrast, if the dataset is comparatively smaller, Naive Bayes would do satisfactorily well.

Table 3.1 summarize the models implemented to predicted the future transactions for Santander bank and their corresponding ROC score on the test set.

TABLE 3.1

Serial no	ML model	ROC score
1	LDA	0.63
2	QDA	0.635
3	Gaussian Naive Bayes	0.887
4	Neural Networks	0.725
5	Ensemble model using Gaussian Naive Bayes	0.888
6	ADA Boost classifier	0.82
7	XGBoost	0.898
8	LGBM	0.922
9	Neural ODE	0.777

IV. DISCUSSION

At the end of this project we were able to extend our knowledge beyond the class by learning and incorporating boosting techniques like ADA Boost, XG Boost and LGBM classifiers. Also, we were able to get hands-on experience by implementing them on real world datasets.

Exploring Neural ODEs gave us a fresh insight in the recent trends that are gaining popularity in the field of AI. We picked up the knack to read upon technical/research papers as well. Implementing Neural ODEs served us a platform to pickup high level ML APIs and libraries like PyTorch, TensorFlow. Also, these techniques requires high computational powers which cannot be delivered by our machines which paved way to developing and deploying ML models on cloud using GPU computations.

Further, to improve our analysis we are short of domain expertise and interpretability of our dataset. Neural ODE gave a better roc than the Neural network but but we cannot generalize the results as we cannot attribute the increase in ROC score due to the choice of the layers in the ODE block or due to the ODE solver.

Neural ODE takes a long time to compute the gradients in the training phase, hence there's a hardware dependency as well to improve our analysis. The ROC obtained via neural ode is done through back-propagating the network through the adjoint solver . However , this method currently is not compatible with GPU. Hence, we had to iterate for lower number of epochs for the program to terminate in due time.

V. REFERENCES

1. Ricky T. Q. Chen, Yulia Rubanova and Jesse Bettencourt. Neural Ordinary Differential Equations 2018. arXiv:1806.07366
2. Ricky T. Q. Chen. PyTorch implementation of differential ODE solvers. https://github.com/rtqichen/torchdiffeq
3. LightGBM and DART. https://lightgbm.readthedocs.io/en/latest/Features.html, https://arxiv.org/abs/1505.01866.
4. Decision Tree Lecture by Trevor Hastie. https://www.youtube.com/watch?v=wPqtzj5VZus

VI. APPENDIX

Source code: https://github.com/mancunian1792/DS5220

VII. CONTRIBUTIONS

1. Ajeya Kempegowda

Handled the class imbalance issue by incorporating the SMOTE technique. Also, he was involved in investigating the performance of QDA and Naive Bayes techniques. Further, he implemented the ensemble method using Gaussian Naive Bayes and also, he looked into the performance of boosting algorithms — Ada boost and XGboost. Tuning the parameters required a lot of time dwelling in the documentation. Finally, he looked into the Neural ODE research paper and try to grasp/discuss the topics relevant to our cause. This required to invest time in exploring related technical articles as well.

2. Deepanshu Parihar

Researched the new popular gradient boosting methods of LGBM and XGBoost to build highly predictive models. Looked into the new techniques of GOSS,DART etc used by these models that allow them to be fast and accurate at the same time. Also implemented a neural network and fine-tuned the hyper-parameters for both the neural net and boosted trees which consumed a lot of time.

3. Harish Ramani

Carried out the exploratory analysis of the dataset. He, built Linear Discriminant analysis and support vector machines model for this dataset. Also, he spent most of his time reading the Neural ODE paper, understanding and implementing it for the current dataset.