STAT 4012
Statistical Principles of Deep Learning with Business Applications
Group Project - Credit Risk Model with DNN
Final Report

NG Ka Fai[*] SIN Chi Man[†] TSE Chun Hei[‡] YUEN Ho Tin[§]
Department of Statistics, The Chinese University of Hong Kong

November 6, 2024

Abstract

The objective of this project is to develop a deep learning algorithm for default classification without reliance on credit history, enabling creditors to grab more business opportunities by assessing the credit risk of individuals with little or no credit history more accurately. The project focuses are (1) tranforming the massive and unstructured raw data into a clean data matrix with careful preprocessing and feature selection, (2) training an Autoencoder to remedy the curse of dimensionality caused by one-hot encoding, (3) addressing the imbalanced label problem by introducing a focal loss function in the model optimization stage, and (4) comparing the performance of a neural network against other machine learning models in default classification with a huge and complex dataset.

1 Introduction

Retail credit providers, such as banks and credit card companies, build credit scoring models to measure the default risk of a loan applicant before making an offer. Those models typically take the credit history of the applicants as inputs, and provide estimates of the probability of default for the applicants by employing statistical techniques or machine learning algorithms.

One significant challenge under this approach is that lenders can hardly assess the default risk of applicants with little or no credit records as the required attributes may not be available for these individuals. Additionally, outright rejection of such applications may be overly risk-adverse and lots of profit opportunities may be missed out.

In this project, we joined a Kaggle competition hosted by Home Credit Group [1], an international non-bank consumer finance corporation primarily focuses on providing credit to individuals with little or no credit history. Thus, they looks for a machine learning algorithm that can accurately predict the probability of default for this type of individuals with stability over time.

[*]Student ID: 1155158792
[†]Student ID: 1155159539
[‡]Student ID: 1155144187
[§]Student ID: 1155142243

Figure 1: Home Credit Group

The rest of this report is structured as follows: In section 2, we introduce the complex schema of the provided dataset, perform an exploratory data analysis, and describe the data preprocessing steps we made. Section 3 proposes the use of an Autoencoder to learn and extract the latent representation of categorical variables. Then in section 4 the detailed specification of the model we trained to predict defaults is presented with an emphasis on the usage of an alternative loss function for an imbalanced classification task. Finally, the model is evaluated in section 5 with benchmarks from a traditional statistical model and a tree-based machine learning algorithm.

2 Data Preprocessing

In this section, we will give a brief structure of our dataset, how we do exploratory data analysis, what problems we found, and how to handle the data issue in the aggregation process.

2.1 Dataset

The below table and Figure 1 show the complexity of the dataset provided by [1].

Type	Numbers	Examples
CSV files	78	train_base.csv
Data Source	4	Credit Bureau A, Credit Bureau B
Depths	3	train_credit_bureau_a_1_1, train_credit_bureau_a_2_1
Groups of transformation	6	M - Masking categories, A - Transform amount
Features	465	annuity_780A, applicationscnt_464L
Unique Samples	1.6M	case_id from 0 to 1642603

To grab a big picture of the dataset, we group them into three, static data, credit history from credit bureau, and credit history from Home Credit (the host). Static data contains personal information, such as address, age, and current debt. Credit history from a credit bureau contains information on the credit card information owned by the person, for example, the outstanding debt. While the credit history from the targeted company contains similar information with the credit bureau, but with more data on the previous applications.

One of the major challenges of this project is that the dataset we are working with is highly unstructured, requiring careful processing of the data. In the upcoming subsection, we will thoroughly examine the data, discuss the problem at hand, and propose potential solutions.

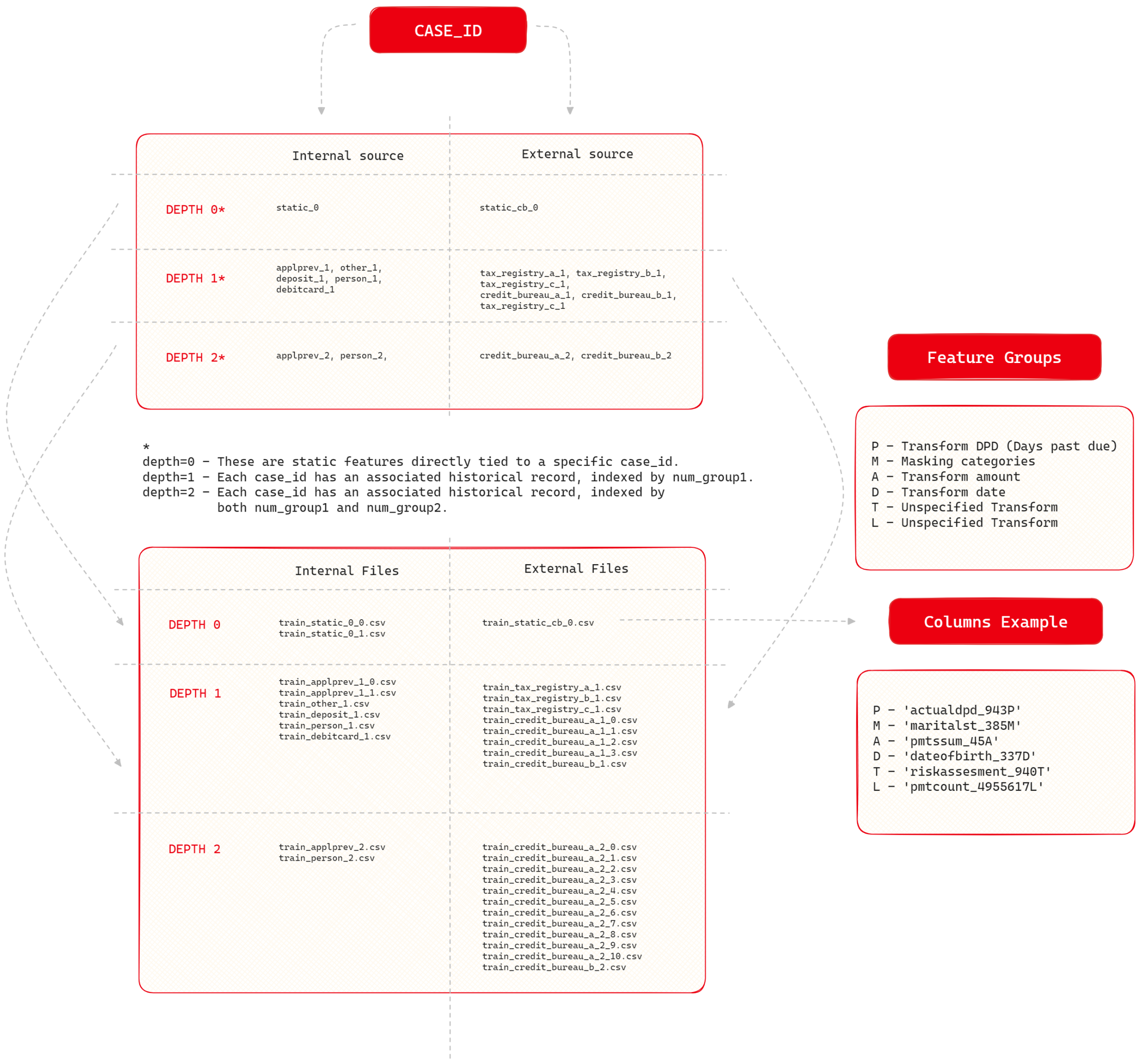

CASE_ID
Internal source
External source
DEPTH 0*
static_0
static_cb_0
DEPTH 1*
applprev_1, other_1, deposit_1, person_1, debitcard_1
tax_registry_a_1, tax_registry_b_1, tax_registry_c_1, credit_bureau_a_1, credit_bureau_b_1, tax_registry_c_1
DEPTH 2*
applprev_2, person_2,
credit_bureau_a_2, credit_bureau_b_2
*
depth=0 - These are static features directly tied to a specific case_id.
depth=1 - Each case_id has an associated historical record, indexed by num_group1.
depth=2 - Each case_id has an associated historical record, indexed by both num_group1 and num_group2.
Internal Files
External Files
DEPTH 0
train_static_0_0.csv
train_static_0_1.csv
train_static_cb_0.csv
DEPTH 1
train_applprev_1_0.csv
train_applprev_1_1.csv
train_other_1.csv
train_deposit_1.csv
train_person_1.csv
train_debitcard_1.csv
train_tax_registry_a_1.csv
train_tax_registry_b_1.csv
train_tax_registry_c_1.csv
train_credit_bureau_a_1_0.csv
train_credit_bureau_a_1_1.csv
train_credit_bureau_a_1_2.csv
train_credit_bureau_a_1_3.csv
train_credit_bureau_b_1.csv
DEPTH 2
train_applprev_2.csv
train_person_2.csv
train_credit_bureau_a_2_0.csv
train_credit_bureau_a_2_1.csv
train_credit_bureau_a_2_2.csv
train_credit_bureau_a_2_3.csv
train_credit_bureau_a_2_4.csv
train_credit_bureau_a_2_5.csv
train_credit_bureau_a_2_6.csv
train_credit_bureau_a_2_7.csv
train_credit_bureau_a_2_8.csv
train_credit_bureau_a_2_9.csv
train_credit_bureau_a_2_10.csv
train_credit_bureau_b_2.csv
Feature Groups
P - Transform DPD (Days past due)
M - Masking categories
A - Transform amount
D - Transform date
T - Unspecified Transform
L - Unspecified Transform
Columns Example
P - 'actualdpd_943P'
M - 'maritalst_385M'
A - 'pmtssum_45A'
D - 'dateofbirth_337D'
T - 'riskassesment_940T'
L - 'pmtcount_4955617L'

Figure 2: Data Schema Overview. Retrieved from [4].

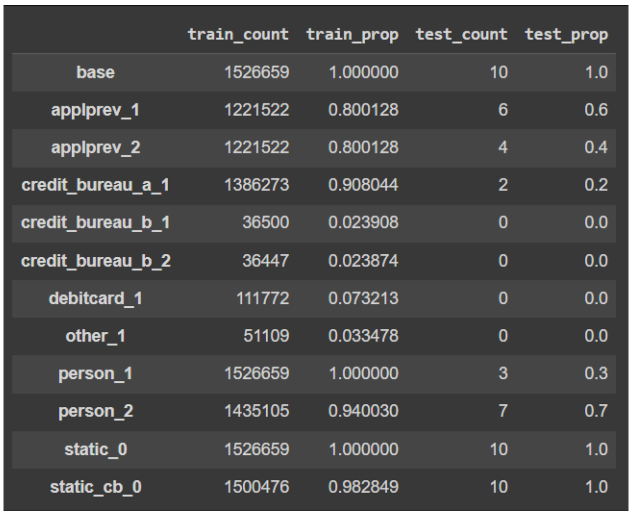
2.2 Data Structure

In our project, we aim to focus on the probability of default of each person. Each person is related with a unique case_id, while the output is either 1 or 0 to represent default or non-default which is stored in the base table. While other data files, such as train_credit_bureau_a_1_1, and train_static_0_0, stored the person's credit history and his/her static data. It is noted that not all case_id in the base file can be found in the other data files as well. For example, there are 1.6M unique case_id in base_file, and there are only 36500 unique case_id in train_credit_bureau_b_1. This suggests a problem that there will be a lot of missing data and create a very sparse dataset. We will further discuss this problem in a later section.

On the other hand, the data exhibits varying levels of depth, indicating that one individual can have multiple entries in a data file. Consider a simple example: a person may apply for multiple credit cards, and for each card, they have made several interest repayments. As a result, the dataset possesses three levels of data depth: individuals, credit cards, and payment records. However, in any Deep Neural Network (DNN), we typically establish a one-to-one mapping between our input (attributes of a case_id) and the output (probability of default). Given the layers' depths, it is not feasible to directly input them into the DNN. To address this issue, data aggregation is necessary, and we will delve into this topic in a subsequent section.

2.3 Missing Data

As we discussed we encountered the missing data issue above. We need to decide how to handle the missing data.



	train_count	train_prop	test_count	test_prop
base	1526659	1.000000	10	1.0
applprev_1	1221522	0.800128	6	0.6
applprev_2	1221522	0.800128	4	0.4
credit_bureau_a_1	1386273	0.908044	2	0.2
credit_bureau_b_1	36500	0.023908	0	0.0
credit_bureau_b_2	36447	0.023874	0	0.0
debitcard_1	111772	0.073213	0	0.0
other_1	51109	0.033478	0	0.0
person_1	1526659	1.000000	3	0.3
person_2	1435105	0.940030	7	0.7
static_0	1526659	1.000000	10	1.0
static_cb_0	1500476	0.982849	10	1.0

Figure 3: Proportion of case_id in different data files

2.3.1 Dropping Tables

In Figure 3, we define the unique case_id in each table as the train_count. For example, there are 1221522 unique case_id, so the train_count is 1221522. Then, we divide this number with the train_count in the base (1526659) and get the train_proportion(0.80). If we then join base and applprev, 20% of case_id cannot be matched and present as missing.

It may not be a problem if train_prop is large enough, the missing values can be treated as numerical features or categorical features and further handled by the model. However, if the train_prop is too small, then the majority of the data or particular features will be missing and the model does not have enough useful data to conclude the features. For instance, credit_bureau_b_1 and credit_bureau_b_2 have train_prop less than 0.025 and many missing values will created if we join them with the base table.

Striking the balance between model dimensional and complexity, we decided to drop the tables with a train_prop smaller than 0.05. We believe the model cannot use these missing data to draw very useful information and make the prediction in default probability. Besides, since our purpose is to model the default probability with people who have little or no credit history, tables of credit information from credit bureaus should also be dropped to avoid the model from being over-relying on credit history to make predictions.

2.3.2 Dropping Features

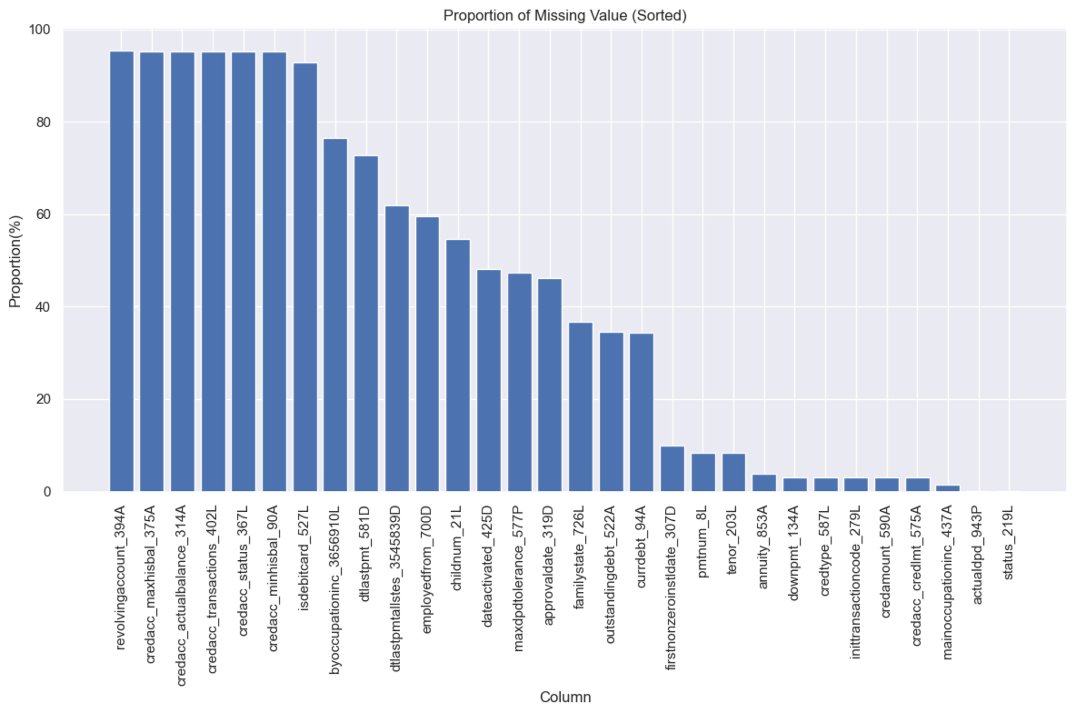


Figure 4: NA Columns

Upon dropping the tables, it becomes evident that a significant amount of data is missing when we attempt to join them. This is primarily because each table contains numerous missing data entries before the join. Figure 4 demonstrates this issue by showcasing the missing proportion of each feature in the appl_prev table.

As observed in the leftmost columns, the missing proportion can exceed 90%. Consequently, the problem of missing values resurfaces, impeding the neurons' ability to effectively capture the impact of these features due to the overwhelming dominance of missing values. As a solution, we have decided to eliminate columns with missing values exceeding 90%.

2.3.3 Imputation of Remaining Missing Values

After dropping tables and features, the next task is to impute missing values for the features that we want to preserve. Conventional approaches involve filling in the missing values through interpolation methods, such as using the global mean or mode. However, this method relies on the assumption that the missing data is missing completely at random (MCAR). Other statistical or machine learning methods such as EM algorithms, Multiple Imputation or K-nearest Neighbors also require the assumption of missing at random (MAR), meaning that we should be able to find the conditional distribution of missing variables by some of the observed variables. Such assumptions appear to be too strong in our case since individuals can intentionally choose not to submit certain data to avoid a decrease in their credit score. For instance, in traditional credit scorecards, there is a strong correlation between credit rating and monthly income. Individuals with lower incomes intentionally conceal this information. In other words, chances are most of the missing values are missing not at random (MNAR). Any imputation scheme can distort the information provided by the dataset as 'missing' itself can also be an important feature.

Therefore, we used the following method to represent missing data instead of imputing data with strong assumptions: If the missing datum is categorical, we simply treat missing as a category. We identify the missing from the data file and missing due to joining the tables as NaN and Nan_byMerge respectively. As for numerical features, it is noted that most of the data are positive values, so we fill the missing values by out-of-range impossible values -1, and -2. By doing this, when we apply one hot encoding to the categorical features, the corresponding Beta will increase if missing plays an important role in the dataset. Also, the non-linearity of Neural Networks can identify the missing data and do not affect the original analysis.

2.4 Data Aggregation

As previously discussed, there can be multiple records per person (case_id) so we have to concatenate them into one record per person before inputting the data into our models. As there are different types of data, we do the concatenation according to the below schema.

Object Type	Aggregation Method
Date	Earliest and Latest Observation
Categorical	First and Last Observation
Numerical	Mean and Standard Deviation
Counter	Sum of total records exist in the table

Noted that dates cannot be directly inputted into the model as they are not numerical. Our approach to address this issue is the find the time difference between the observed data and the date of decision in the base table.

While some of the tables are dropped due to the sparsity after concatenation, we utilize the presence record of cases in each table as features and mark them as counters.

2.5 Imbalance Dataset

The problem of high label imbalance is common in default prediction as we do not expect any healthy financial institution would have the number of defaulted loans taking up a significant proportion of total loans issued. From Figure 5, we can see that near 97% of the labels have the value 0, which represents non-default, and only around 3% of the cases are indeed default. It makes the model extremely difficult to learn how to predict the minority class (i.e. the class of our interest).

When dealing with imbalanced datasets, common approaches include up-sampling and down-sampling. Up-sampling involves generating additional data or duplicating existing data to balance the proportions, aiming for a balanced distribution of label. However, this approach has evident drawbacks. If up-sampling is employed, it would require creating more than 1.4 million samples, potentially leading to severe over-fitting if the random-over-sample method is utilized. Alternatively, if the SMOTE method is used, we take the risk of receiving non-sense samples. Most importantly, creating 1.4 million of additional samples almost doubles the size of our dataset which is already huge, which in turn requires much more computational resources and increases computational cost during model training. On the other hand, deleting over 1.4 million data points would result in the loss of almost all information contained within the dataset. Therefore, neither of them is sound when the sample size is huge.

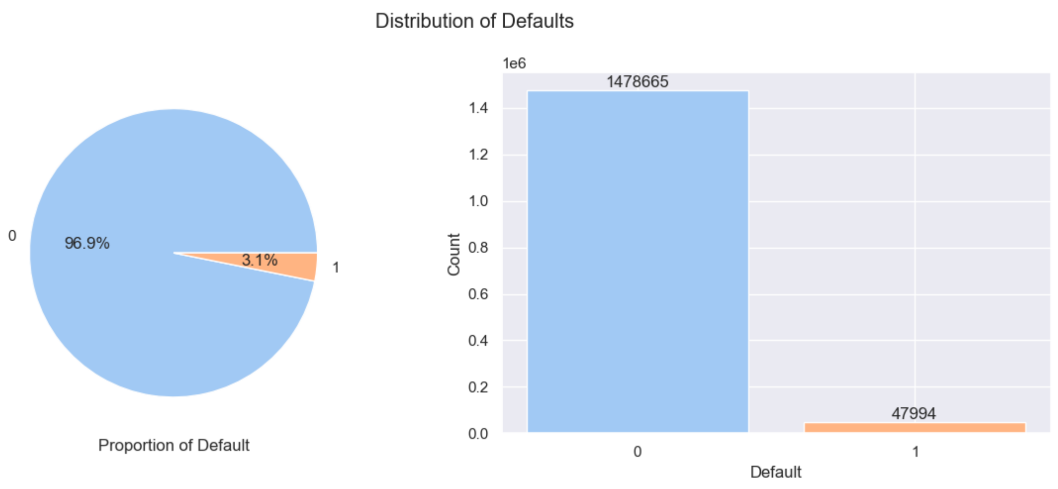


Figure 5: Imbalance Label

Taking into account the limitations associated with up-sampling and down-sampling techniques, we have made the strategic decision not to address the imbalance dataset issue during the data processing phase. Instead, we have chosen to tackle this challenge during the model development stage by adopting the *Focal Loss*, an alternative loss function invented by the Facebook AI research group [2] that specifically addresses the problem of imbalance classification. Technical details will be discussed in section 4.3.

2.6 Train Test Split

The testing set provided by [1] is actually a 10-sample dataset without labels and is to be used to compute scores for the competition only. Thus we have to perform train-test split on our own to measure the performance of our models. As each record is associated with a date of decision and Week_Num indicating the week that the decision of default identification was made, to test whether the model performance can be generalized over time and achieve the objective of model stability, we split the data according to Week_Num, where records after the 80-th week are allocated to the testing set[1]. Finally, the shape of our training and testing datasets is presented below.

DataFrame	Shape
`X_train`	(1388865, 303)
`Y_train`	(1388865, 1)
`X_test`	(137794, 303)
`Y_test`	(137794, 1)

[1]The 10-sample testing set provided by Kaggle also split the original data in this way and they set the cut-off at the 95-th week.

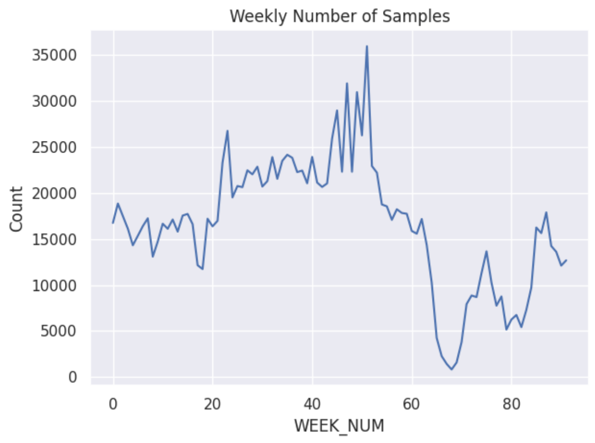


Figure 6: Weekly Number frequency

3 Features Engineering

After all the data manipulation steps, we were left with 303 features, 69 of which are categorical. While numerical features were ready as inputs of our models, we need an encoding scheme for the categorical features to transform them into numerical values.

3.1 One-hot Encoding

Since most of them are not ordinal in nature, using ordinal encoding did not make sense and the only option left is one-hot encoding. However, the dimensionality of our data matrix will increase drastically as some of our features have more than hundreds or even thousands of unique classes.

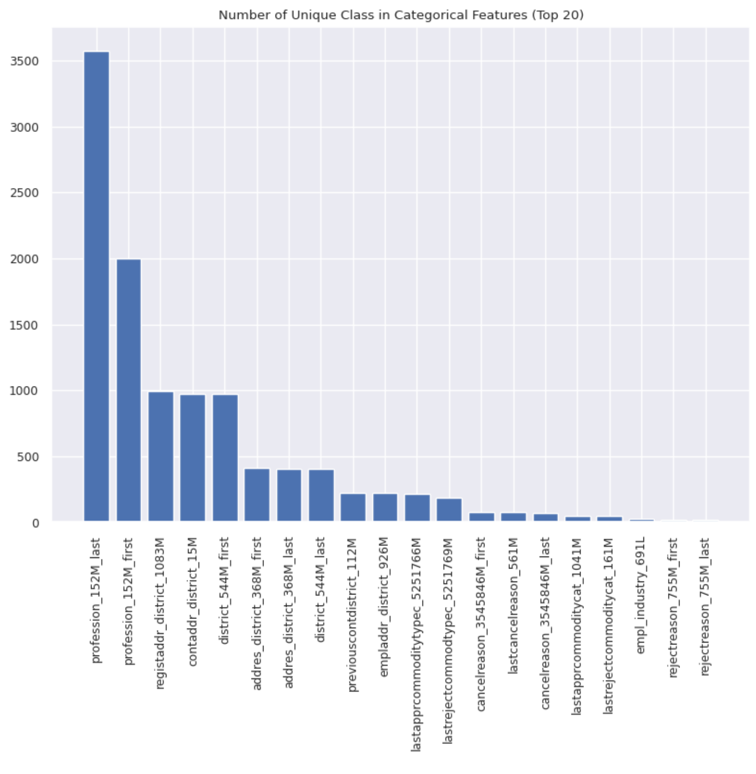


Figure 7: Number of unique class in categorical features

If the one-hot encoded features are directly used for training models without further selection and preprocessing, our final data matrix will contain more than 10,000 covariates, causing memory issues and a high risk of overfitting. To address the problem, we first drop the top 5 variables shown in Figure 7 so that one-hot matrix were left with 2,787 columns, but it is still too large for us to avoid memory issues and still outnumbers numerical variables by a big margin. Therefore, our next step is to compress the dimension further by training an Autoencoder.

3.2 Autoencoder

In a nutshell, an Autoencoder is an unsupervised deep learning model that uses a symmetric neural network to extract key latent information from a dataset with huge dimensions, similar to PCA but capable of capturing non-linear relationships among covariates. It consists of two architecturally symmetric sub-networks: the encoder and the decoder. The encoder transforms inputs into latent variables with much lower dimensions, whereas the decoder reconstructs the inputs with the latent variables. The Autoencoder is then trained by minimizing the reconstruction loss and our final goal is to extract the latent variables by using the encoder part of the network.

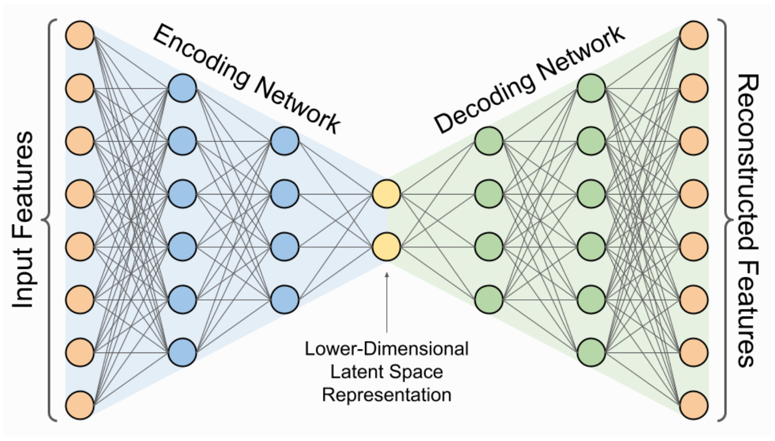


Figure 8: Architecture of an Autoencoder, Retrieved from [3].

In our project, we insert one hidden layer between the input layer and the latent representation. Since our input data are binary in a sense that $x_{ij} \in \{0, 1\} \; \forall \, i, j$, binary-cross entropy is chosen as the loss function and the output layer of the decoder is activated by the sigmoid function. As for the hidden layers and the latent layer, a LeakyReLU activation is used in hope of preserving more information by introducing a small slope a at the negative part of the input axis. In contrast, any negative input will be deactivated in a classical ReLU function, which may cause undesirable information loss.

$$\text{LeakyReLU}(z) = \max(az, z) \quad \text{for a small positive } a \tag{3.1}$$

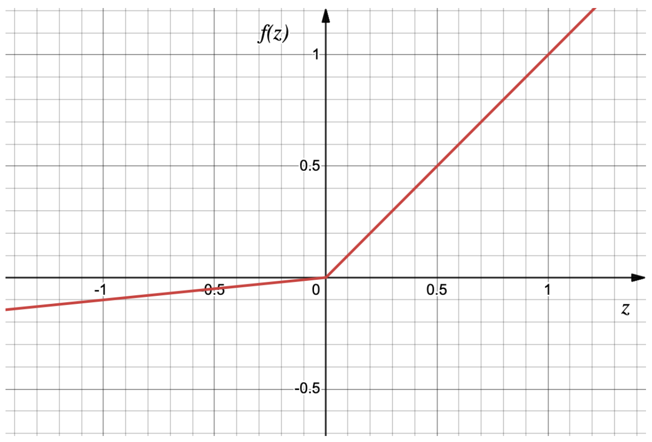


Figure 9: A LeakyReLU function with $a = 0.1$.

We consider the following hyperparameters when building an Autoencoder:

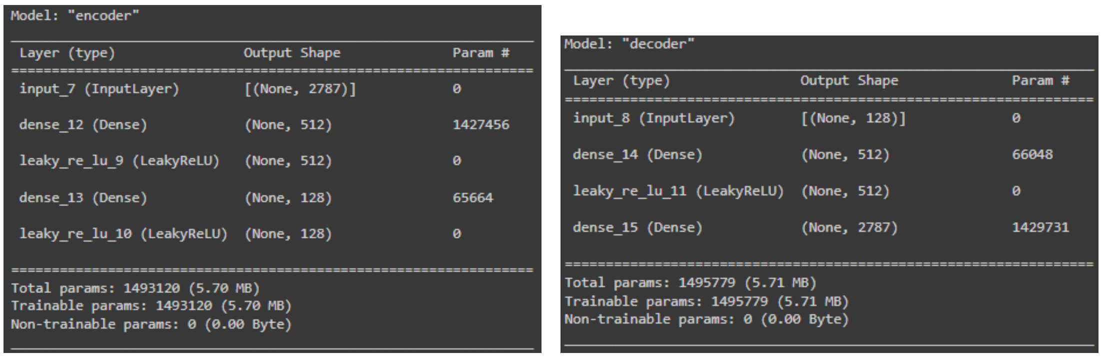
- Number of hidden units: [256, 512],
- Number of latent variables: [64, 128],
- The slope a in LeakyReLU activation function: [0.01, 0.03].

By considering all possible combinations of the hyperparameter and performing hyperparameter-tuning with validation set, the model with the smallest reconstruction loss is selected to compress our dataset.

Number of hidden units	Number of latent variables	a	Validation Loss
256	64	0.01	0.0045
256	128	0.01	0.0021
512	64	0.01	0.0025
512	128	0.01	0.0014
256	64	0.03	0.0019
256	128	0.03	0.0022
512	64	0.03	0.0024
512	128	0.03	0.0019

Table 1: Performance of Autoencoders constructed by combinations of hyperparameters

The fourth model is the best in terms of validation loss, thus is chosen as our encoder to compress the one-hot matrix and transform it into latent variables. In other words, we use the encoder part of the model to compress 2,787 one-hot features into 128 numerical latent variables, successfully avoiding memory issues and the curse of dimensionality by an unsupervised machine learning approach.



```
Model: "encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 2787)]            0

 dense_12 (Dense)            (None, 512)               1427456

 leaky_re_lu_9 (LeakyReLU)   (None, 512)               0

 dense_13 (Dense)            (None, 128)               65664

 leaky_re_lu_10 (LeakyReLU)  (None, 128)               0

=================================================================
Total params: 1493120 (5.70 MB)
Trainable params: 1493120 (5.70 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
Model: "decoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_8 (InputLayer)        [(None, 128)]             0

 dense_14 (Dense)            (None, 512)               66048

 leaky_re_lu_11 (LeakyReLU)  (None, 512)               0

 dense_15 (Dense)            (None, 2787)              1429731

=================================================================
Total params: 1495779 (5.71 MB)
Trainable params: 1495779 (5.71 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 10: The Structure of Our Autoencoder

The DataFrame storing the latent variables is then concatenated with the numerical features to form the features matrix X that is ready to train the models in the next section.

4 Model Development

In this section, we will discuss the development of DNN networks, with a focus on optimizing their performance. As the scope of this project is limited by the nature of the data provided, we found it not suitable to employ advanced deep learning techniques such as CNN, LSTM, GAN or Transformer[2]. Therefore, it is more appropriate to work with the vanilla form of artificial neural networks, the Multilayer Perceptron (MLP). We will explore different parameters and address challenges like the dying ReLU problem and imbalanced datasets. To achieve this, we will use a 5-fold cross-validation technique and experiment with activation and loss functions to create effective DNN models for our specific use case.

4.1 Model Architecture and Regularization Techniques

To optimize the performance of each hidden layer in our DNN network, we implement a specific sequence of layers with the following steps:

1. Dropout Layer: We apply a dropout layer to the input of each hidden layer. Dropout randomly sets a fraction of inputs to zero during training, preventing overfitting and promoting the learning of robust features.
2. Dense Layer with $\mathscr{L}_1$ Regularization: After the dropout layer, we add a dense layer with $\mathscr{L}1$ regularization. $\mathscr{L}_1$ regularization adds a penalty term to the loss function, encouraging sparsity in the weights and leading to simpler and more interpretable models while preventing overfitting.
3. Activation Function: The output of the dense layer is passed through a Leaky ReLU activation function. Leaky ReLU allows for small negative values, addressing the dying ReLU problem and promoting neuron activation throughout the network. This activation function enhances the network's expressive power and facilitates better learning.
4. Batch Normalization: Finally, we apply batch normalization to the output of the Leaky ReLU activation function. Batch normalization normalizes the activations, improving stability and convergence speed during training. It addresses issues like internal covariate shift and accelerates the training process of the DNN network.

[2]Our data are neither sequential nor in matrix form per record in general (except credit history which we found inappropriate to include them as to cater the business nature of the host).

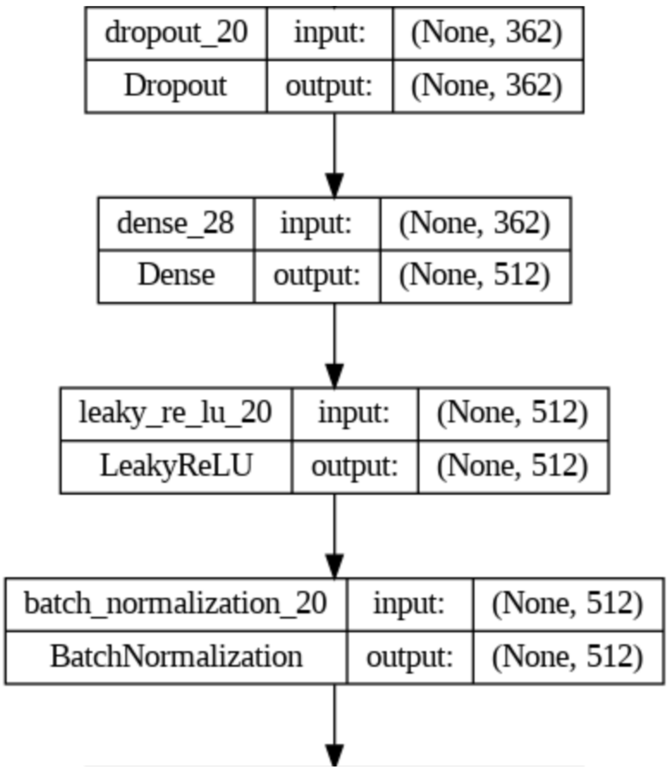


Figure 11: Layers Insider Hidden Layer

By following this architectural sequence and incorporating regularization techniques, we aim to optimize the performance of each hidden layer in our DNN network. This ensures effective learning and generalization from the data.

4.2 Addressing the Dying ReLU Problem with Leaky ReLU

During the training of our DNN models, we could encounter the issue of dying ReLU, which can hinder the learning process and impact the overall performance of the network. The dying ReLU problem typically arises when the weights of the network are inadequately initialized or undergo significant updates, leading to ReLU units becoming unresponsive and failing to activate.

To mitigate the dying ReLU problem and promote effective weight updates and learning in our neural network, we employ the Leaky ReLU activation function adopted in section 3.2 again. The Leaky ReLU introduces a small slope for negative inputs, ensuring non-zero outputs even when the input is negative. Unlike the standard ReLU function, which sets negative inputs to zero, the Leaky ReLU allows for a small negative output. This small negative output ensures that the neurons remain active and responsive to negative inputs, preventing them from becoming completely inactive.

By incorporating the Leaky ReLU activation function into our DNN models, we address the issue of dying ReLU and facilitate the learning process.

4.3 Addressing Imbalanced Datasets and Classification Focus with Focal Loss

While the cross-entropy loss function is commonly used for classification tasks, it struggles with imbalanced datasets and assigns significant loss to correctly classified samples. As we observed in section 2.5, our dataset is indeed highly imbalanced and it is a common problem encountered in default classification models. Therefore, we employ the focal loss function designed by [2] instead of the traditional cross-entropy loss function, specifically to address imbalanced datasets and enhance the model's focus on challenging samples. For a binary classification task with one output neuron, the function reads the form

$$\mathrm{FL}(y,p) = -y\left[\alpha(1-p)^{\gamma}\log(p)\right] - (1-y)\left[(1-\alpha)p^{\gamma}\log(1-p)\right], \tag{4.1}$$

which is equivalent to

$$\mathrm{FL}(y,p) = \begin{cases} -(1-\alpha)p^{\gamma}\log(1-p) & \text{if } y=0, \\ -\alpha(1-p)^{\gamma}\log(p) & \text{if } y=1. \end{cases} \tag{4.2}$$

The function is defined by two hyperparameters, α and γ:

1. α assigns different weights to different classes. For example, if we have a highly imbalanced dataset where the default class comprises only 3% of the samples, we can assign an α value of 0.97 to the default class and 0.03 to the non-default class. By assigning disparate weights, the focal loss accounts for class imbalance and gives more attention to the minority class during training. This helps the model learn more effectively from the imbalanced data and improves its ability to classify the minority class accurately. As α directly addresses the problem of imbalanced classification, it does not require any tuning.
2. γ adjusts the focus on the correctness of classification. In traditional classification scenarios, a predicted probability above 0.5 is considered a correct classification, while a probability below 0.5 is considered incorrect. However, the original cross-entropy loss assigns substantial loss magnitudes to both correctly and incorrectly classified samples. With the focal loss function, as the γ value increases, the loss assigned to correctly classified samples progressively diminishes, approaching a value close to 0. Conversely, the loss assigned to incorrectly classified samples remains significant. This pattern can also be observed for samples with a label of 0. A higher γ value encourages the loss function to concentrate on hard-to-classify samples, effectively giving more attention and effort towards correctly identifying and addressing difficult instances. Although [2] suggests that $\gamma = 2$ works the best for their object detection task, we suspect that the choice of optimal γ is subjected to the dataset and problem context, thus hyperparameter tuning is required to find the best γ in our case.

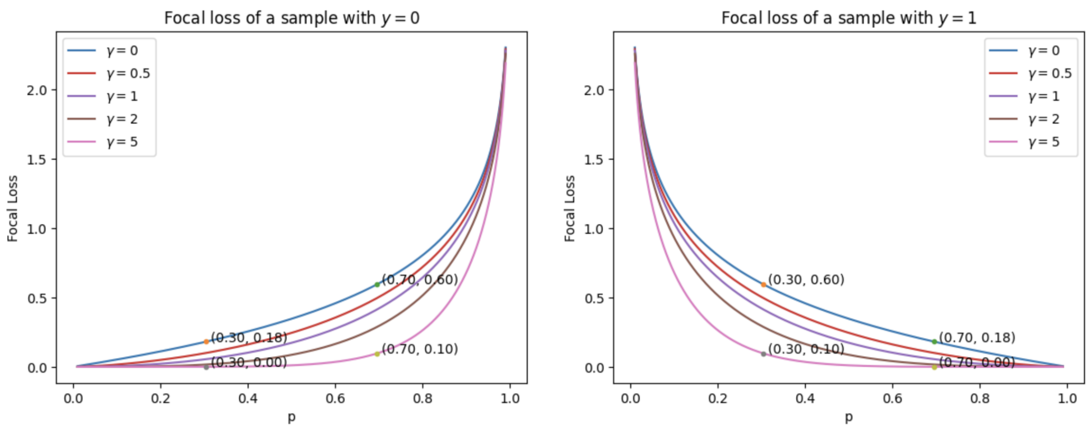


Figure 12: Focal Loss of a Sample with Different Labels

By incorporating the focal loss function into our model, we address the challenges posed by imbalanced datasets and improve the model's focus on challenging samples. α allows us to assign different weights to different classes, balancing the learning between the majority and minority classes. γ further refines the loss assignment by emphasizing the importance of difficult samples and reducing the loss magnitude for correctly classified instances. This combination improves the model's ability to handle imbalanced data and focus on challenging cases.

4.4 Optimization of DNN Network with Hyperparameter Tuning

To optimize our DNN network's performance, we performed a grid search on hyperparameters with 5-fold cross-validation. This approach enables us to systematically evaluate various parameter settings and refine the model architecture, leading to improved overall effectiveness.

4.4.1 Hyperparameters for Model Optimization

To optimize our DNN network, we experiment with different hyperparameters. The following table summarizes the hyperparameters used:

- Model Architecture
  - Shallow Network: [1024, 256, 32]
  - Deep Network: [512, 256, 256, 128, 128, 64]
- Slope of LeakyReLU Activation Function ($a_{\text{LeakyReLU}}$)
  - 0, 0.02, 0.04
- Focal parameter of the Focal Loss Function (γ_{FL})
  - 0, 1, 2

4.4.2 5-Fold Cross-Validation

We implement a 5-fold cross-validation technique to assess and validate the performance of our DNN network. This method involves dividing the dataset into five subsets or folds, where each fold serves as both a training set and a validation set during different iterations. By rotating the folds, we ensure that each subset is used as a validation set at least once, providing a robust evaluation of the model's performance across different subsets of the data, avoiding overfitting to a single validation set.

Throughout the 5-fold cross-validation process, we systematically modify the hyperparameters of our DNN network, such as the model architecture, the slope of the Leaky ReLU function, and the gamma value of the Focal Loss function. By assessing the model's performance on each fold using different combinations of these parameters, we can determine the optimal settings that yield the highest accuracy and best generalization capability.

4.4.3 Best Performing DNN Models after Hyperparameter Tuning

During the model training process, we performed hyperparameter tuning to optimize the performance of our models. The selection of the best models was primarily based on their area under curve (AUC) scores on the validation set.

$\overline{\text{Training Time}}$ (s)	$\overline{\text{Testing Time}}$ (s)	$a_{\text{LeakyReLU}}$	γ_{FL}	Model Architecture	AUC mean	AUC std	Rank
40.5261	17.8359	0.0400	0	[1024 256 32]	0.7958	0.0014	1
37.7294	18.1739	0.0200	0	[1024 256 32]	0.7953	0.0011	2
53.5542	19.7156	0	0	[512 256 256 128 128 64]	0.7951	0.0016	3
36.8970	17.5628	0	0	[1024 256 32]	0.7950	0.0026	4
54.2916	19.8816	0.0200	0	[512 256 256 128 128 64]	0.7944	0.0024	5
54.8498	19.9869	0.0400	0	[512 256 256 128 128 64]	0.7940	0.0027	6
39.7855	17.8273	0.0200	1	[1024 256 32]	0.7915	0.0013	7
37.6410	17.7284	0.0400	1	[1024 256 32]	0.7912	0.0009	8
55.1952	20.1596	0.0200	1	[512 256 256 128 128 64]	0.7907	0.0020	9
54.5182	19.8718	0	1	[512 256 256 128 128 64]	0.7906	0.0010	10
38.9225	17.8173	0	1	[1024 256 32]	0.7905	0.0022	11
55.0820	20.0282	0.0400	1	[512 256 256 128 128 64]	0.7890	0.0035	12
37.7867	17.9051	0.0200	2	[1024 256 32]	0.7854	0.0023	13
37.9909	17.7524	0	2	[1024 256 32]	0.7845	0.0031	14
35.4543	17.7904	0.0400	2	[1024 256 32]	0.7841	0.0019	15
56.4450	20.0343	0	2	[512 256 256 128 128 64]	0.7819	0.0016	16
54.7687	19.8924	0.0200	2	[512 256 256 128 128 64]	0.7815	0.0025	17
53.1418	19.9798	0.0400	2	[512 256 256 128 128 64]	0.7813	0.0028	18

Table 2: Cross Validation Result

Upon analyzing the results, we observe that the models with different hyperparameter configurations exhibit similar performance in terms of validation AUC. All of the top five models achieved an AUC score of 0.79, indicating comparable discrimination capability.

This observation suggests that the hyperparameters we tuned may not have a significant effect on the model's performance, at least in terms of the validation AUC. It is possible that the selected hyperparameters resulted in a reasonably optimal configuration, leading to similar performance across different models.

4.4.4 Loss Analysis on the Best Model

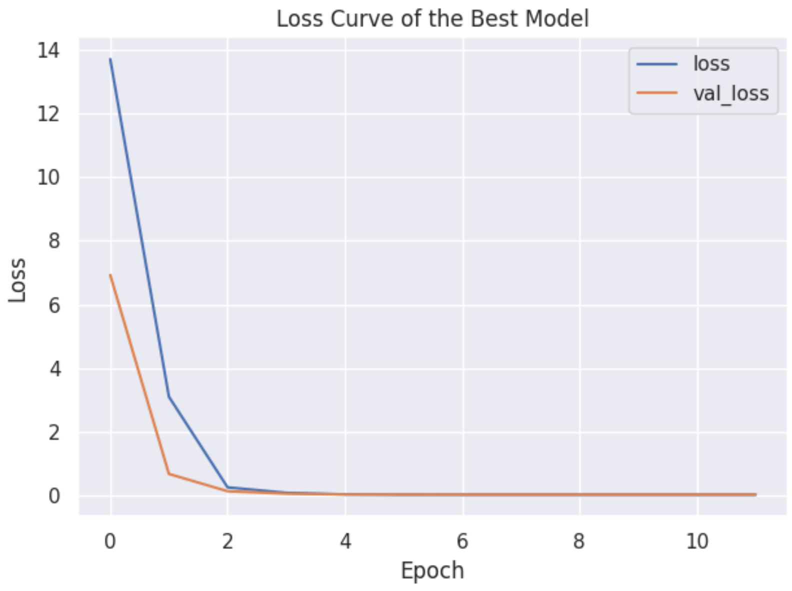


Figure 13: Training and Validation Loss

The graph illustrates the trend of loss during training. The noticeable decrease in loss for both training and validation data indicates the model's effective learning without overfitting. This reduction signifies the model's improvement in accurate predictions, minimizing the disparity between predicted and actual values. Also noted that we employed the early stopping callback so that the training will terminate automatically when no improvement has been made on reducing validation loss.

4.5 Analysis of Hyperparameters

In our evaluation of DNN models, we examined the impact of different Leaky ReLU slope and focal loss gamma values on model performance, specifically focusing on the validation AUC. We wanted to determine if the choice of slope and gamma significantly affected the model's performance.

The results demonstrated that the validation AUC remained relatively stable across various model architectures, ranging from about 0.78 to 0.79. This suggests that the selection of slope and gamma did not have a substantial impact on the model's ability to distinguish between classes.

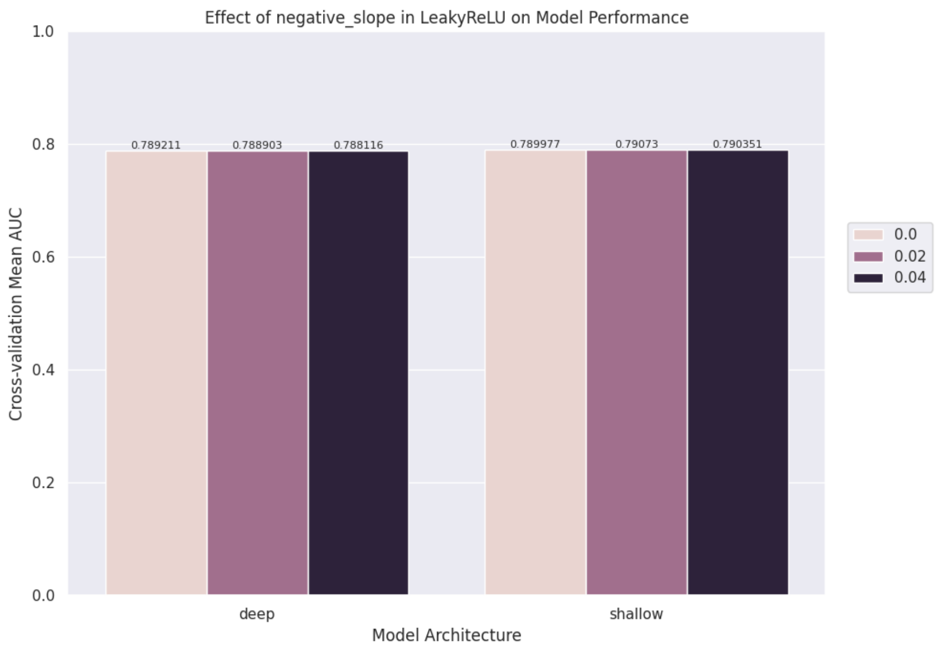


Figure 14: Effect of Negative Slope in Leaky ReLU on Model Performance

One possible explanation for these findings is the inclusion of additional regularization techniques in our model, such as $\mathscr{L}_1$ regularization, batch normalization, and dropout layers. These techniques help regularize the weights and enhance stability during the learning process.

While the Leaky ReLU activation function addresses the dying ReLU problem and gamma in focal loss adjusts attention based on task difficulty, the presence of regularization, batch normalization, and dropout layers in our model may have diminished the potential influence of slope and gamma on performance.

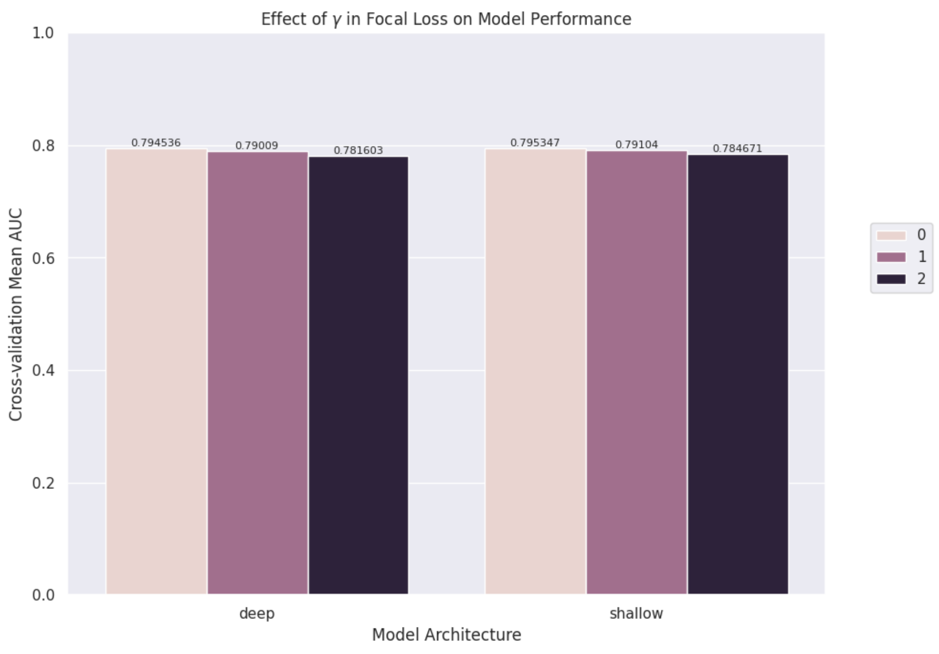


Figure 15: Effect of γ in Focal Loss on Model Performance

5 Performance Evaluation

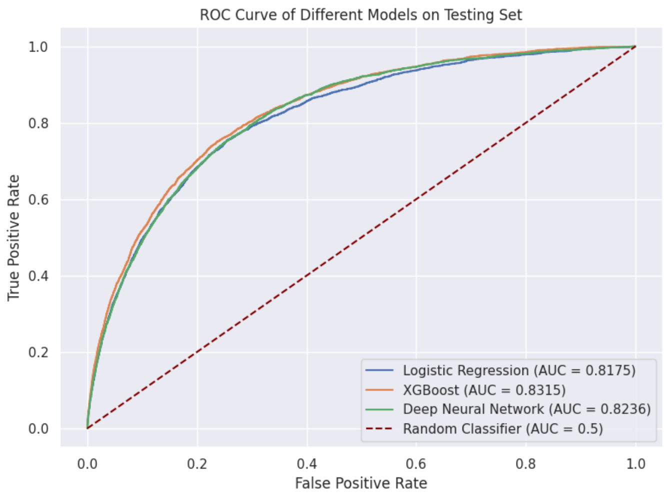


Figure 16: ROC Curve of different models on testing set

Finally, we evaluate the performance of our best model in table 2 using the testing set. For the purpose of benchmarking, we also train two additional models for comparison so that we have in total 3 candidates for evaluation:

- A linear model: Logistic Regression with PCA transformed features[3] (blue line in figure 16);
- A tree-based model: XGBoost (orange line in figure 16);
- The best DNN trained in the last section (green line in figure 16).

As seen from figure 16, all three candidates attain similarly decent results. The fact that the Logistic Regression model has a testing performance comparable to more complex models like XGBoost and DNN is

[3]We picked top components that in total taking up 99% of the total variance of the feature matrix.

notable. The similar performance of different models suggests that the data preprocessing steps, including cleansing and feature engineering, played a critical role in determining the success of the models. This highlights the importance of thorough data preparation, potentially outweighing the impact of selecting between various advanced modeling techniques.

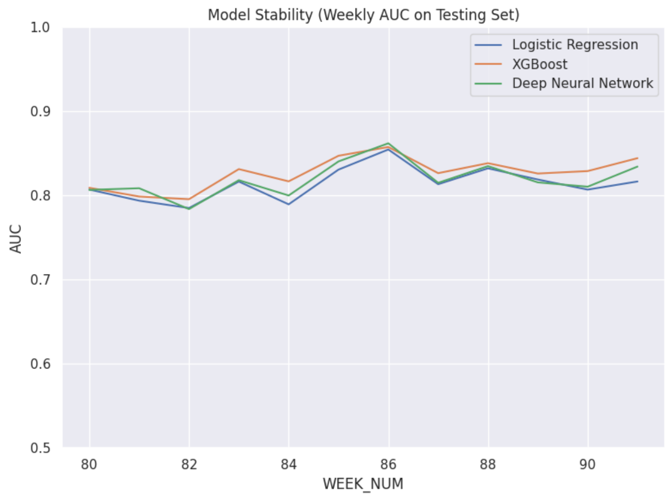


Figure 17: Model stability on testing set

We also evaluated model stability throughout the testing set using AUC in figure 17. Again, all three models have AUC of over 0.8 for most of the time in the testing period, which suggests good model stability.

6 Conclusion

This project explores the use of Deep Neural Network / Multilayer Percepton for dimensionality reduction and default prediction. Our result shows that (1) the problem of data sparsity caused by one-hot encoding can be well-addressed by training an Autoencoder to compress the dimensionality to a desired level; (2) upsampling is not the only way to cope with the problem of imbalanced labels, and using a focal loss function is a decent alternative with less computational cost and memory resources required; and (3) neural networks may not outperform simpler or more interpretable models when the data preprocessing and feature engineering are performed properly and carefully.

A List of Python Programs and Notebooks

- `01_Data_preprocessing.ipynb`
- `02_Autoencoder.ipynb`
- `03_Model_Development.ipynb`
- `utils.py`

References

[1] D. Herman, T. Jelinek, W. Reade, M. Demkin, and A. Howard. Home credit - credit risk model stability, 2024. URL https://kaggle.com/competitions/home-credit-credit-risk-model-stability.

[2] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017. URL http://arxiv.org/abs/1708.02002.

[3] R. O'Connor. Introduction to variational autoencoders using keras, 2022. URL https://www.assemblyai.com/blog/introduction-to-variational-autoencoders-using-keras/.

[4] S. Saharovskiy. [home credit crms, 2024] eda and submission, 2024. URL https://www.kaggle.com/code/sergiosaharovskiy/home-credit-crms-2024-eda-and-submission/notebook.