AmEx Credit Card Default Prediction: A Data-Driven Analysis

Aryamaan Saha (be19b014@smail.iitm.ac.in)
Devansh Sanghvi (be19b002@smail.iitm.ac.in)
Dhananjay Balakrishnan (me19b012@smail.iitm.ac.in)
Siddharth Betala (be19b032@smail.iitm.ac.in)
MS4610 - Introduction to Data Analytics Project (Aug-Dec 2022)

***Abstract*—Using the data containing the financial behaviour of costumers in the past, we would like to predict whether they are likely to default on their payment in the future. In this report, we will discuss the end-to-end approach we have taken with respect to using the data available to making predictions on the given test dataset, and attempt to explain the reasoning and philosophy behind how we have designed our model and how we improved our performance by various methods.**

I. Overview

Our dataset consists of the customer application and bureau data with 51 variables, each denoting various aspects/metrics of the customer's card history. Based on the information at hand, we are required to predict whether the customer will default in a span of 1 year or not – as a card should ideally be issued to an individual who will not default.

The structure of this report will be as follows. First, we will look at the data, make some observations about the nature of it and talk about how we have pre-processed it. Following this, we will motivate the need for the various models we have built and in particular, the bagging-based ensemble model we have built for the same. Towards the end, we will discuss the experimental settings and how we have fine-tuned the hyperparameters to attain optimal results. Finally, we will discuss potential improvements to our model and make a few observations about the task at hand.

II. Dataset

A. Description of the Dataset

We are given 51 variables, named 'mvar1', 'mvar2', and so on, and in our train data, we have a total of 83000 entries. We are also given a data dictionary which explains the relevance/definition of each feature. One thing we notice immediately on looking at the data is that there are a lot of missing values, and these have been annotated unevenly by various labels such as 'missing', 'na', NaN, and 'N/A'.

On outputting the number of unique values in each feature, we have concluded that 'mvar47' and 'mvar50' are categorical variables. 'mvar47' has 'L' or 'C' based on the type of product that the applicant has applied for and 'mvar50' has the values 4 or 5 – and apparently represent a compound feature created using 2 other variables.

Looking at the various descriptions of the various features in the dataset as given in the file 'Data_Dictionary.csv', it does appear as though quite a few of the features might be correlated. This is demonstrated above in figure 1 where we have plotted the correlation matrix heatmap.

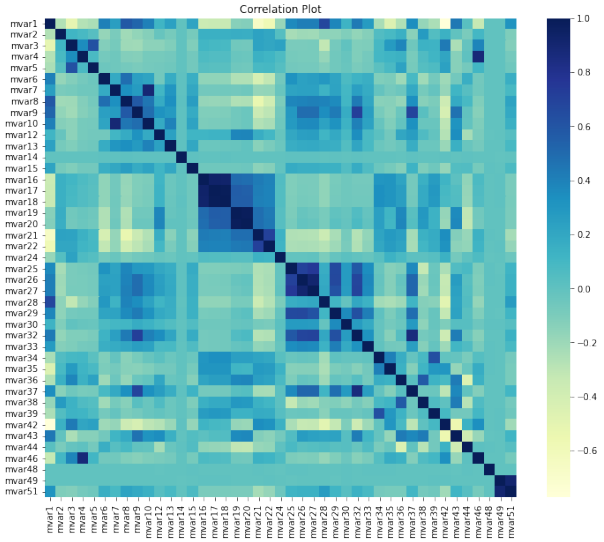

Fig. 1. Correlation plot of various features

B. Feature Selection

As mentioned in the previous subsection, it looks like there are quite a few missing values. While we normally impute the missing values in a dataframe, it statistically does not make sense to impute the values in columns which are missing more than 50% of their values. So, based on this and figure 2 , we drop the columns 'mvar11', 'mvar23', 'mvar31', 'mvar40', 'mvar41', and 'mvar45'.

The next step is to select the features that are important for the model. For this, we tried 3 different routines - using the correlation between features, using the VIF (variance inflation factor) values of various features, and using the importance of the features learnt by the various boosting algorithms. Each of these methods have been briefly explained below.

1) **Correlation Based:** Here, we use the correlation between various features to drop the redundant ones. As the calculation of correlation does not require the missing values to be filled, we leave them as is, and we drop the

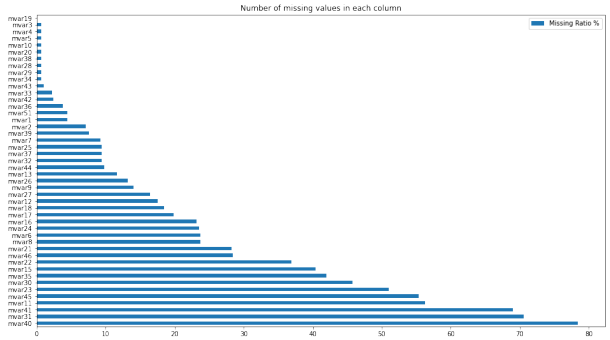

Fig. 2. The number of missing values in each column, in ascending order

features with correlations over a particular threshold. In our case, we have chosen that threshold to be 0.6.

2) **VIF Based:** In this method, we compute the VIF values of the various features after filling the NaNs with an arbitrarily chosen value, -1 in our case. After doing this, we remove the features with a relatively high VIF value. For our experiment, we removed the features with a VIF value of 7 or above.
3) **Using the learnt feature importance:** Finally, we follow a method where we implement a boosting algorithm with backward step-wise feature selection. In this, we fit a boosting model on the given data repeatedly, with a decreasing number of features each time. We initially start with all the features, and on each iteration remove the least useful features iteratively until we attain the best validation accuracy.

C. Observations of Feature Selection

What we noticed on doing the various feature selection techniques mentioned above was pretty interesting and counter-intuitive. We noticed that on dropping any features, by either method, the validation accuracy and F1 went down, and it did seem like the model was performing best.

Although, the thing to note here is that the validation accuracy is very similar even on removing certain features. For example, we got a validation accuracy of 0.7223424071 on using all the features, but got a validation accuracy of 0.7178155043 simply by training on the best 20 features, as determined by the method of backward step-wise feature selection. This goes to show that very similar performances can be obtained with a marginally very small number of parameters, however, as our primary target was to achieve a higher score on the leaderboard – we went with the model trained on all the features.

III. Exploratory Data Analysis

A. Class Imbalance:

One of the first things that is apparent is the class imbalance in the dataset. We will need to account for this imbalance when we are training, and that is something we have done. This has been visualized in figure 3.

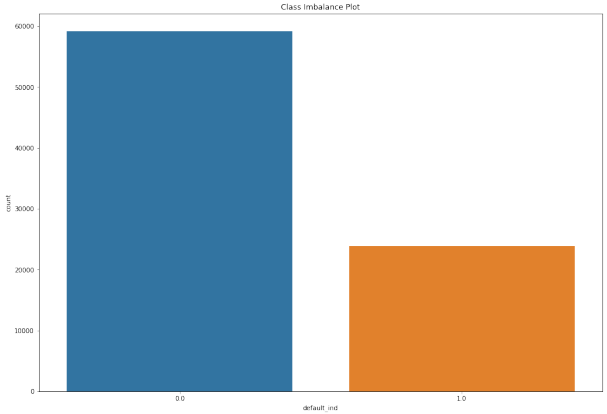

Fig. 3. The class imbalance in the dataset

B. The categorical variables:

Here, we also attempt to visualize both the variables we have taken to be categorical: 'mvar47' and 'mvar50'. Based

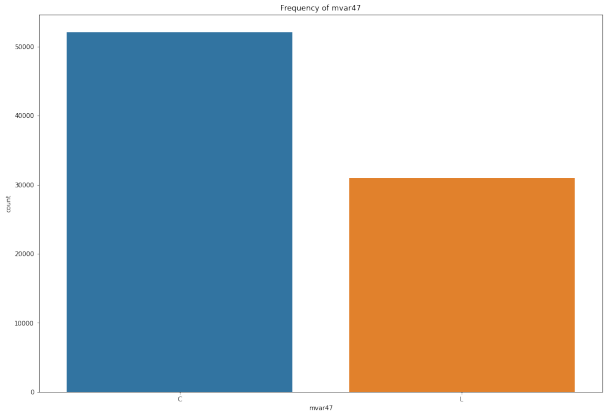

Fig. 4. Frequency of 'mvar47'

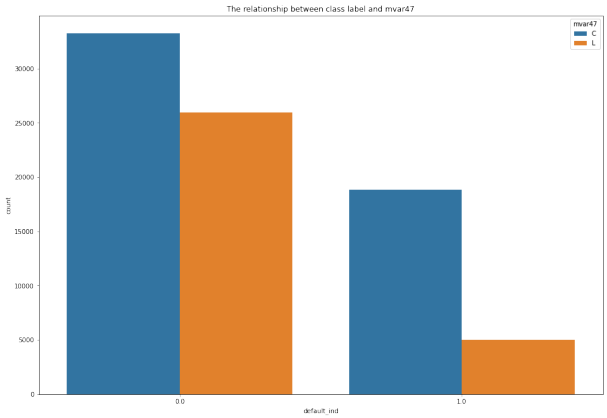

Fig. 5. Effect of 'mvar47' on class label

on the figures 4, 5, 6, and 7, we can conclude that the categorical variables have significant impact on the output class, and should be included in the model we train.

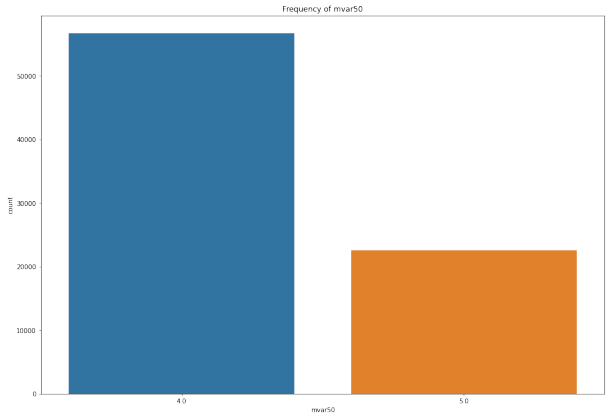

Fig. 6. Frequency of 'mvar50'

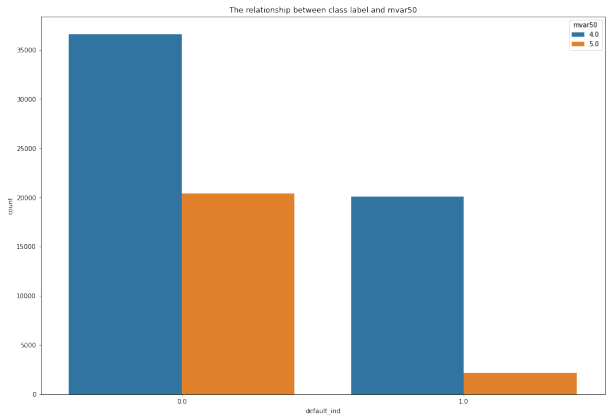

Fig. 7. Effect of 'mvar50' on class label

IV. Models

For this task, we have used a bunch of different gradient-boosting algorithms, and we have also trained a small Neural Network. The various models we have used are:

A. Catboost

The basic idea behind CatBoost is similar to other gradient boosting algorithms: it creates a model that makes predictions based on an ensemble of weaker models. In CatBoost, the models are decision trees. To train a decision tree, the algorithm begins by selecting a root node and then splits the data into two groups based on the value of a feature. The process is then repeated recursively on each group until the tree is fully grown. To train the model, CatBoost uses a variant of the gradient descent algorithm to minimize the loss function. It does this by adjusting the weights of the trees in the ensemble based on the gradient of the loss function.
One key feature of CatBoost is that it can handle categorical data natively, without the need to perform one-hot encoding or other preprocessing. It can also automatically identify and use categorical features, making it easier to work with data that has a mix of continuous and categorical features. Another advantage of CatBoost is that it includes a number of regularization techniques to help prevent overfitting, including bagging, feature importance calculation, and the ability to specify a prior for model coefficients. It also includes options for parallelization, which can help speed up training on larger datasets.
In the growing procedure of the decision trees, CatBoost does not follow similar gradient-boosting models. Instead, CatBoost grows oblivious trees, which means that the trees are grown by imposing the rule that all nodes at the same level, test the same predictor with the same condition, and hence an index of a leaf can be calculated with bitwise operations. The oblivious tree procedure allows for a simple fitting scheme and efficiency on CPUs, while the tree structure operates as a regularization to find an optimal solution and avoid overfitting. Overall, CatBoost is a powerful and flexible library for gradient boosting that can be a good choice for many machine learning tasks, particularly when working with categorical data. It can be more efficient and accurate than traditional gradient boosting methods, particularly when working with large datasets or datasets with a large number of features.

B. Extreme Gradient Boosting (XGBoost)

Like other gradient boosting algorithms, XGBoost works by creating a model that makes predictions based on an ensemble of weaker models. In XGBoost, the models are decision trees. To train a decision tree, the algorithm begins by selecting a root node and then splits the data into two groups based on the value of a feature. The process is then repeated recursively on each group until the tree is fully grown. To train the model, XGBoost uses a variant of the gradient descent algorithm to minimize the loss function. It does this by adjusting the weights of the trees in the ensemble based on the gradient of the loss function.
One key feature of XGBoost is that it is designed to be highly scalable and efficient, making it a good choice for working with large datasets. It also includes a number of regularization techniques to help prevent overfitting, such as column subsampling and shrinkage (also known as "learning rate"). Another advantage of XGBoost is that it includes a number of diagnostic tools, such as the ability to visualize the feature importance and training progress, which can be helpful for understanding how the model is making predictions and identifying potential issues during model development.

C. Histogram-based Gradient Boosting (HistGB)

Histogram-based gradient boosting is a variant of gradient boosting that uses histograms to approximate the distribution of the features in the training data. This can make the training process more efficient, particularly when working with large datasets. In classical gradient boosting, the decision trees are trained using a technique called "exact split finding," which involves searching for the optimal split point for each feature at each tree node. This can be computationally intensive, particularly when working with large datasets or datasets with a large number of features.

In contrast, histogram-based gradient boosting builds a histogram of the values for each feature and then uses this histogram to approximate the distribution of the feature. This can significantly reduce the number of calculations required to train the model, as it allows the algorithm to skip the exact split finding step and instead use the histogram to approximate the optimal split point.

There are a number of advantages to using histogram-based gradient boosting over classical gradient boosting. First, it can be significantly faster, particularly when working with large datasets or datasets with a large number of features. Second, it can be more accurate, as the histogram-based approach can better capture the distribution of the features in the training data. Finally, it can be more memory efficient, as it requires less memory to store the histograms than the full dataset.

Overall, histogram-based gradient boosting can be a good choice when working with large datasets or datasets with a large number of features, as it can provide a more efficient and accurate training process. As discussed earlier, in histogram-based gradient boosting, instead of using a linear model, such as a decision tree, to fit the residual errors, a histogram is used to approximate the distribution of the errors. This allows the model to learn more complex non-linear relationships between the features and the target variable.

Mathematically, gradient boosting works by iteratively fitting a sequence of simple models, such as decision trees, to the residual errors of the previous model. . Let's say we have a training dataset with n examples and a target variable y that we want to predict. At each iteration, the gradient boosting algorithm fits a model to the residual errors of the previous model such that the overall prediction of the ensemble model can be written as:

$$f(x) = f_0(x) + f_1(x) + ... + f_n(x) \tag{1}$$

where $f_0(x)$ is the initial prediction, and $f_1(x)$ to $f_n(x)$ are the successive models trained to fit the residual errors of the previous models.

In histogram-based gradient boosting, the models are histograms, rather than linear models, and the residual errors are approximated using the histogram bins. This allows the model to capture more complex non-linear relationships between the features and the target variable. To train the histogram-based gradient boosting model, we first need to determine the number of bins to use in the histogram. This can be done using a validation dataset, where the number of bins is selected such that it minimizes the loss function, such as mean squared error. This is done internally by the HistGradientBoostingClassifier. Once the number of bins has been determined, the histogram-based gradient boosting model can be trained using gradient descent, where the parameters of the histogram, such as the bin edges, are updated in each iteration to minimize the loss function. This process continues until the model reaches convergence, or until a maximum number of iterations is reached.

D. Multi-Layered Perceptron (MLP)

A multi-layered perceptron (MLP) is a feed-forward neural network consisting of at least three layers of nodes - an input layer, a hidden layer, and a final output layer. An MLP can consist of multiple layers of hidden layers. The input layer consists of the input to the network fed in, either raw or processed. Every other node in the layers output non-linear transformations of information received from the connections from the previous layers. The nonlinearity is achieved with the help of an activation function. Some common activation functions are as follows:

- Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$
- Tanh: $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- ReLU: $f(x) = max(0, x)$

MLPs are fully connected. This means that each node in one layer connects with a certain weight w_{ij} to every node in the following layer. The feed-forward propagation is the process by which each layer computes a set of node outputs and passes the information on to the next layer till the output layer. In a regression problem, the output is continuous-valued, while in a classification problem, it is generally a set of classwise prediction probabilities or simply the predicted class itself. Weight prediction is done in a supervised learning fashion. Since weights are initialized randomly, the starting iterations of the network will give wildly diverging predictions from the true labels. To make the weights converge to their optimal values, the total errors of the predictions are computed in each iteration and fed back into the nodes to aid in adjusting the weights accordingly. This is known as backpropagation.

Each weight is updated after a portion of the data is looked at and the errors computed, in a technique called stochastic gradient descent, one of the simplest and most popular optimization algorithms. If the error function with respect to the weights can be visualized as a valley, then the gradient descent algorithm can be thought of as taking small steps down the gradient till the optima (minima) is reached. To avoid too large, or too small steps, the gradient is scaled by a factor called the learning rate. This way, the final MLP learns to approximate a complex function that is not linearly separable.

On training and tuning these various models on our given data, we got leaderboard scores ranging from over 54% to 60%. However, one interesting thing that we noticed was that there was not a lot of overlap between the various models' predictions. This meant that different models were working well on different parts of the data. This led us to make a bagging-based ensemble model, where we considered each of the individual models' predictions after training them independently and decided on the final prediction based on a simple voting procedure.

V. Experimental Setting and Training

Going into the various models we have used, one thing to be noted is that the class imbalance in the dataset had to

be explicitly handled for all the predictors. This was done in the boosting models by setting a 'class_weights', and was handled in the Neural Network by modelling it into the loss function. We obtained these weights by inversely weighting the frequency of each class

A. Catboost

- As catboost does not require imputation of empty values, these were left as is - for the model to learn from.
- SAs the categorical variables only had 2 classes, they were directly encoded with either 0s or 1s. No scaling was used as Catboost does not require any scaling for performance.
- For our catboost model, we performed hyperparameter tuning using GridSearchCV.
- In the case of Catboost, we observed that the best set of hyperparameters we obtained were also the default hyperparameters. These hyperparameters can be looked up from the catboost documentation. Regardless, some of them have been listed in the table below.

Hyperparameter	Best Value
Max. Number of iterations	1000
Depth	6
Loss Function	logloss
L2 leaf regularization	50
Leaf Estimation Iterations	10

B. XGBoost

- As XGBoost requires the empty values in the data to be imputed, we tried various imputation schemes. Initially, we went with class-wise mode and mean/median imputations, however, we found that it did not perform as well. Later, on further reading, we found out that trees work better when they are given the empty data to work on too, so we achieved that by filling the empty entries by a sufficiently large negative number (-999 in our case).
- StandardScaler from the sklearn library was used to scale all the numerical features using their mean and standard deviation. As the categorical variables only had 2 classes, they were directly encoded with either 0s or 1s.
- In XGBoost also, we performed hyperparameter tuning using GridSearchCV.
- The optimal parameters we obtained have been listed in the table below.

Hyperparameter	Best Value
Subsample	0.5
Column sample by tree	1
Maximum depth	6
Minimum weight of child node	15
Learning rate	0.1

C. HistGB

- For Histogram-based gradient boosting, the categorical variables 'mvar47' and 'mvar50' were one-hot encoded for both training and test data while preprocessing. StandardScaler from the sklearn library was used to scale all the numerical features using their mean and standard deviation.
- We performed Hyperparameter tuning using GridSearchCV here as well.
- The best hyperparameters obtained from the model(as given below) were then used to train the Histogram-based gradient boosting classifier on the training dataset, and predictions were obtained for the test set.

Hyperparameter	Best Value
Max. Number of iterations	100
Learning rate	0.2
L2 regularization penalty	50

D. Neural Network

To prepare the data for the neural network, the following data preprocessing was done:

- Stray values were replaced by NaN throughout the dataset. Columns with more than 30% missing values were dropped. The application key column was dropped as well as it adds no predictive information.
- The missing values in the continuous-valued columns in the training set were imputed using the median of the columns. The missing values for the same in the test set were also imputed using the median calculated from the train set.
- The missing values in the categorical columns (only 'mvar47') in the training set were imputed using the mode of the columns. The missing values for the same in the test set were also imputed using the mode calculated from the train set. Note that the missing information in the test set in all the above cases is being imputed from the train set, and not the test set itself. This is done to prevent information leakage.
- Feature selection was done by dropping the features with the largest variance inflation factor (VIF) in an iterative process. The iterations are repeated and one feature is removed at each iteration until we get VIF values under a certain threshold for all features. We have considered this threshold as 10.

Architecture

The neural network architecture contained an input layer, a 64-node hidden layer followed by batch normalization, a 16-node hidden layer followed by batch normalization, and an output sigmoid layer yielding the predicted class i.e. whether an individual will default or not. The other specifics of the model were:

- Keeping in mind the major class imbalance in the data (71.25% non-defaulters), a cost-sensitive learning technique was used to train the neural network wherein the cost of misclassification is inversely proportional to the class frequency in the dataset.

- The model was trained with the binary cross entropy loss function and Adam optimizer.
- 15% of the training data was kept aside in each epoch for the purpose of validation using the total F1 score metric. After 100 epochs, the model weights corresponding to the best validation F1 score of 54.7% was saved and predictions on the test set were made.

E. Bagging Model

As we had 4 models (an even number, could result in ties), we tried 3 variants of bagged models.

- Variant 1: As HistGB performed the worst, we discarded its predictions and considered the other three models' votes.
- Variant 2: We considered all 4 models' predictions, and whenever we had a tie in the voting, we went with the prediction by the Catboost model as it was our best-performing one.
- Variant 3: A voting classifier between Catboost and HistGB to see which one dominated the improvement in performance from Variant 1 to Variant 2.

VI. Results, Observations, and Conclusion

Model	**Leaderboard**
Catboost	60.68%
XGBoost	58.05%
Neural Network	57.34%
HistGB	56.14%
Bagging NN, Catboost, & XGBoost	60.71%
Bagging NN, Catboost, XGBoost & HistGB	**60.79%**
Voting classifier between Catboost and HistGB	60.69%

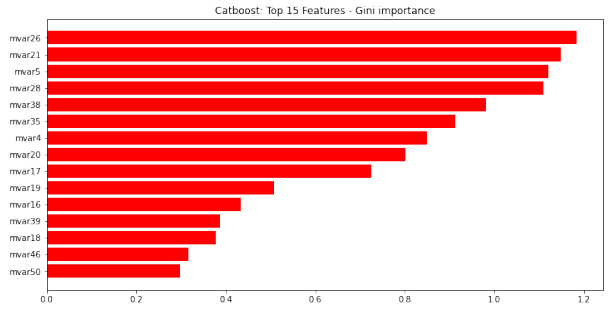

Fig. 8. Most useful features - as given by Catboost, using the gini score

- It is to be noted that any sort of bagging has improved results, reaffirming the fact that different models were working well on different parts of the data available to us.
- The most important features as learnt by catboost are given in figure 8. This can be used by us to gauge important on what features are the most important to make accurate predictions.
- The data is very well represented with even a small amount of features, as observed in the feature selection section.
- Further work can be done by creating new hybrid features by clubbing together existing, but similar features. We can also further exploit the fact that a small fraction of features explain the data pretty well.

To conclude, we have successfully managed to build an ensemble model with sufficient reasoning to perform the task of predicting whether a customer will default - giving a pretty good score on the unseen test data, indicative of the robustness of the model.