Project Report

---

1) Project Overview

American Express (Amex), which is the largest payment card issuer globally, launched 'American Express - Default Prediction' challenge on Kaggle. The objective is to classify the customers either as a potential non-defaulter (0) or a potential defaulter (1) by analyzing their monthly customer profile. The customer is expected to pay the due amount in 120 days after their latest statement date failing which they are considered to be a defaulter.

The challenge details can be found in below link:
https://www.kaggle.com/competitions/amex-default-prediction

The input features are anonymized and normalized and are broadly classified into below categories as described in the data page of the challenge:

- D_* = Delinquency variables
- S_* = Spend variables
- P_* = Payment variables
- B_* = Balance variables
- R_* = Risk variables

The aim is to create a model which can identify the defaulters (minority class) with a high probability while limiting the misclassification of non-defaulters.

1.1) Problem Statement

Create a machine learning model that can aid AMEX to correctly predict (with a high probability) as to whether their customers (new / existing) are expected to 'default' or 'not default' in their payments.

1.2) Challenges

There are 3 main challenges in this project are as follows:

1. The biggest challenge is that the dataset is highly imbalanced and the minority class of defaulters comprise of less that 25% of the records in training set i.e. 1377869 out of 5531451 total records.
2. All features are anonymized and normalized and only the broad categories they belong to is known. So, domain knowledge isn't of much use as actual feature names are unknown.
3. The dataset is huge both in terms of the number of records and the number of features for each record. Hence, extensive EDA is required to extract and feed only relevant information to the model

1.3) Evaluation Metrics

The data after EDA and preprocessing is partitioned such that defaulters and non-defaulters are equally represented in the training set. Also, given that it is essential to identify the defaulters correctly while limiting the misclassification of non-defaulters, ROC_AUC is chosen as a suitable metric for this binary classification problem.

2) Exploratory Data Analysis (EDA)

The dataset for the problem is huge and contains nearly 5.53 million records and 190 features in the training set. Hence, it is important to explore the data well and remove unnecessary features while retaining maximum information.

Also, the target values are highly imbalanced containing approximately 75% of defaulters and only 25% of non-defaulters.

To easily load the complete training data used for EDA, a lightweight version of the dataset in parquet format is used from the below link: https://www.kaggle.com/datasets/raddar/amex-data-integer-dtypes-parquet-format

Before begining the analysis, the target values is assigned to the training set from the train_labels.csv increasing the final columns count to 191. A customer identified as defaulter in the train_labels.csv is assumed to have a defaulter's behavior for every row of the customer in the Training dataset (depicting different credit card statements for the customer).

Given the enormous size of the dataset, it is impossible to analyze the data completely with Pandas Profiler. To counter this issue, Pandas profiler is executed with minimal parameter set to True to gain some initial insights on the feature columns. The profiler generated alerts for 169 out of 191 columns. The alerts are raised for one of the following reasons:

1. Too many zeros in the column
2. Highly Skewed data in the column
3. Missing large number of values in the column
4. High cardinality i.e. too many distinct values in the column

So, EDA is performed in two stages for alerted columns and non-alerted columns each.

2.1) EDA for columns alerted by Pandas Profiler

A simple approch to reduce the data size can be to drop all columns alerted by Pandas for the above issue. However, instead of directly dropping all the columns alerted under these categories, the columns under each of these categories is evaluated further as described below:

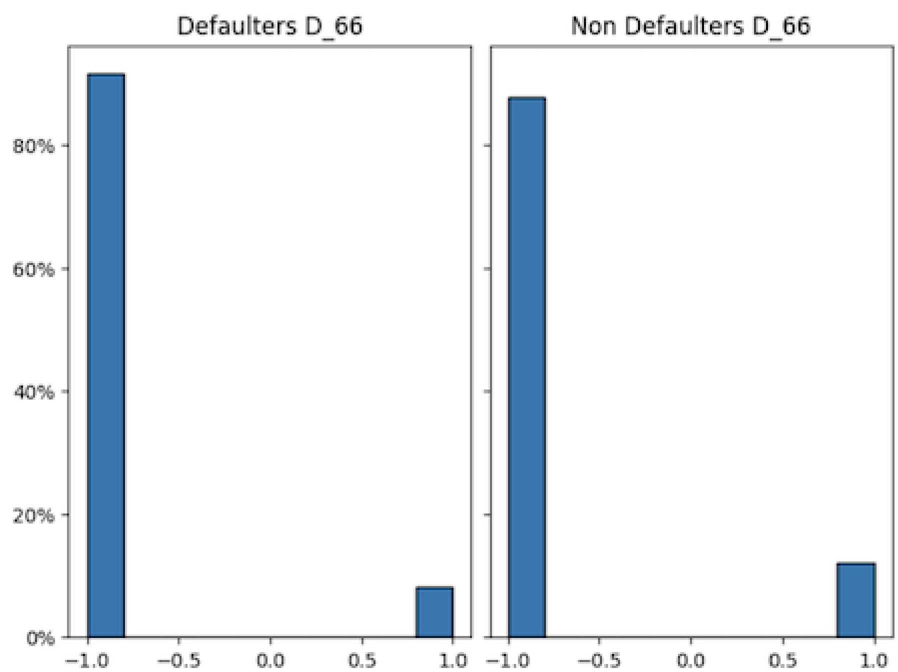
1. Too many zeros

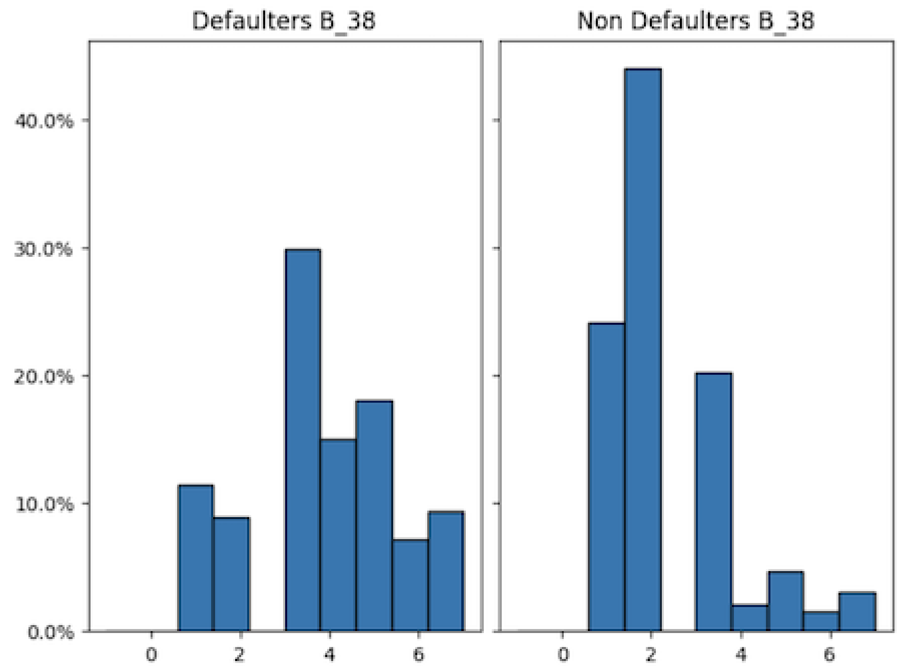
Analysis for categorical columns

While too many zeros in columns with continous variable may not be useful, it can be one of an important category for a categorical variable. Hence, all categorical columns are studied further to identify the ones that can improve the model's predictive performance.

To facilitate this study, multiple histograms are used. Firstly, a basic histogram is created for each categorical column to get the frequency count of values and to ensure zero is an actual category as anticipated. It was found that most categorical columns had 0 as a valid category.

Finally, separate relative frequency histograms for defaulters and non-defaulters are created for each categorical variable to see if they exihit similar or different behavior for each group. For instance, consider the relative frequency histograms generated for the categories D_66 and B_38 below:





As can be seen, D_66 doesn't seem to behave much differently for each group of customers whereas B_38 shows different trend for each.
From above analysis, categorical columns: B_30, B_38, D_64, D_117 and, D_120 that exhibit different trends for each group are retained.

Analysis for columns with only 10% or lesser zeros

Since the training dataset contains almost 5.5 million records when in a column zeros are present for even 1% of data the record number is greater than the threshold for pandas profiler and the column is flagged as having too many zeros.

So, columns with only 10% or lesser values in data as zeros i.e. ‘D_82', 'D_106', 'D_122', 'D_126', 'D_135', 'D_137', 'D_138’, 'R_26' is also analyzed with relative frequency histograms and D_82 and D_122 are identified as useful.

Target

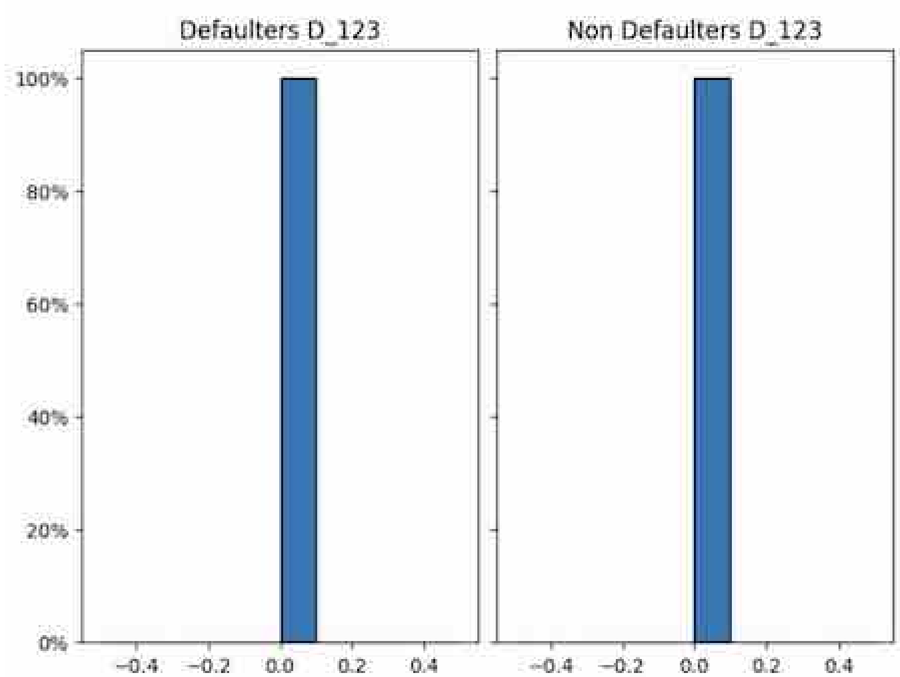
Since defaulters with target value zero are a majority class of the training dataset, target column is expected to have many zeros and must be retained for training.

Summary Missing Zeros

All columns alerted by Pandas profile for too many zeros except these B_30, B_38, D_64, D_117, D_120, D_82, D_122 and target are identified to be safe to drop without much information loss.

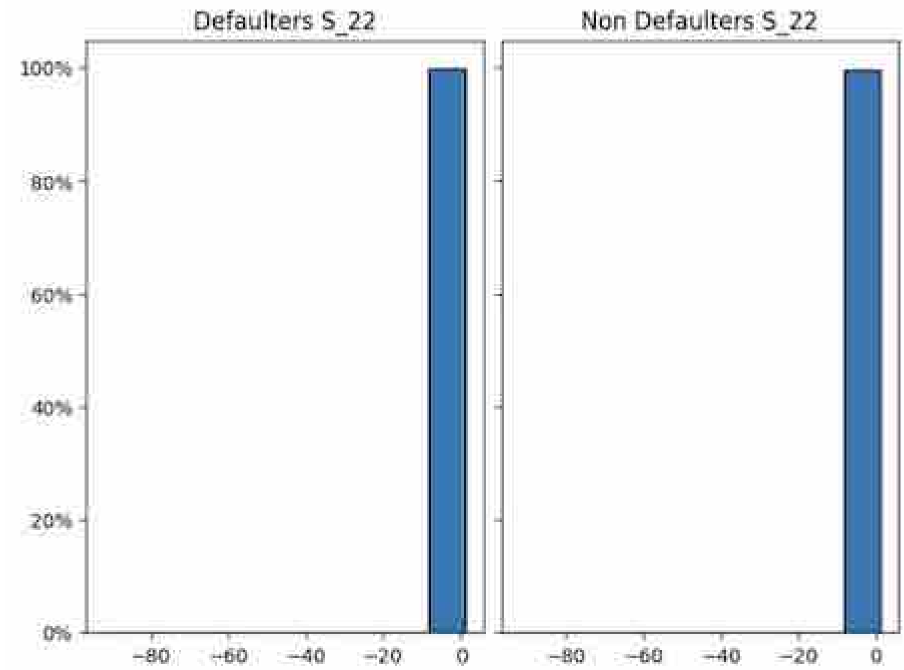
2. Highly skewed

The highly skewed columns are analyzed to see if the distribution is useful when the outliers are removed. To facilitate this, all values above 99 percentile and below 1 percentile are removed from the distribution and features like D_123 which display no change in distribution are removed.

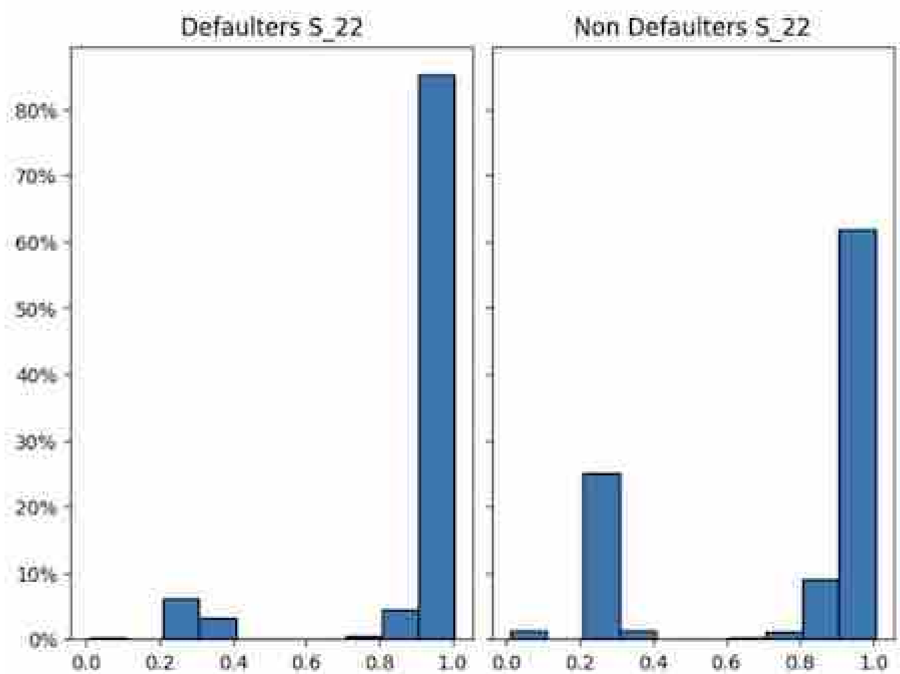


D_123 without outliers

However, the distribution of some features like S_22 improve drastically after the removal of outliers as seen below:



S_22 with Outliers



S_22 without Outliers

Summary Highly Skewed Columns

‘D_61', 'S_22', 'S_23', 'S_24' and 'B_40’ shows improved distribution after removal of outliers and must be retained.

3. Large number of Missing values

Columns with relatively less missing data i.e. less than 10% are explored to discover interesting distributions (if any) with a relative frequency Histogram as above.

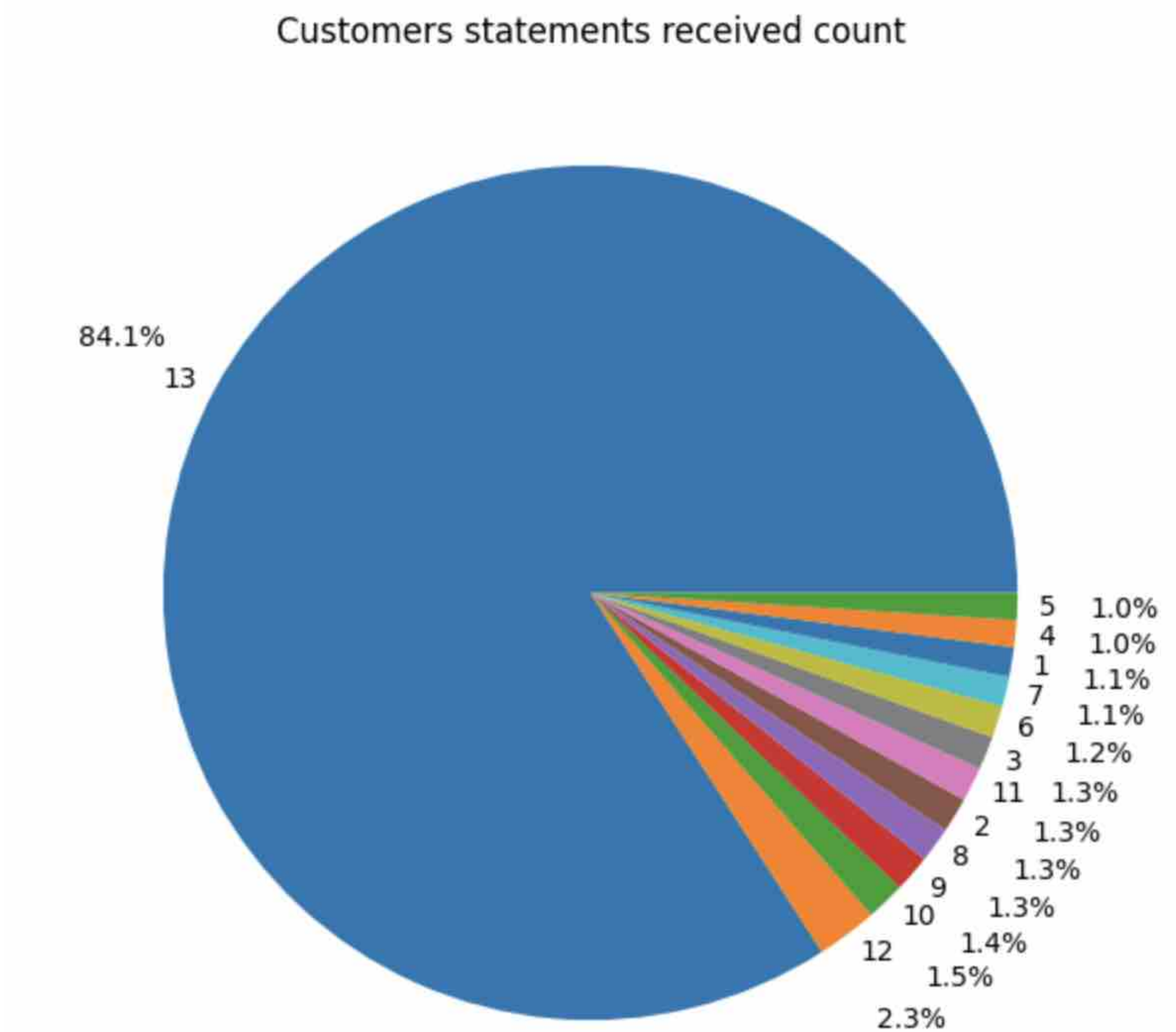
Summary Missing values

Columns ‘P_3’, 'D_55', 'D_104', 'R_27', 'D_115', 'D_118', 'D_119', 'D_121' and, 'D_128’ have good information and will be retained.

4. High Cardinality

Columns with too many distinct values can prove to be useless. However, the columns alerted in this case i.e. 'customer_ID' and 'S_2' are expected to have many distinct values.
'customer_ID' is unique identifier for customers in the dataset.
'S_2' represents the customer's credit card statement date. And even though this might be expected to be limited due to the expected repetition of month and year values, not all customers receive the statements on same day of the month.

Standalone these columns don't provide a lot of information. However, as seen below nearly 84% of customers have received 13 credit card statements. Thus, some pattern can be discovered by the model for customer and month combination. Thus, the model can benefit from engineering this new feature.



Customers statement count

Summary for High Cardinality

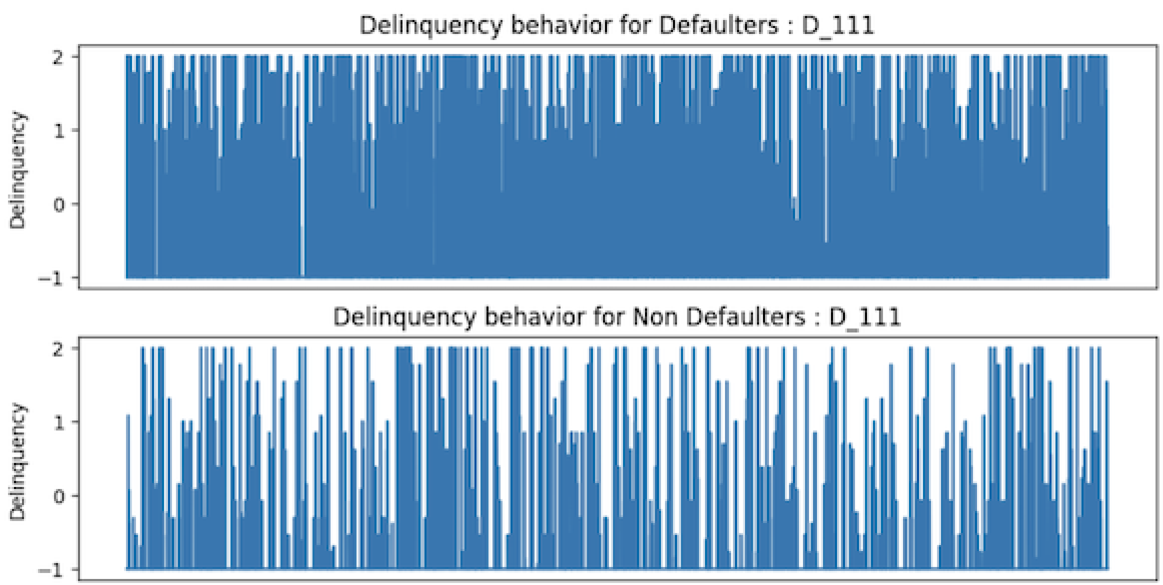
'customer_ID' and 'S_2' are to be retained till engineering a new, useful feature.

2.2) EDA for columns not alerted by Pandas Profiler

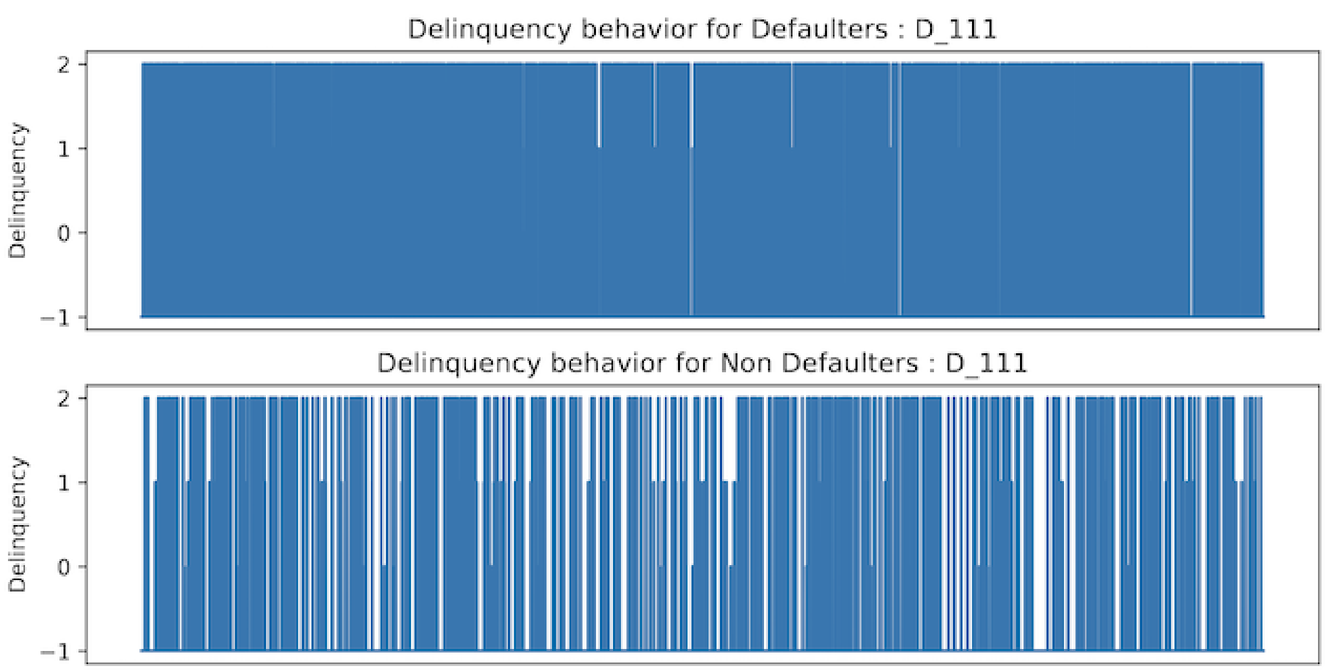
A simple approach can be to include all the columns not alerted by Pandas profiler. However, given the huge dataset size, it is better to analyze the columns and determine if they really add any value.

To study this, columns were categorized into categories they belonged to i.e. either Payments, Risk, Spend, Delinquency or Balance.

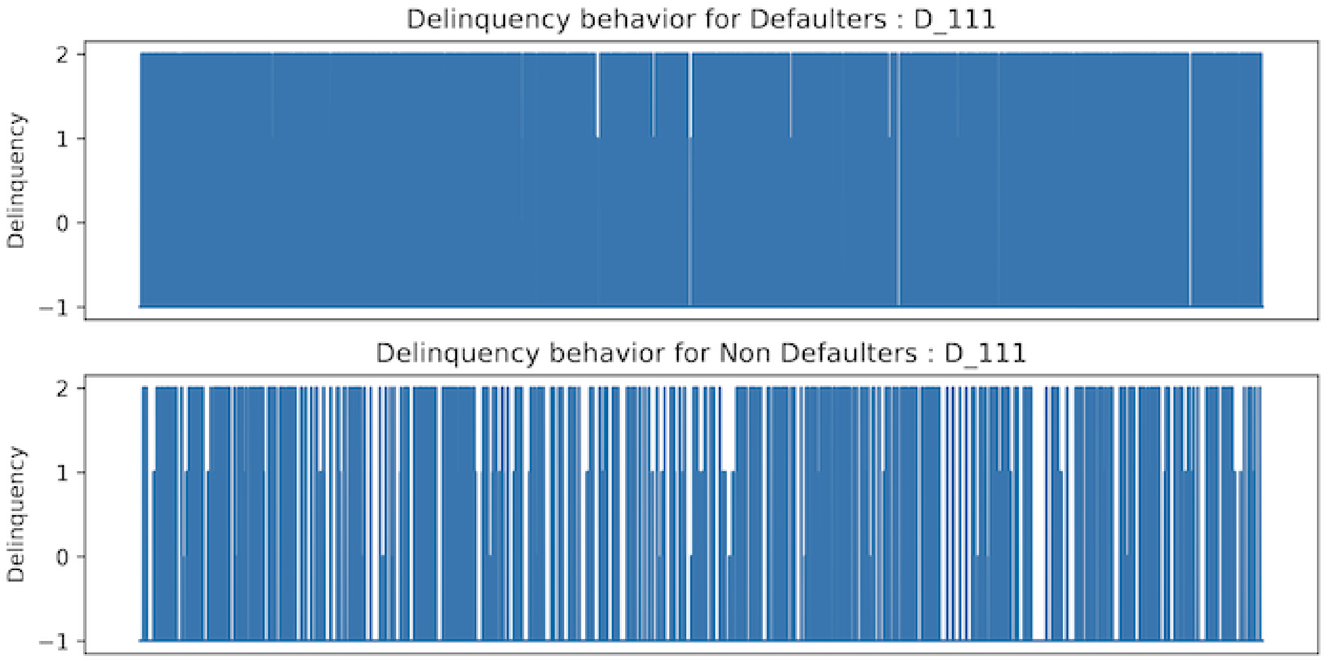
Then, for a group of 100K randomly selected defaulters and non-defaulters each, column in each category was visualized as a line graph for the mean, min and max values of the column to see if they help understand the data better.



D_111 behavior by mean of column values



D_111 behavior by min of column values



D_111 behavior by max of column values

Summary

Columns like D_111 which shows distinct behavior for defaulters vs non-defaulter and can help the model classify the two groups better must be retained.

As a contrast, ‘R_9’, 'B_1', 'B_2', 'B_7', 'B_18', 'B_25', 'B_28', 'B_37', 'P_2', 'D_47 ', 'D_52 ', 'D_60 ', 'D_66 ', 'D_68', 'D_102 ', 'D_112 ', 'D_124 ', 'D_144', and 'S_11’ are identified to be less useful.

3) Data Preprocess

Based on the EDA and to address the need for smaller Training, Validation and Test datasets, the data is preprocessed as below:

1. All columns alerted by Pandas profiler for closer examination except the following are dropped:

```
['customer_ID', 'S_2', 'P_3', 'D_55', 'D_104', 'R_27', 'D_115',
'D_118', 'D_119', 'D_121', 'D_128', 'D_61', 'S_22', 'S_23',
```

```
'S_24', 'B_40', 'B_30', 'B_38', 'D_117', 'D_120', 'D_64',
'D_68', 'D_82', 'D_122', 'target']
```

2. The following columns not alerted by Pandas profiler but not found useful either are dropped:

```
['R_9', 'B_1', 'B_2', 'B_7', 'B_18', 'B_25', 'B_28', 'B_37',
'P_2', 'D_47', 'D_52', 'D_60', 'D_66', 'D_68', 'D_102', 'D_112',
'D_124', 'D_144', 'S_11']
```

3. Outliers from the retained skewed columns are replaced by setting all the low end outliers to value at 1 percentile and setting all the high end outliers to value at 99th percentile value of the column. This process in also called Winsorizing. The columns are reexamined to ensure a good distribution is noticed after Winsorizing the below retained Skewed columns as expected:

```
['D_61', 'S_22', 'S_23', 'S_24', 'B_40']
```

4. Feature engineering is performed to create a new column CID_month. CID_month is derived by concatenating the values in column 'customer_ID' and the month value extracted from column 'S_2'. This concatenated value is then hashed and normalized. Thus, rows with same customer_ID and month in S_2 are expected to have the same value in the new column: 'CID_month'.
5. Data from the above modified dataset is then sampled such that the training set has 91000 records with 45500 records belonging to defaulters and the remaining 45500 belonging to non-defaulters. Maximum rows for a customer available in the original dataset is tried to be maintained and hence, the data for training set is not randomly sampled. Out of the remaining 5.44 million records, 20K records are chosen randomly and then 14K are assigned to Validation set and 6K are assigned to a test set. It is ensured that both validation and test sets have good proportions of defaulters and non-defaulters.

4) Benchmark model and metrics

ROCAUC is chosen as an evaluation metric. As a rule of thumb from Hosmer and Lemeshow in [_Applied Logistic Regression] (https://onlinelibrary.wiley.com/doi/book/10.1002/9781118548387), the following table can be used to understand how good the score is:

- **0.5** = No discrimination
- **0.5-0.7** = Poor discrimination
- **0.7-0.8** = Acceptable discrimination
- **0.8-0.9**= Excellent discrimination
- **>0.9** = Outstanding discrimination
  So, 0.7 or above can be used as a benchmark score for the binary classifier.

However, the score might vary a bit based on the dataset. So, to find a suitable benchmark score for this challenge's dataset, a Logistic Regression binary classification model is created.

Since Logistic Regression cannot be built using columns with missing data, a Simple imputer is used to replace the missing values in each column with the mean of that column.

The Logistic Regression model is built using the following configurations: penalty='l2', solver='sag', random_state=0, class_weight={0:18, 1:180}, max_iter=800.

The trained model produces decent scores on Test dataset as seen below indicating that the EDA is performed well:

```
Accuracy Score: 0.7928333333333333
Confusion Matrix:
[[3376 1124]
 [ 119 1381]]
Area Under Curve: 0.8354444444444443
```

```
Recall score: 0.9206666666666666
F1 score: 0.6896379525593009
```

The ROC_AUC score particularly is excellent for the benchmark model i.e. '0.835' as seen above and is set as the new threshold for comparing performances of solutions obtained using AutoGluon's Tabular predictor.

Benchmark Result = '0.835'

5) AutoGluon model Training

AutoGluon's TabularPredictor is used to see if a better model can be arrived at for predicting whether a customer is potentially a defaulter.

5.1) Preparation

The following link is referred to understand how to train an AutoGluon model in AWS Sagemaker such that it can be later deployed in Sagemaker: https://auto.gluon.ai/stable/tutorials/cloud_fit_deploy/cloud-aws-sagemaker-training.html

The wrapper classes used in the above guide are provided within python scripts in the below link and are reused for the purpose of this project: https://github.com/aws/amazon-sagemaker-examples/tree/main/advanced_functionality/autogluon-tabular-containers

Python scripts imported into the project are:

1. ag_model.py
2. deserializers.py
3. sagemaker_utils.py
4. serializers.py

5.2) Configuration

An AutoGluonSagemakerEstimator defined in the inference script ag_model.py is used to create a training job. A single instance of type 'ml.m5.2xlarge' is used for training.

5.3) Parameters selected

The model is configured as a binary classification problem since it needs to identify if the potential customer is either a Defaulter (1) or a Non-Defaulter (2).

Since, the goal is to try to build a machine learning model that captures the potential defaulters with high probability while maintaining a low false positive rate i.e. avoiding misrepresenting a good customer as a potential defaulter, ROC_AUC was selected as a suitable evaluation metric for this problem. As ROC_AUC can give misleading results for a highly imbalanced dataset, a balanced training set is created for the problem with equal number of records for defaulters and non-defaulters.

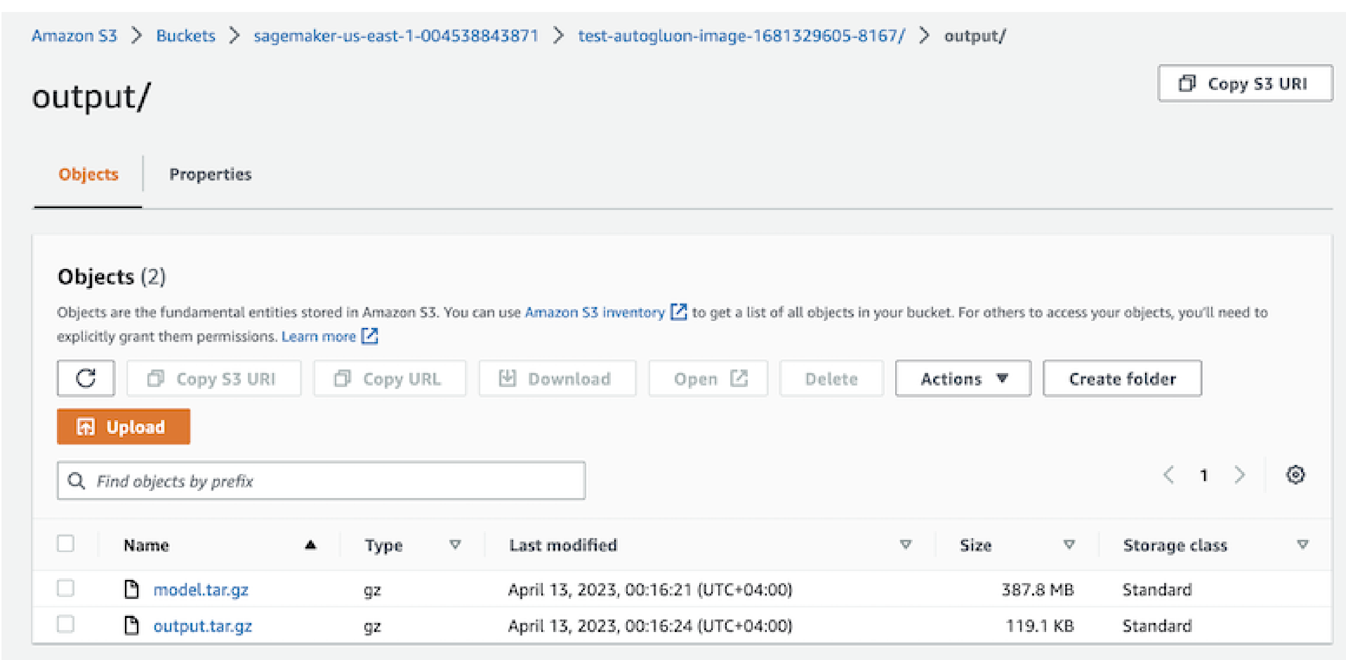
While training the model the presets is set to 'best_quality' to get the most accurate overall predictor. Also, to avoid overrunning the AWS credit during training a time limit of 600 seconds is specified.

A separate validation dataset is provided for evaluating the model performance during training to ensure that the model does not randomly separate records from the training set. 'use_bag_holdout' is set to True and 'tuning_data' is set to validation dataset so that the model uses the provided validation data as the hold out data to score models and determine weighted ensemble weights.

Complete configuration in YAML format used for Predictor construction and model training is as seen below:

```
# AutoGluon Predictor constructor arguments

# - see
https://github.com/autogluon/autogluon/blob/v0.5.2/tabular/src/a
```

```
utogluon/tabular/predictor/predictor.py#L56-L181

ag_predictor_args:
        eval_metric: roc_auc
        label: target
        problem_type: binary
        learner_kwargs: {ignored_columns:["customer_ID", "S_2"]}

# AutoGluon Predictor.fit arguments
# - see
https://github.com/autogluon/autogluon/blob/v0.5.2/tabular/src/a
utogluon/tabular/predictor/predictor.py#L286-L711

ag_fit_args:
        presets: best_quality
        time_limit: 600
        num_bag_folds: 2
        num_bag_sets: 1
        num_stack_levels: 0
        use_bag_holdout: true
```



Model saved after training and used for deployment

5.4) Training Results

With the preprocessed data and above configurations, AutoGluon is able to generate brilliant models with great predictive performance.

The best model identified by AutoGluon is WeightedEnsemble_L2 which gives excellent score on both test and validation sets i.e. 0.9265 and 0.9236 respectively.

```
Loaded data from: /opt/ml/input/data/test/test.csv | Columns = 47 / 47 | Rows = 6000 -> 6000
model  score_test  score_val  pred_time_test  pred_time_val    fit_time  pred_time_test_marginal  pred_time_val_marginal  fit_time_margin
al  stack_level  can_infer  fit_order
0      WeightedEnsemble_L2    0.926551   0.923608        4.107594      27.040528  196.565640                0.004968              0.0
02330           2.590427          2       True         13
1           XGBoost_BAG_L1    0.925854   0.919466        0.119128       0.807682   63.369500                0.119128              0.8
07682          63.369500          1       True         11
```

Snapshot from leaderboard

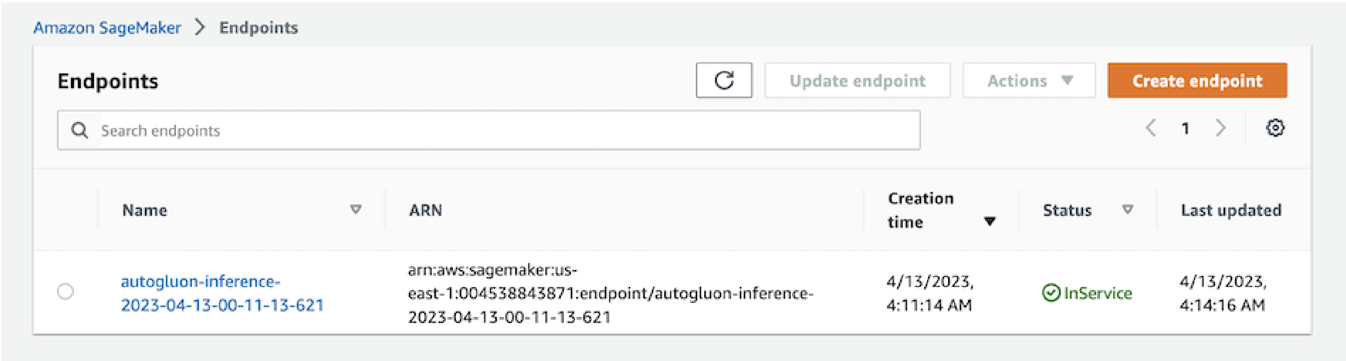
6) Deployment and Inference

Inorder to make the model available to an end-user for running predictions on a dataset, an endpoint for the model is created in AWS Sagemaker and a Lambda function that facilitates inferencing for the end-user is also created.

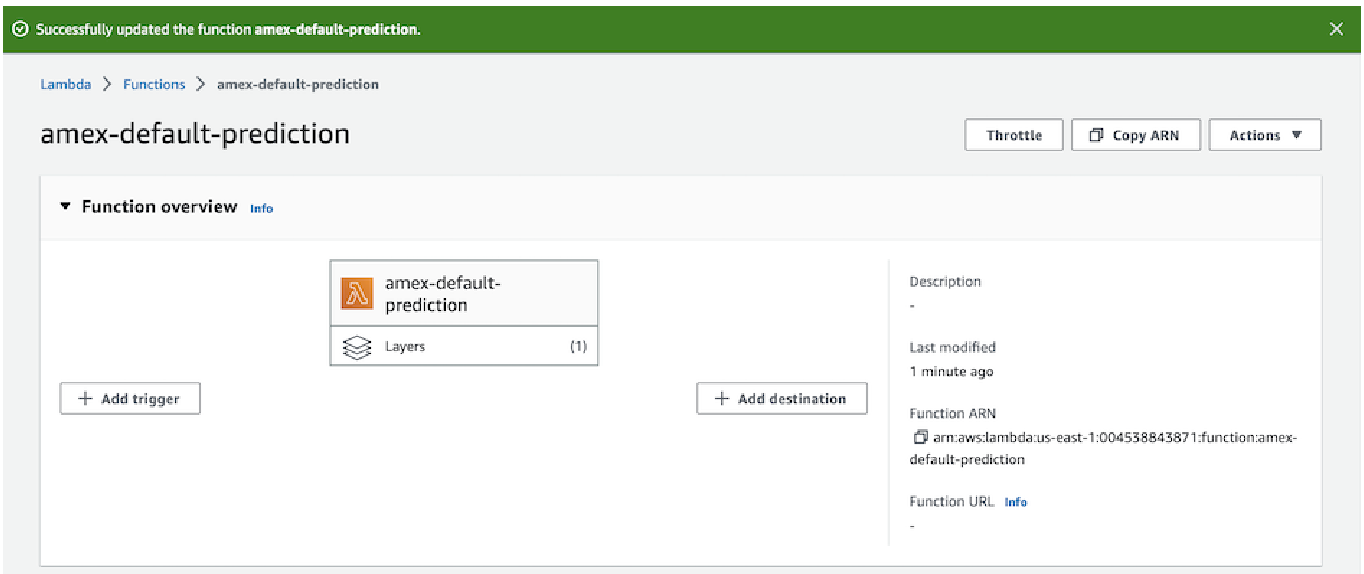
The following guide providing information on deploying an AutoGluon model in AWS Sagemaker is referenced: https://auto.gluon.ai/stable/tutorials/cloud_fit_deploy/cloud-aws-sagemaker-deployment.html

For deployment, an instance of AutoGluonNonRepackInferenceModel from ag_model.py is created and deployed. It uses the inference_script.py as an entry point and an instance of type 'ml.m5.2xlarge'. Deployment with a smaller instance type i.e. 'ml.t2.medium' was attempted but issues were encountered on trying to run inference for more than one record at a time.

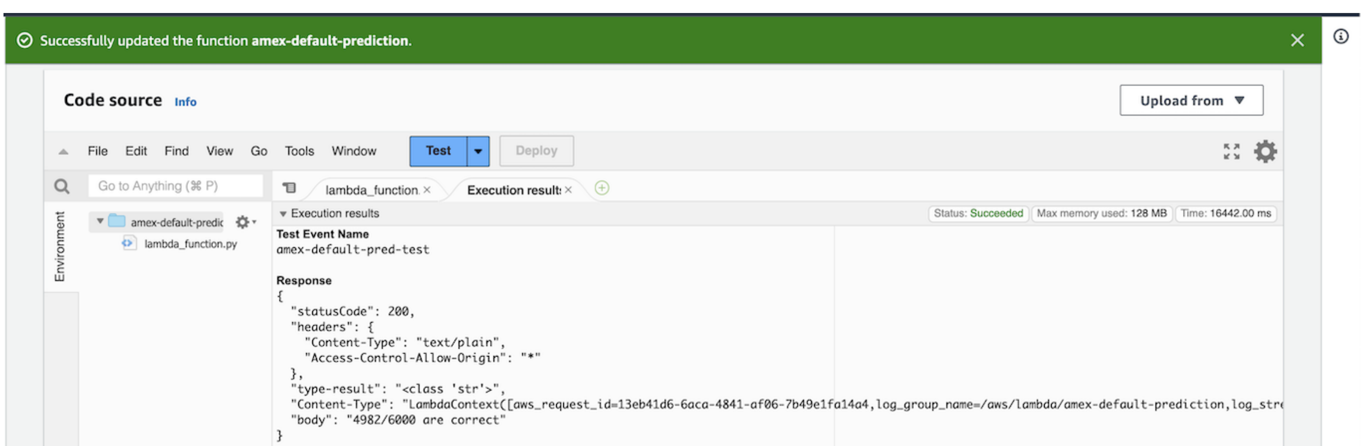
To run inference a Lambda function with name 'amex-default-prediction' is created. Running inference on the test dataset with this function produces accurate results for 4982 records out of 6000 records. The inference is evaluated to ensure that out of all the accurate predictions, defaulter cases are also predicted sufficiently.



**Endpoint deployed in Sagemaker



Defined Lambda function



Test Result

	pred	actual
0	1	1
1	1	1
2	0	0
3	1	1
4	1	1

Test Result snapshot

9) Project Refinement

Impressive results is received from the first round of AutoGluon Tabular Predictor as discussed in section 5.4. Twelve models were trained by AutoGluon in this round and all of them scored very high on both validation and test set with ROC_AUC for many above 0.9 as seen below:

```
model  score_test  score_val  pred_time_test  pred_time_val    fit_time  pred_time_test_marginal  pred_time_val_marginal  fit_time_margin
al  stack_level  can_infer  fit_order
0        WeightedEnsemble_L2    0.926551   0.923608        4.107594      27.040528  196.565640                0.004968                0.0
02330          2.590427           2       True         13
1             XGBoost_BAG_L1    0.925854   0.919466        0.119128       0.807682   63.369500                0.119128                0.8
07682         63.369500           1       True         11
2            LightGBM_BAG_L1    0.922136   0.918469        1.240256       8.401661   45.676009                1.240256                8.4
01661         45.676009           1       True          4
3          LightGBMXT_BAG_L1    0.921918   0.917791        1.912325      16.272626   53.713734                1.912325               16.2
72626         53.713734           1       True          3
4            CatBoost_BAG_L1    0.920886   0.915619        0.054096       0.252894  192.159055                0.054096                0.2
52894        192.159055           1       True          7
5  RandomForestEntr_BAG_L1    0.920549   0.919312        0.476655       4.073381   33.606960                0.476655                4.0
73381         33.606960           1       True          6
6     NeuralNetTorch_BAG_L1    0.918949   0.917232        0.162580       0.798352   10.475005                0.162580                0.7
98352         10.475005           1       True         12
7  RandomForestGini_BAG_L1    0.918880   0.917411        0.521979       4.172811   29.813052                0.521979                4.1
72811         29.813052           1       True          5
8     ExtraTreesEntr_BAG_L1    0.918226   0.917158        0.798731       4.537413    5.428264                0.798731                4.5
37413          5.428264           1       True          9
9     ExtraTreesGini_BAG_L1    0.916705   0.916439        0.783296       4.246898    5.606423                0.783296                4.2
46898          5.606423           1       True          8
10   NeuralNetFastAI_BAG_L1    0.899065   0.893578        0.205287       1.189221   74.923853                0.205287                1.1
89221         74.923853           1       True         10
11    KNeighborsUnif_BAG_L1    0.839027   0.836201        0.831545       9.364735    0.200778                0.831545                9.3
64735          0.200778           1       True          1
12    KNeighborsDist_BAG_L1    0.838427   0.836499        0.705422       9.445793    0.208828                0.705422                9.4
45793          0.208828           1       True          2
```

AutoGluon round 1 results

However, to see if the model outcome can be refined further several rounds of hyperparameter tuning are performed as discussed in below section.

9.1) Hyperparameter Tuning (Round 1):

In this round, the following parameters were added to the fit method retaining other configurations for TabularPredictor.fit() as per AutoGluon round 1:

```
'hyperparameters': 'default',
'hyperparameter_tune_kwargs': 'auto'
```

These settings essentially activate hyperparameter tuning with minimal configuration and train to build the best possible model within 600 seconds.

This increased the ROC_AUC score to '0.927513' for test set.

9.2) Hyperparameter Tuning (Round 2):

In this round, a Hyperparameter search space is provided with the following parameters to the fit method:

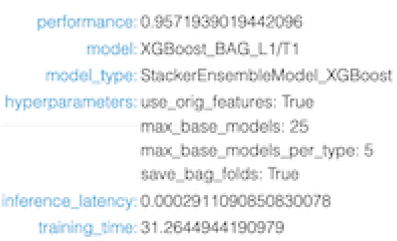
```
hyperparameters = {
        'XGB' : {
                'n_estimators': 15000,
                'learning_rate': autogluon.core.space.Real(0.01,
0.1, log=True),
                'objective': 'binary:logistic',
                'eval': 'auc',
                'booster': 'gbtree',
                'max_cat_to_onehot': autogluon.core.space.Int(2,
4),
                'use_orig_features': True,
                'max_base_models' : 25,
                'max_base_models_per_type' : 5,
                'save_bag_folds' : True
        }
}

hyperparameter_tune_kwargs = {
        'num_trials': 5,
        'scheduler' : 'local',
```

```
        'searcher': 'auto'
}
```

and the training time limit is doubled to 1200 seconds.

This setting attempted to refine the hyperparameters by providing a range of values to try from for parameters learning_rate and max_cat_to_onehot.

Training is limited to XGBoost model as it was the best model obtained in previous round of hyperparameter tuning. Also, hyperparameters identified by the best model in previous round was added as seen below:



performance:	0.9571939019442096
model:	XGBoost_BAG_L1/T1
model_type:	StackerEnsembleModel_XGBoost
hyperparameters:	use_orig_features: True max_base_models: 25 max_base_models_per_type: 5 save_bag_folds: True
inference_latency:	0.0002911090850830078
training_time:	31.2644944190979

The 'ROC_AUC' slightly reduced to '0.925471' for test results. However, the accuracy of non-defaulters prediction increased with a minor decrease in the prediction for defaulters.

9.3) Hyperparameter Tuning (Round 3):

Here the hyperparameters are simplified as below and trained for only 600 seconds:

```
hyperparameters: {
        XGB: {
                'n_estimators': 15000,
                'learning_rate': 0.03,
                'objective': 'binary:logistic',
                'eval': 'auc',
                'booster': 'gbtree'
        }
}
```

```
hyperparameter_tune_kwargs: {
        'num_trials': 5,
        'scheduler' : 'local',
        'searcher': 'auto'
}
```

'ROC_AUC' reduced further for test set to '0.924118' and accuracy remained comparable.

10) Summary

Summarizing results from all models above:

Models	ROC_AUC (Test)	ROC_AUC (Validation)	Accuracy (out of 6000)	Accuracy (Defaulters prediction out of 1500)	Accuracy (Non-Defaulters prediction out of 4500)
Benchmark model	0.835		4757	1381	3376
AutoGluon Round 1 best model	0.926551	0.923608	4982	1355	3627
HPO best model	0.927513	0.957133	4994	1354	3640
HPO 2 best model	0.925471	0.970340	5010	1345	3665
HPO 3 best model	0.924118	0.974521	5011	1346	3665

Both models highlighted in yellow perform significantly better than the benchmark model and can be considered based on the needs of Amex.

If Amex wants to capture as many potential defaulters as possible then, model from HPO round will serve better. On the other hand, if they want to limit misclassifying non-defaulters while capturing as many defaulters as possible then the best model from Hyperparameter round 3 is a better choice.

11) Conclusion

Thus, by performing a detailed data exploration and processing the data accordingly, a model with good predictive performance can be built even with minimal training as seen from the results of the baseline model.

Also, it can be seen that, AutoGluon's Tabular Predictor when provided with a good dataset can arrive at a better model in lesser amount of time. Given that AutoGluon models are easily deployable in AWS as demonstrated above, the benefits of AWS like accessing high performance instances at lower costs, easily scaling the instances etc. can be procurred while training and deploying AutoGluon models.

8) References

1. https://www.kaggle.com/competitions/amex-default-prediction
2. https://www.kaggle.com/datasets/raddar/amex-data-integer-dtypes-parquet-format
3. https://www.kaggle.com/code/ambrosm/amex-eda-which-makes-sense
4. https://auto.gluon.ai/stable/tutorials/cloud_fit_deploy/cloud-aws-sagemaker-training.html
5. https://github.com/aws/amazon-sagemaker-examples/tree/main/advanced_functionality/autogluon-tabular-containers
6. https://auto.gluon.ai/stable/api/autogluon.tabular.TabularPredictor.html
7. https://auto.gluon.ai/stable/api/autogluon.tabular.TabularPredictor.fit.html#autogluon.tabular.TabularPredictor.fit
8. https://review.udacity.com/#!/reviews/3993698
9. https://onlinelibrary.wiley.com/doi/book/10.1002/9781118548387
10. https://www.statology.org/what-is-a-good-auc-score/