A MACHINE LEARNING APPROACH TO CREDIT RISK ASSESSMENT

BUAN 6341

AMEX

KODAM, RAKSHIT SAI
RIVERA, JANREY
SUBBAGARI, SRIVARDHINI REDDY

STRATEGY

CONSERVATIVE – Steady profit growth

- Rationale: Maximize profits while buffering against risks in stable markets

AGGRESSIVE – Do high-risk, high rewards apply in the credit card markets?

- Rationale: Expand market share in high-growth markets

Training Set – 2017-May to 2018-Jan

Threshold	Approval Rate	Default Rate	Revenue	Loss	Profit
0.05	52.46%	1.54%	$40,941,820	$3,641,944	$37,299,876
0.10	58.41%	2.66%	$52,799,532	$9,690,636	$43,108,895
0.15	62.00%	3.67%	$61,429,939	$17,476,911	$43,953,028
0.20	64.85%	4.71%	$69,090,305	$27,089,322	$42,000,983

Testing Set 1 – 2017-Mar to 2017-Apr

Threshold	Approval Rate	Default Rate	Revenue	Loss	Profit
0.05	55.45%	2.10%	$8,094,514	$938,006	$7,156,509
0.1	61.94%	3.52%	$10,502,897	$2,456,519	$8,046,378
0.15	65.85%	4.84%	$12,262,266	$4,197,099	$8,065,168
0.2	69.01%	6.11%	$13,768,839	$6,207,953	$7,560,885

Testing Set 2 – 2018-Feb to 2018-Mar

Threshold	Approval Rate	Default Rate	Revenue	Loss	Profit
0.1	53.08%	1.08%	$15,507,530	$873,407	$14,634,123
0.15	56.56%	1.62%	$18,375,304	$1,749,848	$16,625,456
0.2	59.39%	2.24%	$21,023,022	$3,222,195	$17,800,828
0.25	61.79%	2.90%	$23,384,902	$4,748,911	$18,635,991

PREDICT CREDIT DEFAULT

MMM-YY	COUNT	DEFAULT RATE
Mar-17	30,237	23.24%
Apr-17	31,213	23.79%
May-17	31,412	23.70%
Jun-17	31,788	25.28%
Jul-17	32,326	25.22%
Aug-17	33,506	25.32%
Sep-17	33,699	25.80%
Oct-17	34,737	25.80%
Nov-17	35,199	26.41%
Dec-17	36,662	26.81%
Jan-18	39,102	27.27%
Feb-18	41,887	27.27%
Mar-18	47,145	28.39%

- The target variable is whether our customer will default or not
- We explored data taken from **2017-Mar** to **2018-Mar**
- The dataset contained **5.5m** observations from **458,913 unique customer** observations.

Our scope of work:

- For this study, we decided to focus on (relatively) simpler models like XGBoost and Neural Networks rather than exploiting the time-series nature of the dataset

- Balancing model performance and saving computational resources

FEATURES

Category	# of Features
Delinquency	96
Spending	22
Payment	3
Balance	40
Risk	28

Feature	Min	1 Percentile	5 Percentile	Median	95 Percentile	99 Percentile	Max	Mean	% Missing
P_2	-0.46	0.005	0.220	0.68	0.97	1.01	1.01	0.65	1.23%
B_1	-1.83	0.001	0.002	0.03	0.61	1.04	1.32	0.13	0%
S_3	-0.49	0.007	0.060	0.17	0.62	1.02	4.37	0.23	18.29%
D_45	0.00	0.003	0.008	0.15	0.76	1.00	1.60	0.24	0.08%
B_2	0.00	0.003	0.020	0.81	1.01	1.01	1.01	0.62	0.07%

- The top 5 features have the best SHAP scores.
- As expected, the **payment feature** is a massive indicator for model prediction.

SAMPLING

	Time Period	# Observations	Default Rate
Train	2017-05-01 to 2018-01-31	308,431	25.80%
Test 1	2017-03-01 to 2017-04-30	61,450	23.52%
Test 2	2018-02-01 to 2018-03-31	89,032	27.86%

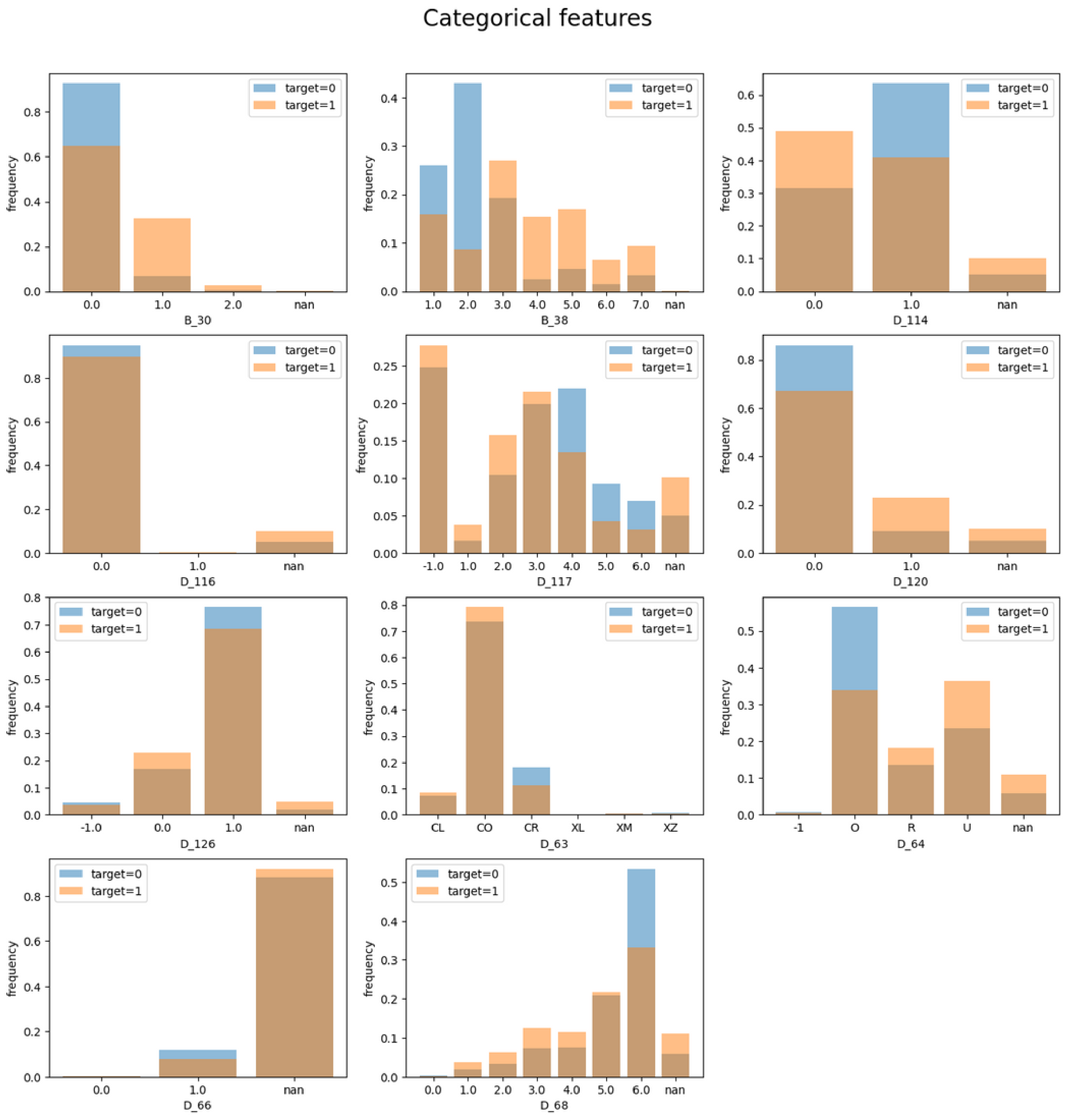
- Divided the data set into 3 sections: training set, and 2 validation sets
- We train the model using the training set, and evaluate its performance on unseen data using the two testing sets
- To avoid over-fitting, 2 testing sets were created to account for seasonality
- Our final model was chosen by implementing bias-variance analysis on all 3 sets while staying cognizant of the potential seasonality of the different time periods

DATA PROCESSING / ONE-HOT ENCODING

- Every feature has at most **eight categories** (including a NaN category). One-hot encodings are feasible.
- The distributions for target=0 and target=1 differ. This means that every feature gives some information about the target.

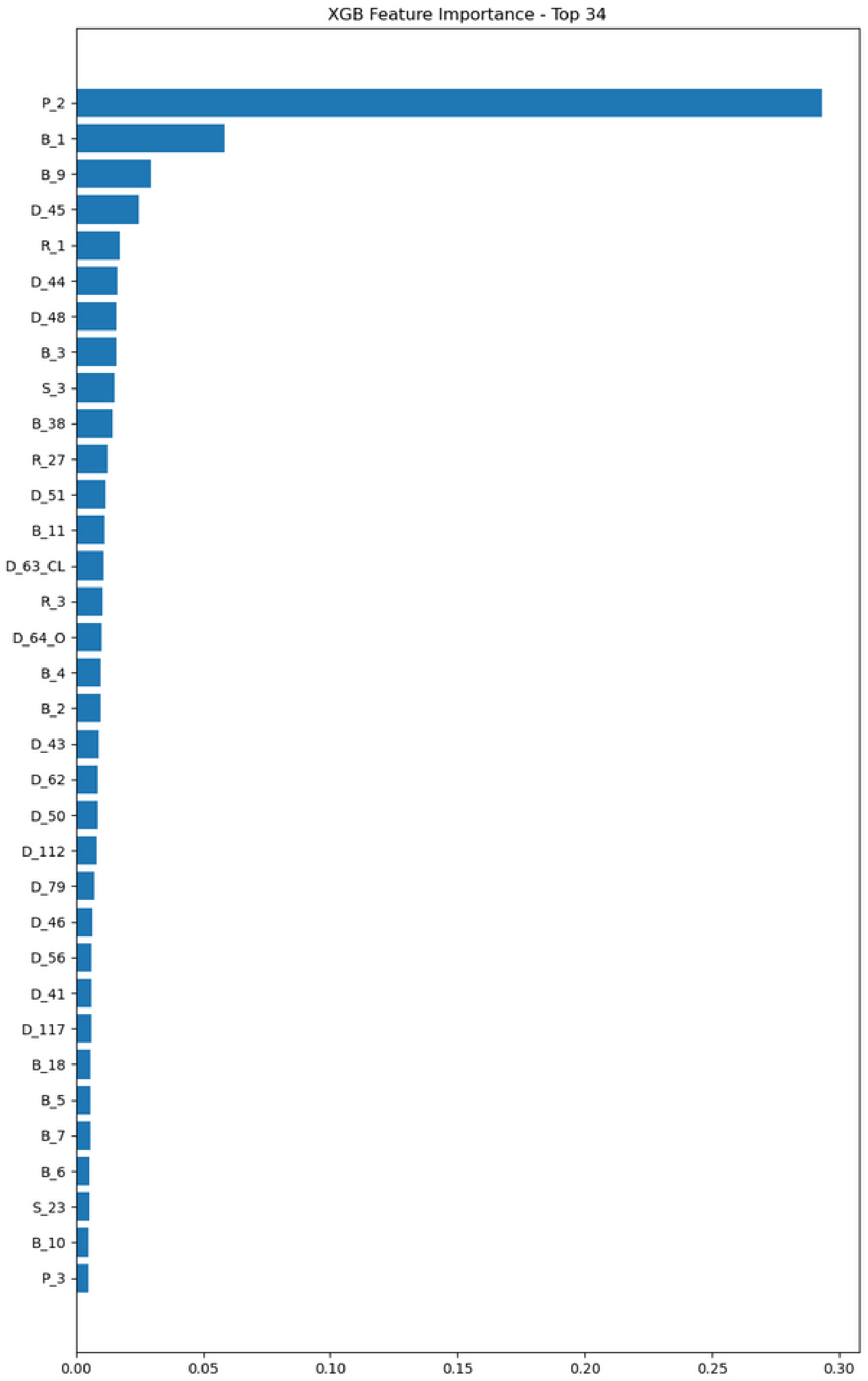
```python
# from our inspection above, non-ordinal relationships are assumed to exist for the categorical features
# 'D_63' and 'D_64'
from sklearn.preprocessing import OneHotEncoder
# create OneHotEncoder object
encoder = OneHotEncoder()
# fit and transform the data
encoded_data = encoder.fit_transform(df[['D_63', 'D_64']]).toarray()
```

```
Data columns (total 11 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   D_63_CL   458913 non-null  float64
 1   D_63_CO   458913 non-null  float64
 2   D_63_CR   458913 non-null  float64
 3   D_63_XL   458913 non-null  float64
 4   D_63_XM   458913 non-null  float64
 5   D_63_XZ   458913 non-null  float64
 6   D_64_-1   458913 non-null  float64
 7   D_64_O    458913 non-null  float64
 8   D_64_R    458913 non-null  float64
 9   D_64_U    458913 non-null  float64
 10  D_64_nan  458913 non-null  float64
dtypes: float64(11)
```

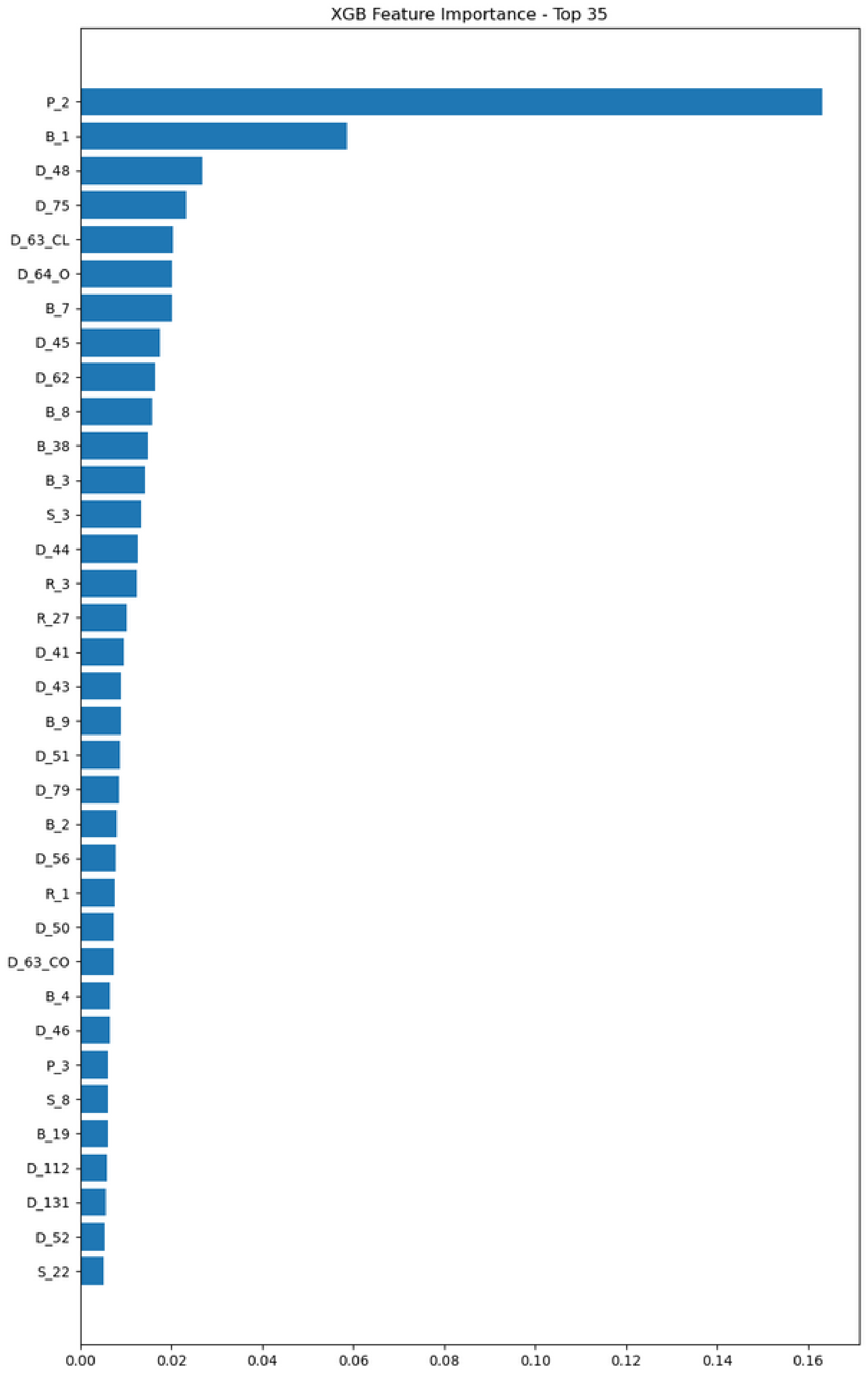


FEATURE SELECTION

XGB Default parameters - Top 5: P_2, B_1, B_9, D_45, R_1



XGB with parameters selected - Top 5: P_2, B_1, D_48, D_75, D_63_CL



- First, we used default parameters for XGBoost to calculate feature importance
- The second approach used parameters for the same model to calculate feature importance (trees = 300, learning rate = 0.5, max depth = 4, subsample = 0.5, features = 0.5, scale positive weight = 5)
- **Adds a little bit of complexity** (at least to the feature selection) but it gives us a peek at **what features stand out early at this stage**
- Given a threshold of >0.5%, we are left with **27 merged** features significantly impacting our model

Category	**# of features**	**# selected**
Delinquency	96	14
Spending	22	1
Payment	3	2
Balance	40	7
Risk	28	3

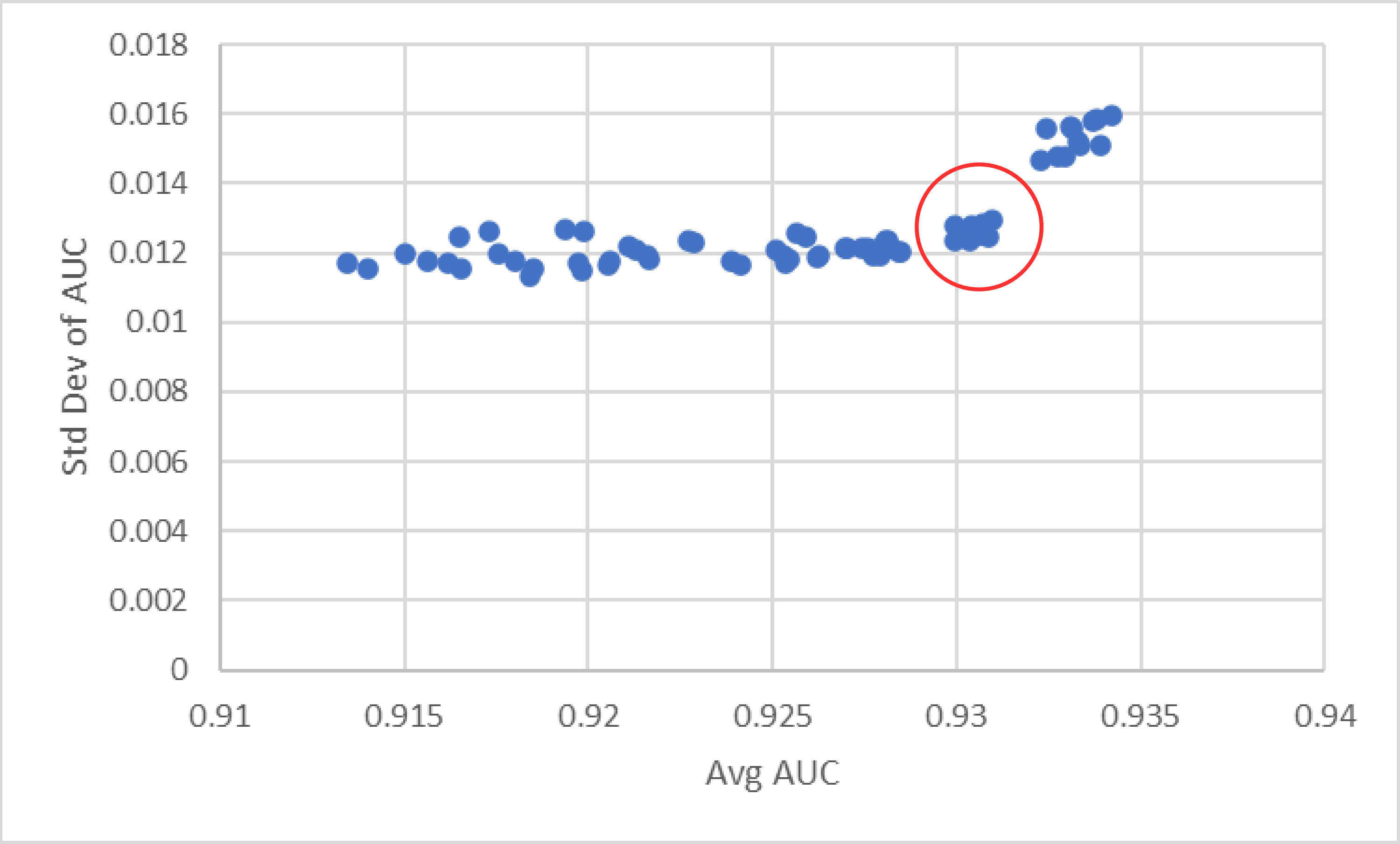
XGB GRID SEARCH- HYPERPARAMETER TUNING

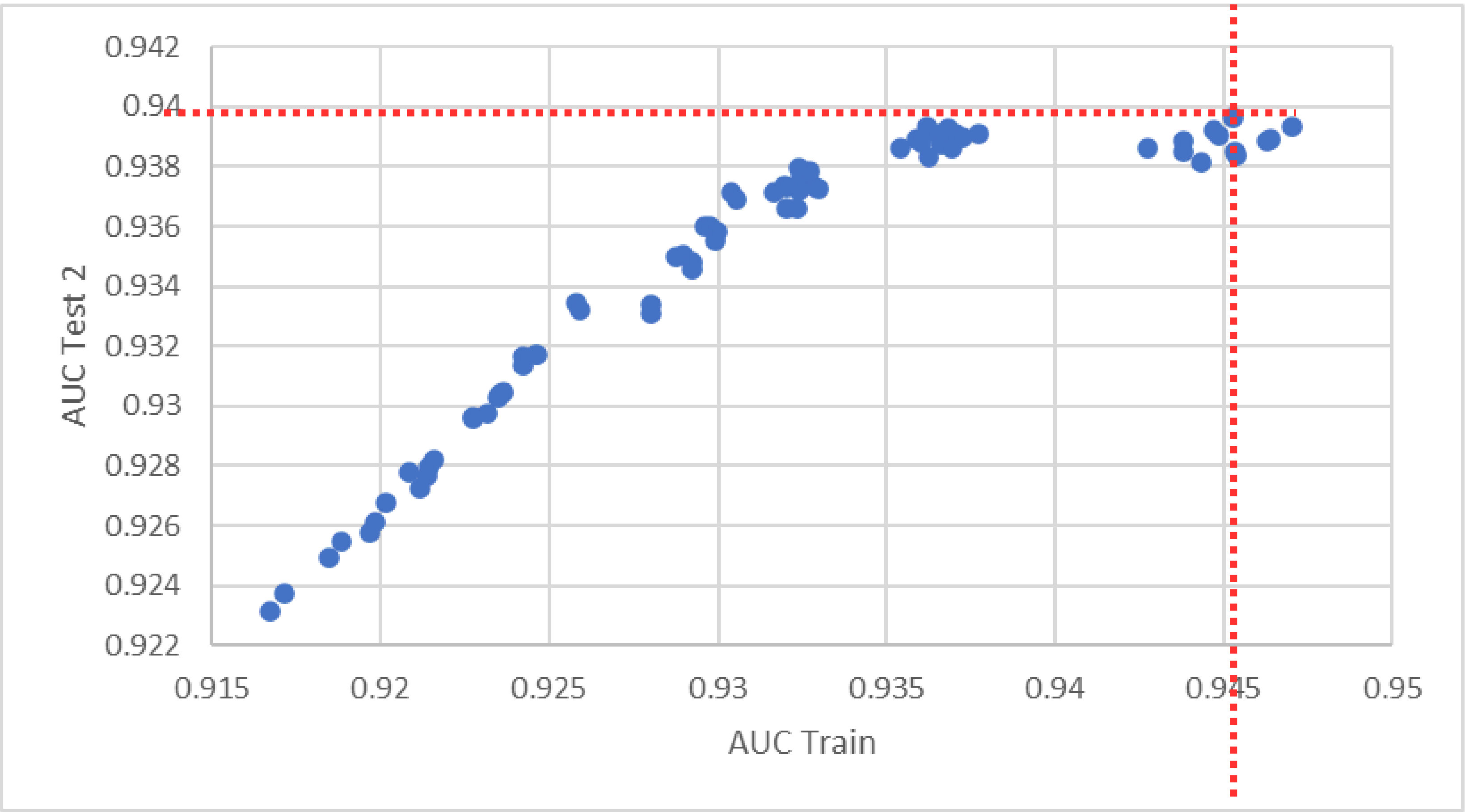
```
table = pd.DataFrame(columns = ["Num Trees", "Learning Rate", "Subsample", "% Features", "Weight of Default", "AUC Train"

row = 0
for num_trees in [50, 100, 300]:
  for lr in [0.01, 0.1]:
    for perc_obs in [0.5, 0.8]:
        for perc_features in [0.5, 1.0]:
            for scale_weight in [1, 5, 10]:
                xgb_instance3 = xgb.XGBClassifier(n_estimators=num_trees, learning_rate = lr, subsample=perc_obs,
                                                  colsample_bytree=perc_features, scale_pos_weight=scale_weight,
                                                  random_state=seed)
                model_grid = xgb_instance3.fit(X_train, Y_train)

                table.loc[row,"Num Trees"] = num_trees
                table.loc[row,"Learning Rate"] = lr
                table.loc[row,"Subsample"] = perc_obs
                table.loc[row,"% Features"] = perc_features
                table.loc[row,"Weight of Default"] = scale_weight
                table.loc[row,"AUC Train"] = roc_auc_score(Y_train, model_grid.predict_proba(X_train)[:,1])
                table.loc[row,"AUC Test 1"] = roc_auc_score(test1['target'], model_grid.predict_proba(xtest1)[:,1])
                table.loc[row,"AUC Test 2"] = roc_auc_score(test2['target'], model_grid.predict_proba(xtest2)[:,1])

                row = row + 1

table
```

- We wanted our model to perform at the level we need for production
- For the same reason that we dropped most parameters, and although we sacrifice in model's performance, we gain a lot in computational efficiency by identifying weak features and excluding those before hyperparameter tuning
- Hyperparameter tuning save us time in training and evaluating multiple models with different parameters
- We go through an iterative cycle of training and validation using **varying degrees of complexity in our models** (from less to more complex)
- **72 models trained** and **validated twice**, total of **216 iteration**

#of trees = 50, 100, 300 **Learning rate** = 0.01, 0.1 **#of observations used in each tree** = 50%, 80% **% of features used in each tree** = 50% 100% **Weight of default observations** = 1, 5, 10

XGB GRID SEARCH





- Red circle indicates high AUC with lower standard deviation

- Point of intersection = highest AUC with minimal variances between Train and Test 2

FINAL MODEL: **#of trees** = 100 **Learning rate** = 0.1 **#of observations used in each tree** = 50% **% of features used in each tree** = 50% **Weight of default observations** = 1

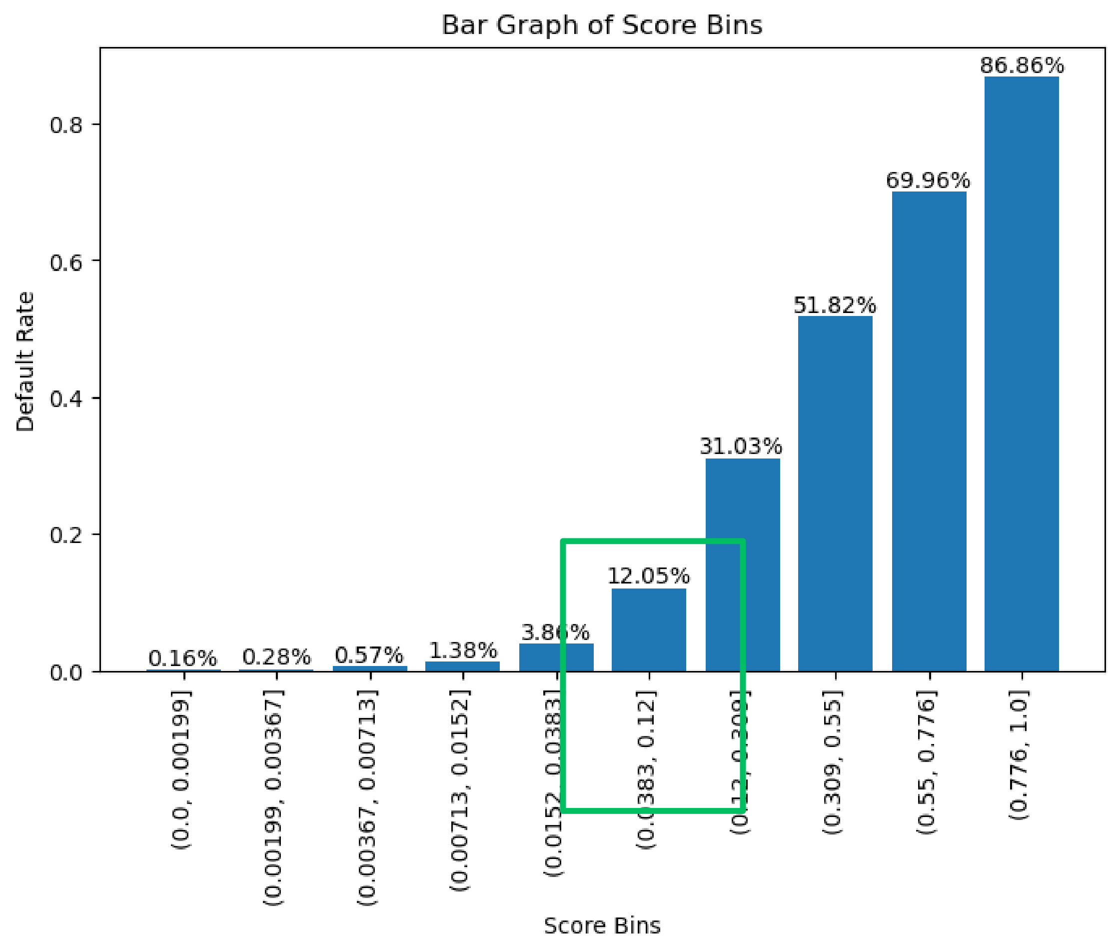
CONSIDERATION: **#of trees** = 100 **Learning rate** = 0.1 **#of observations used in each tree** = 50% **% of features used in each tree** = 50% **Weight of default observations** = 5

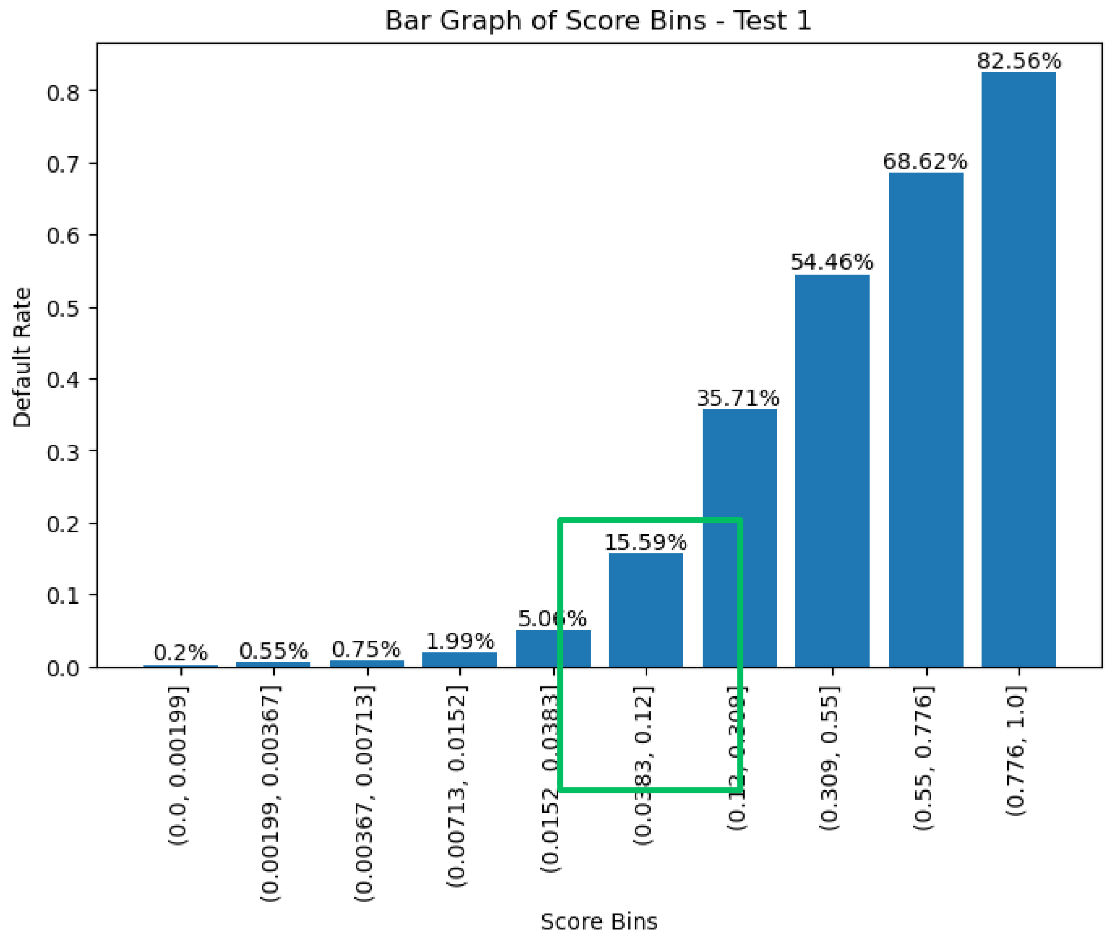
Note: For future predictions, we will continue to compare our final model with our 2nd model for consideration that adjusts for unbalanced data.

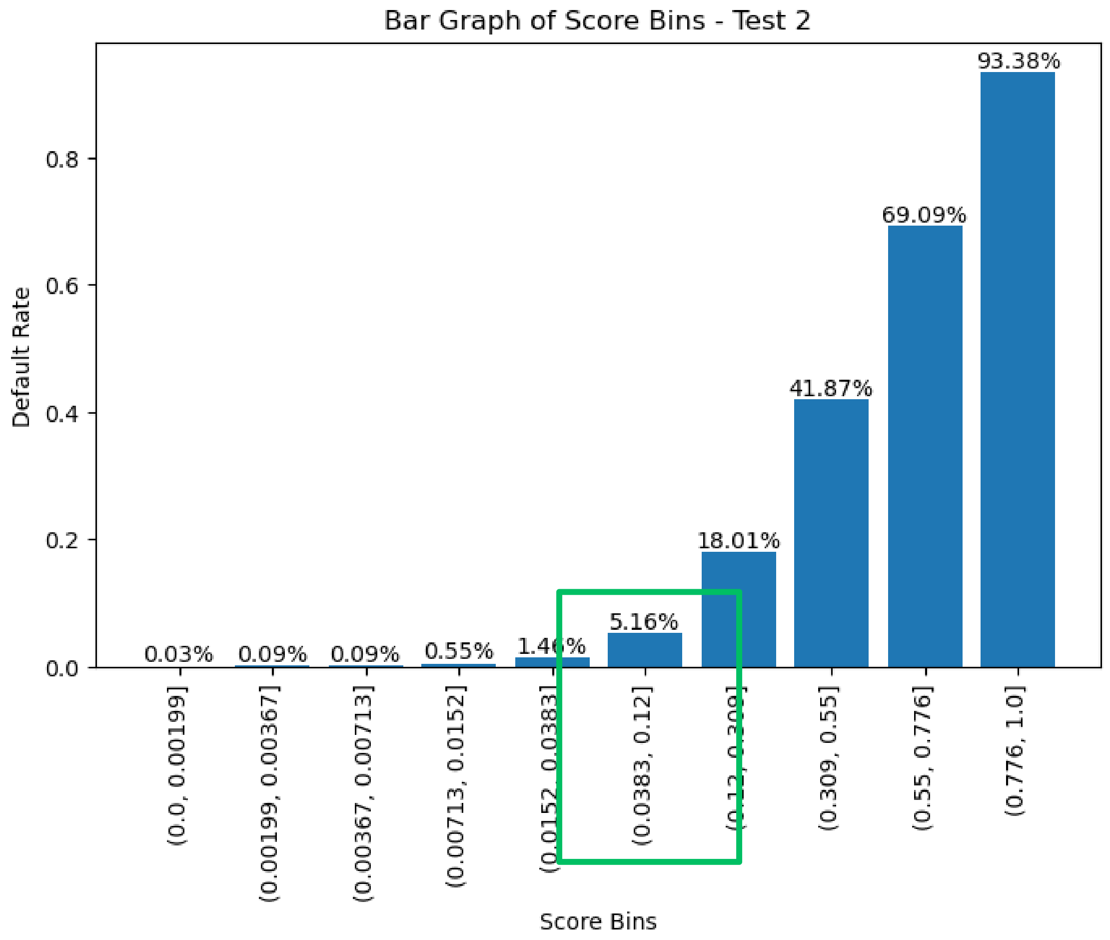
XGB GRID SEARCH- FINAL MODEL

FINAL MODEL: **#of trees** = 100 **Learning rate** = 0.1 **#of observations used in each tree** = 50% **% of features used in each tree** = 50% **Weight of default observations** = 1

- For all charts, the **6th bin (0.0383, 0.12]** is the range of threshold that meets our criteria of less than 10% default
- You see our selected model outperforming the most for Test 2







XGB SHAP ANALYSIS

- Enables interpretation for an XGB model
- Features are listed in decreasing order of importance
- As P2 increase, the probability of default decreases
- Increase in B1, S3 and B4 causes increase in probability of default

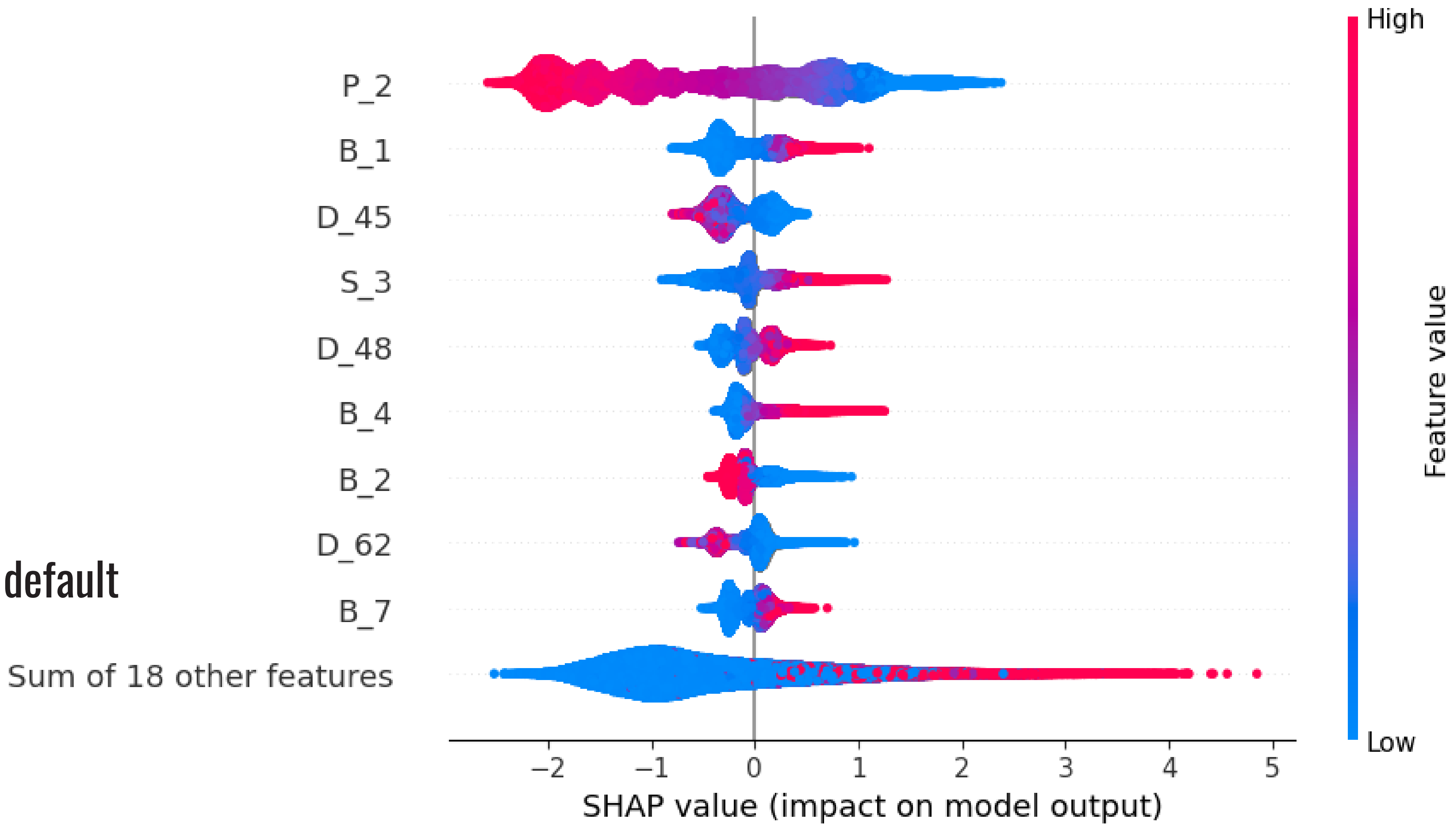
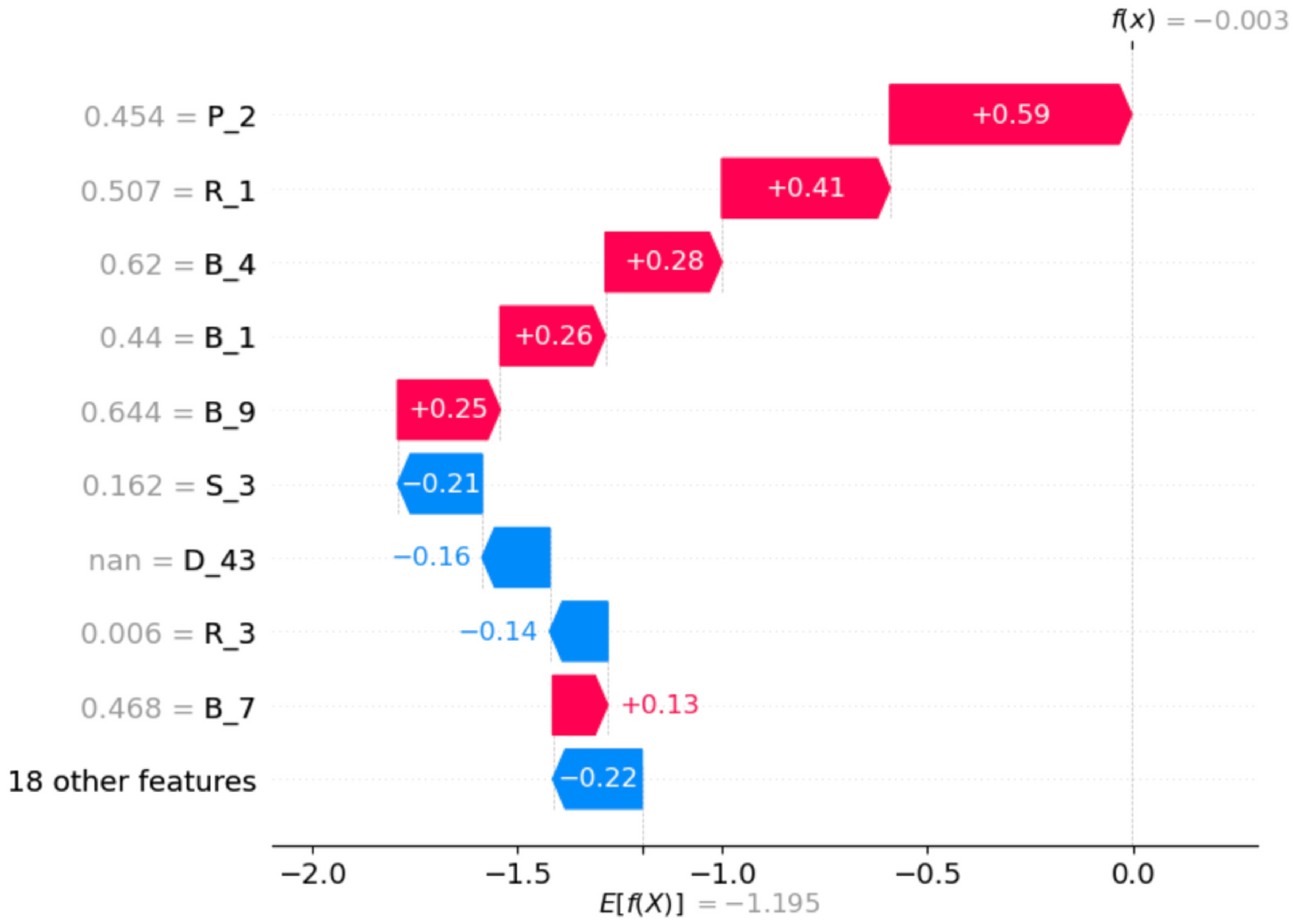


FIG: SHAP TRAIN

XGB SHAP ANALYSIS

- The red color indicates a positive contribution and decreases the probability of default.
- The Blue color indicates a negative contribution and increases the probability that a customer can default.



NEURAL NETWORK - DATA PROCESSING

Normalization

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc.fit(X_train)
```

```
StandardScaler()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
X_train_normalized = sc.transform(X_train)
X_test1_normalized = sc.transform(xtest1)
X_test2_normalized = sc.transform(xtest2)
```

```
type(X_test2_normalized)
```

```
numpy.ndarray
```

```
# convert to Pandas DF
X_train_normalized = pd.DataFrame(X_train_normalized, columns=X_train.columns)
X_test1_normalized = pd.DataFrame(X_test1_normalized, columns=xtest1.columns)
X_test2_normalized = pd.DataFrame(X_test2_normalized, columns=xtest2.columns)
```

```
type(X_test2_normalized)
```

```
pandas.core.frame.DataFrame
```

```
X_train_normalized.fillna(0,inplace=True)
X_test1_normalized.fillna(0,inplace=True)
X_test2_normalized.fillna(0,inplace=True)
```

Missing Value Imputation

```
X_train_normalized['P_3'] = np.where((X_train_normalized['P_3'] > 2.404090), 2.404090, X_train_normalized['P_3'])
X_train_normalized['B_9'] = np.where((X_train_normalized['B_9'] > 2.816933), 2.816933, X_train_normalized['B_9'])
X_train_normalized['D_44'] = np.where((X_train_normalized['D_44'] > 4.072960), 4.072960, X_train_normalized['D_44'])
X_train_normalized['S_3'] = np.where((X_train_normalized['S_3'] > 4.017150), 4.017150, X_train_normalized['S_3'])
X_train_normalized['R_1'] = np.where((X_train_normalized['R_1'] > 4.189105), 4.189105, X_train_normalized['R_1'])
X_train_normalized['R_3'] = np.where((X_train_normalized['R_3'] > 3.975631), 3.975631, X_train_normalized['R_3'])
X_train_normalized['D_48'] = np.where((X_train_normalized['D_48'] > 1.901949), 1.901949, X_train_normalized['D_48'])
X_train_normalized['D_51'] = np.where((X_train_normalized['D_51'] > 3.624639), 3.624639, X_train_normalized['D_51'])
X_train_normalized['D_50'] = np.where((X_train_normalized['D_50'] > 1.516028), 1.516028, X_train_normalized['D_50'])
X_train_normalized['D_43'] = np.where((X_train_normalized['D_43'] > 3.972277), 3.972277, X_train_normalized['D_43'])
X_train_normalized['D_46'] = np.where((X_train_normalized['D_46'] > 3.116399), 3.116399, X_train_normalized['D_46'])
X_train_normalized['D_41'] = np.where((X_train_normalized['D_41'] > 4.595613), 4.595613, X_train_normalized['D_41'])
X_train_normalized['D_62'] = np.where((X_train_normalized['D_62'] > 3.521453), 3.521453, X_train_normalized['D_62'])
X_train_normalized['D_79'] = np.where((X_train_normalized['D_79'] > 4.258512), 4.258512, X_train_normalized['D_79'])
X_train_normalized['D_56'] = np.where((X_train_normalized['D_56'] > 3.843661), 3.843661, X_train_normalized['D_56'])
X_train_normalized['B_4'] = np.where((X_train_normalized['B_4'] > 3.773781), 3.773781, X_train_normalized['B_4'])

X_train_normalized['B_1'] = np.where((X_train_normalized['B_1'] < -0.585182), -0.585182, X_train_normalized['B_1'])
X_train_normalized['B_7'] = np.where((X_train_normalized['B_7'] < -0.796128), -0.796128, X_train_normalized['B_7'])
X_train_normalized['D_50'] = np.where((X_train_normalized['D_50'] < -0.312234), -0.312234, X_train_normalized['D_50'])
X_train_normalized['P_3'] = np.where((X_train_normalized['P_3'] < -3.431286), -3.431286, X_train_normalized['P_3'])
X_train_normalized['D_46'] = np.where((X_train_normalized['D_46'] < 2.770751), 2.770751, X_train_normalized['D_46'])

X_train_normalized.describe(percentiles=[0.01, 0.99]).transpose()
```

Before Outllier Treatment

```
X_train_normalized.describe(percentiles=[0.01, 0.99]).transpose()
```

	count	mean	std	min	1%	50%	99%	max
P_2	304837.0	-6.229839e-16	1.000002	-4.353256	-2.611577	0.134552	1.459009	1.476754
B_1	308431.0	1.516479e-16	1.000002	-9.266342	-0.585182	-0.433134	4.179080	5.682981
B_9	308431.0	5.740975e-18	1.000002	-0.665863	-0.665036	-0.566823	2.816933	52.512180
D_45	308179.0	-7.215861e-17	1.000002	-1.002609	-0.990234	-0.318900	3.118147	5.580388
R_1	308431.0	4.415699e-17	1.000002	-0.348937	-0.348414	-0.322758	4.189105	13.174340
D_44	292363.0	1.254103e-16	1.000002	-0.537534	-0.536847	-0.502273	4.072960	23.565394
D_48	267763.0	2.423233e-16	1.000002	-1.211756	-1.177791	-0.280988	1.901949	9.663061
B_3	308180.0	-8.702307e-17	1.000002	-0.553516	-0.552463	-0.511675	3.772076	6.277054
S_3	252291.0	1.937287e-16	1.000002	-3.637199	-1.136830	-0.331381	4.017150	20.954206
B_38	308180.0	-4.430438e-16	1.000002	-1.060005	-1.060005	-0.426984	2.738120	2.738120
R_27	299783.0	-5.404821e-16	1.000002	-2.853932	-2.779023	0.363147	0.380596	0.380954
D_51	308431.0	1.084029e-16	1.000002	-0.590150	-0.589537	-0.559974	3.624639	10.605510
D_63_CL	308431.0	-8.347275e-16	1.000002	-0.288573	-0.288573	-0.288573	3.465325	3.465325
R_3	308431.0	6.154351e-17	1.000002	-0.587849	-0.586964	-0.543518	3.975631	34.823289
D_64_O	308431.0	-1.046202e-15	1.000002	-1.031977	-1.031977	0.969014	0.969014	0.969014
B_4	308431.0	-2.353371e-17	1.000002	-0.774819	-0.771499	-0.402850	3.773781	13.909349
B_2	308180.0	-1.570012e-16	1.000002	-1.562139	-1.554580	0.478247	0.967684	0.968450
D_43	211346.0	-2.236221e-16	1.000002	-0.722303	-0.710066	-0.310297	3.972277	38.700598
D_62	265616.0	4.885356e-18	1.000002	-0.821733	-0.797768	-0.420544	3.521453	46.198277
D_50	130977.0	1.035861e-16	1.000004	-2.328496	-0.312234	-0.118068	1.516028	123.600977
D_112	308137.0	3.574734e-16	1.000002	-2.354057	-2.348619	0.426712	0.442771	0.443097
D_79	301096.0	-3.412239e-17	1.000002	-0.315897	-0.315395	-0.290650	4.258512	40.598503
D_46	233593.0	9.012258e-16	1.000002	-46.212135	-2.770751	-0.093012	3.116399	49.126602
D_56	139162.0	-2.535957e-16	1.000004	-1.042717	-0.942001	-0.257854	3.843661	51.162086
D_41	308180.0	-4.312733e-17	1.000002	-0.290617	-0.290002	-0.259721	4.595613	36.625908
B_7	308431.0	2.417012e-17	1.000002	-8.236380	-0.796128	-0.470047	3.595779	4.611585
P_3	282754.0	9.203648e-16	1.000002	-10.678295	-3.431286	0.093074	2.404090	7.485231

After Outllier Treatment

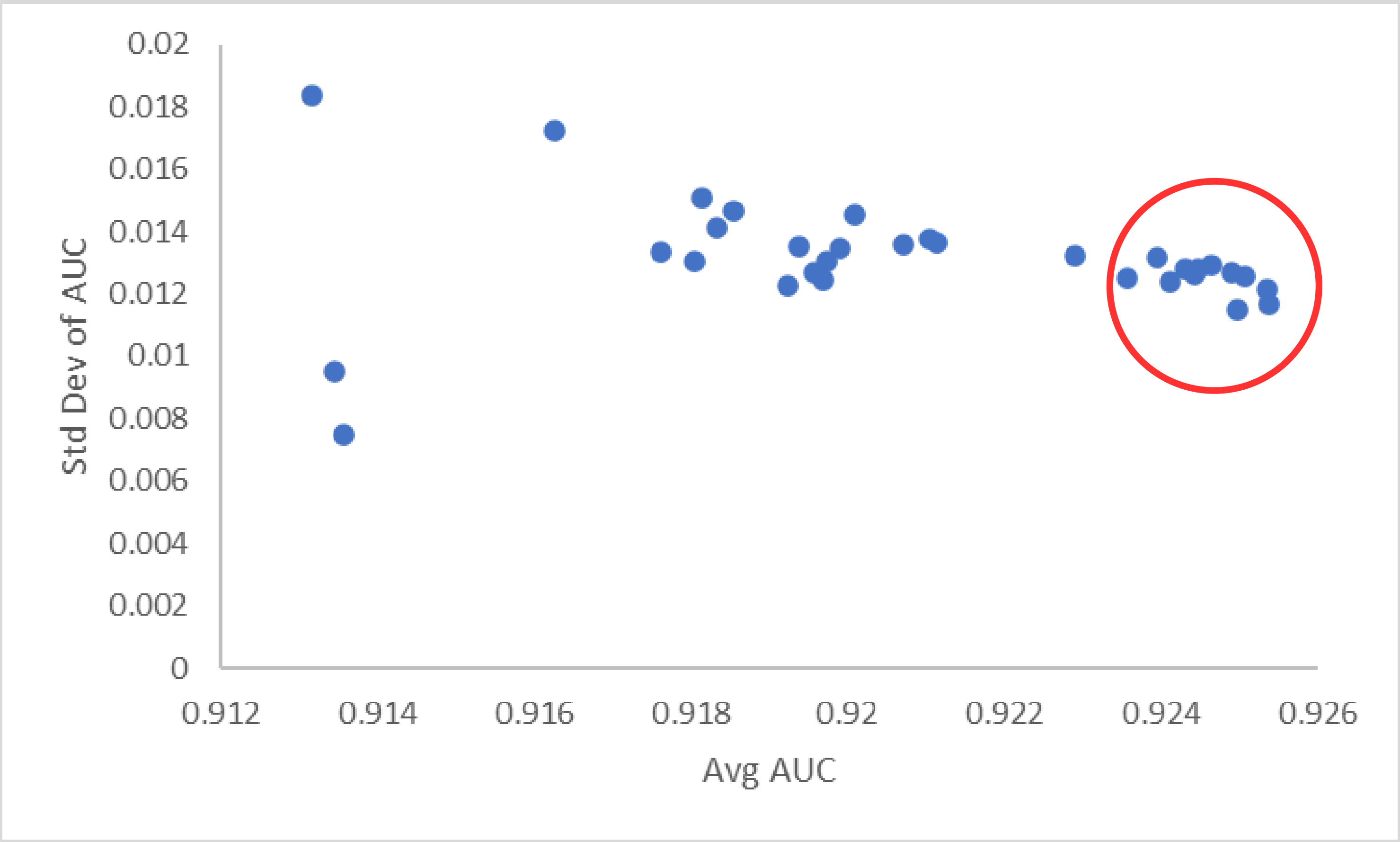
```
X_train_normalized.describe(percentiles=[0.01, 0.99]).transpose()
```

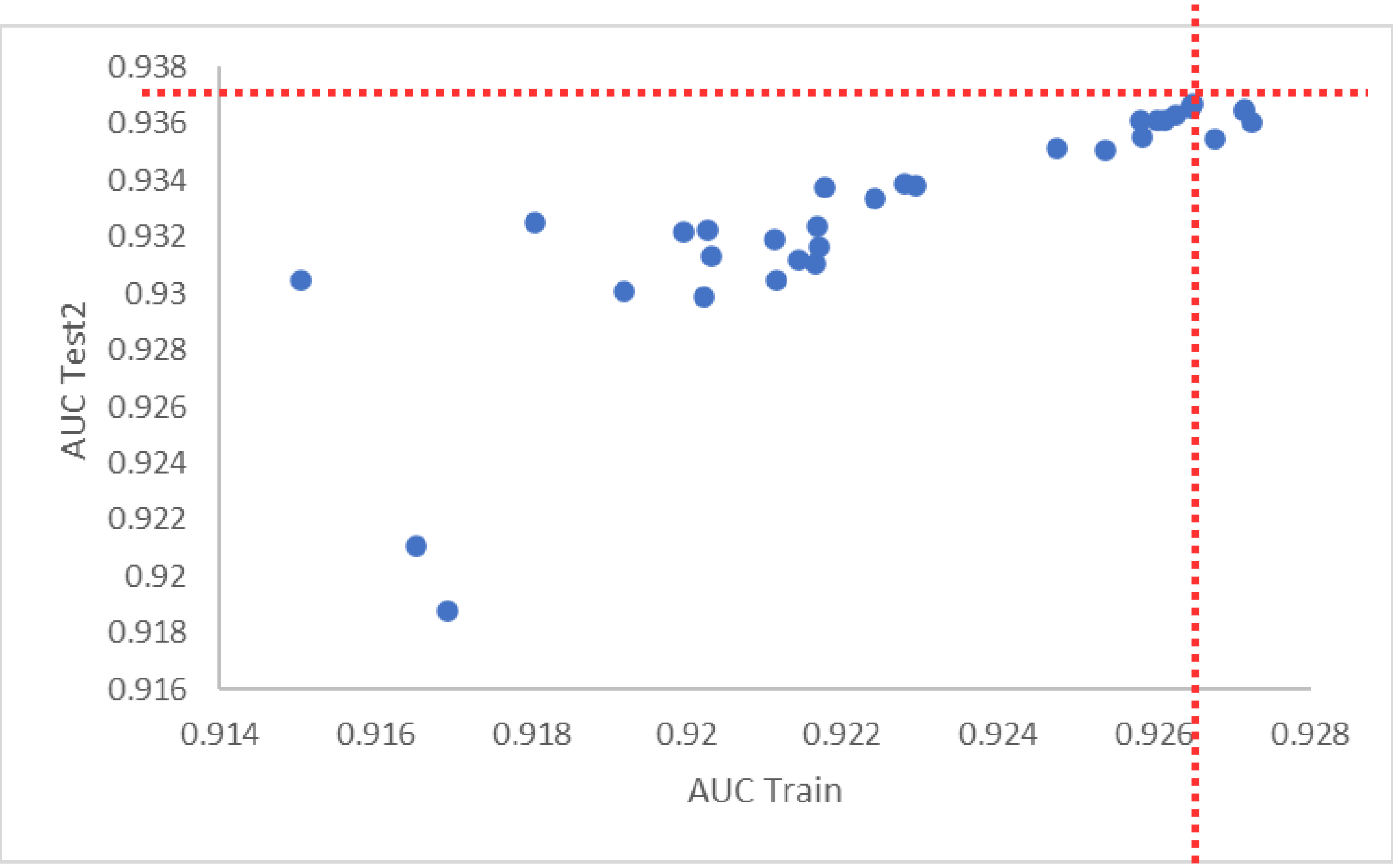
	count	mean	std	min	1%	50%	99%	max
P_2	304837.0	-6.229839e-16	1.000002	-4.353256	-2.611577	0.134552	1.459009	1.476754
B_1	308431.0	3.009003e-04	0.999487	-0.585182	-0.585182	-0.433134	4.179080	5.682981
B_9	308431.0	-1.733387e-02	0.881151	-0.665863	-0.665036	-0.566823	2.816833	2.816933
D_45	308179.0	-7.215861e-17	1.000002	-1.002609	-0.990234	-0.318900	3.118147	5.580388
R_1	308431.0	-1.749606e-02	0.897798	-0.348937	-0.348414	-0.322758	4.189104	4.189105
D_44	292363.0	-1.255397e-02	0.928565	-0.537534	-0.536847	-0.502273	4.072946	4.072960
D_48	267763.0	-3.377872e-03	0.991251	-1.211756	-1.177791	-0.280988	1.901940	1.901949
B_3	308180.0	-8.702307e-17	1.000002	-0.553516	-0.552463	-0.511675	3.772076	6.277054
S_3	252291.0	-1.362732e-02	0.921988	-3.637199	-1.136830	-0.331381	4.017139	4.017150
B_38	308180.0	-4.430438e-16	1.000002	-1.060005	-1.060005	-0.426984	2.738120	2.738120
R_27	299783.0	-5.404821e-16	1.000002	-2.853932	-2.779023	0.363147	0.380596	0.380954
D_51	308431.0	-7.967703e-03	0.962405	-0.590150	-0.589537	-0.559974	3.624637	3.624639
D_63_CL	308431.0	-8.347275e-16	1.000002	-0.288573	-0.288573	-0.288573	3.465325	3.465325
R_3	308431.0	-1.962885e-02	0.859806	-0.587849	-0.586964	-0.543518	3.975631	3.975631
D_64_O	308431.0	-1.046202e-15	1.000002	-1.031977	-1.031977	0.969014	0.969014	0.969014
B_4	308431.0	-1.187226e-02	0.938046	-0.774819	-0.771499	-0.402850	3.773641	3.773781
B_2	308180.0	-1.570012e-16	1.000002	-1.562139	-1.554580	0.478247	0.967684	0.968450
D_43	211346.0	-2.238638e-02	0.829263	-0.722303	-0.710066	-0.310297	3.972088	3.972277
D_62	265616.0	-1.239389e-02	0.916883	-0.821733	-0.797768	-0.420544	3.521021	3.521453
D_50	130977.0	-3.261793e-02	0.285804	-0.312234	-0.312234	-0.118068	1.515967	1.516028
D_112	308137.0	3.574734e-16	1.000002	-2.354057	-2.348619	0.426712	0.442771	0.443097
D_79	301096.0	-1.623519e-02	0.877319	-0.315897	-0.315395	-0.290650	4.258511	4.258512
D_46	233593.0	2.774793e+00	0.036242	2.770751	2.770751	2.770751	3.116356	3.116399
D_56	139162.0	-2.115267e-02	0.830290	-1.042717	-0.942001	-0.257854	3.843260	3.843661
D_41	308180.0	-2.498805e-02	0.790516	-0.290617	-0.290002	-0.259721	4.595515	4.595613
B_7	308431.0	1.549305e-04	0.999697	-0.796128	-0.796128	-0.470047	3.595779	4.611585
P_3	282754.0	2.688686e-03	0.926653	-3.431286	-3.431107	0.093074	2.404077	2.404090

NEURAL NETWORK - GRID SEARCH

```
# fine tuning with Grid Search
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV


def build_classifier(num_hidden_layers, num_nodes, activation, dropout_rate):
    # first step: create a Sequential object, as a sequence of layers. B/C NN is a sequence of layers.
    classifier = Sequential()

    # run for loops for different hidden layers
    classifier.add(Dense(num_nodes, activation = activation, input_shape=(X_train.shape[1],)))

    # adding hidden layers
    for i in range(num_hidden_layers-1):
        classifier.add(Dense(num_nodes, activation = activation))
        if dropout_rate != 0:
            classifier.add(Dropout(rate = dropout_rate))

    # add the output layer
    classifier.add(Dense(units=1, activation='sigmoid'))

    #compiling the NN
    classifier.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
    return classifier
```

NUERAL NETWORK - GRID SEARCH





- Red circle indicates high AUC with lower standard deviation

- Point of intersection = highest AUC with minimal variances between Train and Test 2

FINAL MODEL: #of trees = 100 **Learning rate** = 0.1 **#of observations used in each tree** = 50% **% of features used in each tree** = 50% **Weight of default observations** = 1

CONSIDERATION: #of trees = 100 **Learning rate** = 0.1 **#of observations used in each tree** = 50% **% of features used in each tree** = 50% **Weight of default observations** = **5**

Note: For future predictions, we will continue to compare our final model with our 2nd model for consideration that adjusts for unbalanced data.

NEURAL NETWORK - GRID SEARCH

- please continue the rest

FINAL MODEL

- please continue the rest

STRATEGY

- We could either adopt a conservative or an aggressive strategy.
- The only criterion is that the Default rate is less than 10%
- For the training sample

  Aggressive approach - Threshold = 35% , Default Rate = 9.63%

  Conservative approach - Threshold = 15% , Default Rate = 4.84%

To compensate for the increased risk, we also wanted to incorporate the impact of expected losses incurred by credit lenders in both approaches. This would help us appreciate the stark difference in both the approaches

Threshold	Approval Rate	Default Rate	Revenue	Loss	Profit
0.05	55.45%	2.10%	$8,094,514	$938,006	$7,156,509
0.10	61.94%	3.52%	$10,502,897	$2,456,519	$8,046,378
0.15	65.85%	4.84%	$12,262,266	$4,197,099	$8,065,168
0.20	69.01%	6.11%	$13,768,839	$6,207,953	$7,560,885
0.25	71.64%	7.28%	$15,138,962	$8,768,141	$6,370,821
0.30	74.11%	8.50%	$16,473,032	$11,898,610	$4,574,422
0.35	76.29%	9.63%	$17,594,083	$15,359,050	$2,235,033
0.40	78.33%	10.72%	$18,628,268	$19,061,697	$-433,430
0.45	80.43%	11.84%	$19,705,125	$23,516,885	$-3,811,760
0.50	82.35%	12.95%	$20,674,972	$28,283,832	$-7,608,860
0.55	84.38%	14.05%	$21,773,224	$33,287,406	$-11,514,181
0.65	88.36%	16.30%	$23,684,280	$45,306,163	$-21,621,883
0.70	90.30%	17.43%	$24,481,152	$52,096,683	$-27,615,531
0.75	92.22%	18.64%	$25,109,983	$59,336,395	$-34,226,412
0.80	94.10%	19.79%	$25,720,180	$67,284,938	$-41,564,758
0.85	95.90%	20.90%	$26,309,361	$76,172,154	$-49,862,794
0.90	97.66%	21.98%	$26,817,401	$84,935,950	$-414,869,030
1.00	100.00%	25.80%	$140,321,739	$614,247,776	$-473,926,038

STRATEGY

- We could either adopt a conservative or an aggressive strategy.
- The only criterion is that the Default rate is less than 10%
- For the test sample

  Aggressive approach - Threshold = 55% , Default Rate = 9.08%

  Conservative approach - Threshold = 35% , Default Rate = 4.46%

To compensate for the increased risk, we also wanted to incorporate the impact of expected losses incurred by credit lenders in both approaches. This would help us appreciate the stark difference in both the approaches

Threshold	Approval Rate	Default Rate	Revenue	Loss	Profit
0.05	47.38%	0.59%	$11,613,906.00	$284,431.00	$11,329,475.00
0.10	53.08%	1.08%	$15,507,530.00	$873,407.00	$14,634,123.00
0.15	56.56%	1.62%	$18,375,304.00	$1,749,848.00	$16,625,456.00
0.20	59.39%	2.24%	$21,023,022.00	$3,222,195.00	$17,800,828.00
0.25	61.79%	2.90%	$23,384,902.00	$4,748,911.00	$18,635,991.00
0.30	63.90%	3.66%	$25,518,788.00	$6,913,915.00	$18,604,873.00
0.35	66.04%	4.46%	$27,685,061.00	$9,508,034.00	$18,177,027.00
0.40	68.14%	5.40%	$29,639,404.00	$13,444,092.00	$16,195,312.00
0.45	70.21%	6.48%	$31,469,447.00	$18,226,795.00	$13,242,652.00
0.50	72.40%	7.72%	$33,481,660.00	$24,506,190.00	$8,975,470.00
0.55	74.64%	9.08%	$35,217,952.00	$31,848,679.00	$3,369,273.00
0.65	79.14%	12.05%	$38,430,993.00	$51,595,495.00	-$13,164,502.00
0.70	81.51%	13.76%	$39,785,488.00	$64,725,588.00	-$24,940,100.00
0.75	83.95%	15.55%	$40,917,876.00	$79,265,906.00	-$38,348,030.00
0.80	86.56%	17.54%	$41,835,618.00	$96,842,931.00	-$55,007,313.00
0.85	89.32%	19.68%	$42,640,326	$117,953,536	$-75,313,209
0.90	92.32%	22.00%	$43,223,405	$143,287,288	$-100,063,883
0.95	95.71%	24.66%	$43,484,987	$175,990,483	$-132,505,496
1.00	100.00%	27.87%	$43,555,254	$224,040,707	$-180,485,453

STRATEGY

	Train			Test1			Test2			Total		
Threshold	Approval Rate	Default Rate	Revenue	Approval Rate	Default Rate	Revenue	Approval Rate	Default Rate	Revenue	Approval Rate	Default Rate	Revenue
0.05	52.46%	1.54%	40,941,820.00	55.45%	2.10%	8,094,514.00	47.38%	0.59%	11,613,906.00	38.82%	1.41%	20,216,746.67
0.1	58.41%	2.66%	52,799,532.00	61.94%	3.52%	10,502,897.00	53.08%	1.08%	15,507,530.00	43.36%	2.42%	26,269,986.33
0.15	62.00%	3.67%	61,429,939.00	65.85%	4.84%	12,262,266.00	56.56%	1.62%	18,375,304.00	46.10%	3.38%	30,689,169.67
0.2	64.85%	4.71%	69,090,305.00	69.01%	6.11%	13,768,839.00	59.39%	2.24%	21,023,022.00	48.31%	4.35%	34,627,388.67
0.25	67.34%	5.78%	75,957,884.00	71.64%	7.28%	15,138,962.00	61.79%	2.90%	23,384,902.00	50.19%	5.32%	38,160,582.67
0.3	69.61%	6.86%	82,340,704.00	74.11%	8.50%	16,473,032.00	63.90%	3.66%	25,518,788.00	51.91%	6.34%	41,444,174.67
0.35	71.75%	7.97%	88,374,758.00	76.29%	9.63%	17,594,083.00	66.04%	4.46%	27,685,061.00	53.52%	7.35%	44,551,300.67
0.4	73.83%	9.10%	94,046,757.00	78.33%	10.72%	18,628,268.00	68.14%	5.40%	29,639,404.00	55.08%	8.41%	47,438,143.00
0.45	75.87%	10.22%	99,684,458.00	80.43%	11.84%	19,705,125.00	70.21%	6.48%	31,469,447.00	56.63%	9.51%	50,286,343.33
0.5	77.95%	11.43%	105,060,174.00	82.35%	12.95%	20,674,972.00	72.40%	7.72%	33,481,660.00	58.18%	10.70%	53,072,268.67
0.55	80.00%	12.64%	110,173,886.00	84.38%	14.05%	21,773,224.00	74.64%	9.08%	35,217,952.00	59.76%	11.92%	55,721,687.33
0.65	84.30%	15.31%	120,066,676.00	88.36%	16.30%	23,684,280.00	79.14%	12.05%	38,430,993.00	62.95%	14.55%	60,727,316.33
0.7	86.50%	16.75%	124,267,151.00	90.30%	17.43%	24,481,152.00	81.51%	13.76%	39,785,488.00	64.58%	15.98%	62,844,597.00
0.75	88.79%	18.22%	128,507,598.00	92.22%	18.64%	25,109,983.00	83.95%	15.55%	40,917,876.00	66.24%	17.47%	64,845,152.33
0.8	91.15%	19.77%	132,181,312.00	94.10%	19.79%	25,720,180.00	86.56%	17.54%	41,835,618.00	67.95%	19.03%	66,579,036.67
0.85	93.58%	21.40%	135,281,750.00	95.90%	20.90%	26,309,361.00	89.32%	19.68%	42,640,326.00	69.70%	20.66%	68,077,145.67
0.9	95.95%	23.00%	137,809,640.00	97.66%	21.98%	26,817,401.00	92.32%	22.00%	43,223,405.00	71.48%	22.33%	69,283,482.00
0.95	98.26%	24.57%	139,572,753.00	99.11%	22.92%	27,149,257.00	95.71%	24.66%	43,484,987.00	73.27%	24.05%	70,068,999.00
1	100.00%	25.80%	140,321,739.00	100.00%	23.52%	27,323,178.00	100.00%	27.87%	43,555,254.00	75.00%	25.73%	70,400,057.00

Comparing the total sample
Aggressive approach - Threshold = 45% , Default Rate =9.51 %
Conservative approach - Threshold = 25% , Default Rate = 5.32%

STRATEGY

We believe that the conservative approach is better than the aggressive approach based on the fact that the net profit generated is higher. Despite the fact that the loss values are approximated, the increase in revenue by increasing the acceptance rate seems to have been compensated by default in balances.