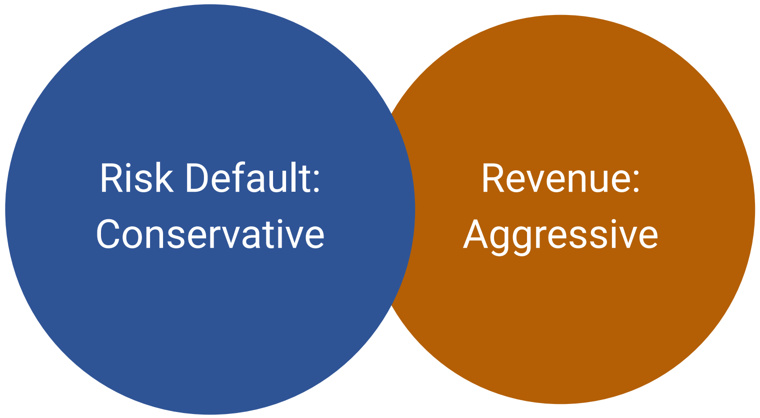
Credit Risk Modeling

1. Executive Summary

- The problem statement related to predicting probability of default of credit card customer.
- Building a classification model would solidify our strategy. They are as follows:
  - Conservative Strategy: While choosing to minimise risk of default as our primary parameters, we choose threshold of ~0.4
  - Aggressive Strategy: While choosing Revenue as our primary parameter, we choose threshold of ~0.6

Strategy	Threshold	Train				Test-1				Test-2				Total			
		Total	Default	Rate	Revenue	Total	Default	Rate	Revenue	Total	Default	Rate	Revenue	Total	Default	Rate	Revenue
Aggressive	0.6	242866	24949	0.102727	615.5625	60641	7338	0.121007	142.0335	56035	5712	0.101936	149.9783	359490	37878	0.105366	908.0072
Conservative	0.4	203484	7699	0.037836	474.1081	50809	2764	0.0544	110.7274	46255	1749	0.037812	114.6355	300622	12131	0.040353	700.2154



- Conservative strategy would always have a lesser threshold than the Aggressive threshold.
- The strategies are from a trade off between risk and revenue.

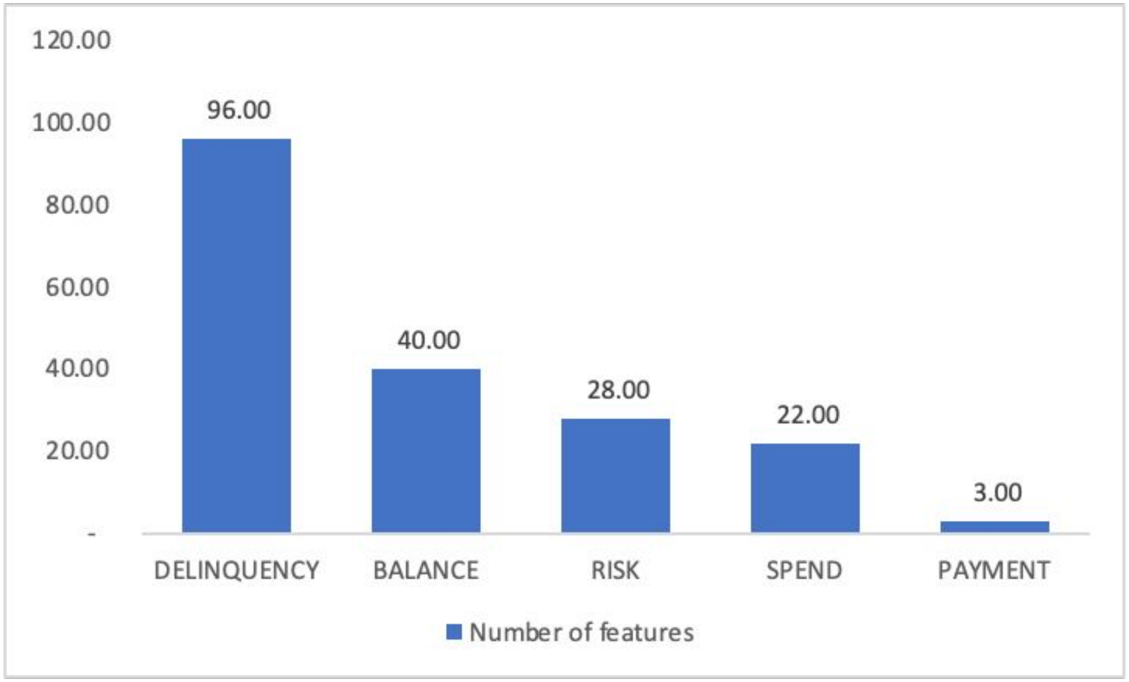
2. Data

- Our dataset contained a set of **458,913** unique customers and 190 aggregated profile features which we will use to determine the probability that a customer will default.
- The target variable is "1"if the customer defaults, else "0".
- A default event occurs if the customer does not make the required payment within 120 days of the date of their most recent statement.
- This model is going to be used in credit approval decisioning, if we approve or not a credit product we define as:
  - Default: 0 Credit approved
  - Not Default: 1 Credit rejected
- If the applicant misses **3 consecutive payments** in the next 12 months.

MONTH	#OBSERVATION	DEFAULT RATE
2017/03	30,545	0.2277
2017/04	31,120	0.2333
2017/05	31778	0.2379
2017/06	32160	0.2448
2017/07	32542	0.2511
2017/08	33300	0.2549
2017/09	33864	0.2577
2017/10	34174	0.266
2017/11	35535	0.267
2017/12	36049	0.27
2018/01	39141	0.2741
2018/02	41066	0.2867
2018/03	47631	0.2902
TOTAL	458913	0.2608

3. Features

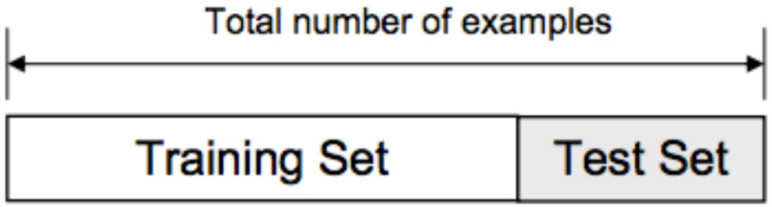
- We see that most features are not equally distributed among their categories, where we have a high number of delinquencies and low number of full payments



CATEGORY	# OF VALUES
DELINQUENCY	96.00
BALANCE	40.00
RISK	28.00
SPEND	22.00
PAYMENT	3.00

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

4. Feature Engineering



- Using S_2, we divided the data into train, test1 and test2 datasets.
- You can see the most significant feature description using the SHAP analysis shown in later slides.

```
test1 = dev_set[dev_set['S_2'] < 201705]

test2 = dev_set[dev_set['S_2'] > 201801]

train1 = dev_set[(dev_set['S_2'] <= 201801) & (dev_set['S_2'] >= 201705)]
```

Feature	count	mean	std	min	1%	5%	50%	95%	99%	max	Nulls
P_2	452,970	0.6499	0.2430	(0.4338)	0.0109	0.2256	0.6815	0.9739	1.0056	1.0100	5,943
D_42	88,180	0.1875	0.2473	(0.0003)	0.0029	0.0072	0.1184	0.5882	1.0984	4.1892	370,733
B_4	458,913	0.1686	0.2177	0.0000	0.0007	0.0035	0.0811	0.6086	0.9911	3.4899	-
D_45	458,528	0.2378	0.2416	0.0000	0.0026	0.0080	0.1561	0.7585	0.9949	1.5973	385
D_48	397,627	0.3852	0.3255	(0.0096)	0.0014	0.0135	0.2951	0.9403	1.0037	8.9671	61,286

5. Data Processing – One Hot Encoding

D_63	
CL	34307
CO	344667
CR	74758
XL	629
XM	1510
XZ	3042

```
df_set[['S_2','D_63']].groupby(["D_63"]).count()
```

```
D_63_dummies = pd.get_dummies(df_set.D_63)
```

```
D_63_dummies.head(10)
```

	CL	CO	CR	XL	XM	XZ
0	0	0	1	0	0	0
1	0	1	0	0	0	0
2	0	1	0	0	0	0
3	0	1	0	0	0	0
4	0	1	0	0	0	0
5	0	1	0	0	0	0
6	0	1	0	0	0	0
7	0	1	0	0	0	0
8	0	1	0	0	0	0
9	0	1	0	0	0	0

	S_2
D_64	
-1	3340
O	231328
R	67523
U	122710

```
D_64_dummies = pd.get_dummies(df_set.D_64)
```

```
D_64_dummies.head(10)
```

	-1	O	R	U
0	0	1	0	0
1	0	1	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	0	0
5	0	0	1	0
6	0	0	1	0
7	0	0	1	0
8	0	0	0	1
9	0	1	0	0

6. Feature Selection

```python
from itertools import compress
from sklearn.feature_selection import VarianceThreshold
def fs_variance(df, threshold:float=0.05):
    """
    Return a list of selected variables based on the threshold.
    """
    # The list of columns in the data frame
    features = list(df.columns)

    # Initialize and fit the method
    vt = VarianceThreshold(threshold = threshold)
    _ = vt.fit(df)

    # Get which column names which pass the threshold
    feat_select = list(compress(features, vt.get_support()))

    return feat_select
columns_to_keep=fs_variance(data)
# We are left with 85 columns (excluding target), which passed the threshold.
train_final=data[columns_to_keep]
len(columns_to_keep)
```

```python
xgb_model1 = xgb.XGBClassifier(random_state = 42)
model1 = xgb_model1.fit(X_train, Y_train)
```

✓ 8m 3.9s

Loading...

```python
feature_importance_1 = {'Feature':X_train.columns,'Importance':model1.feature_importances_}
feature_importance_1 = pd.DataFrame(feature_importance_1)
```

- The two methods we used are as follows:
    - Variance Threshold: If it is insignificant at 0.05, then we remove it.
    - XGBoost feature importance: Run a XGBoost model and remove all variables which are not significant.
- We choose the XGBoost method as this takes multicollinearity relationship and other complex relation into consideration while building the model and calculate feature importance.

7. XGBoost - Grid Search

```python
row = 0
for numtrees in [50, 100, 300]:
    for LR in [0.01, 0.1]:
        for Subsample in [0.5, 0.8]:
            for colsample_bytree in [0.5, 1]:
                for scale_pos_weight in [1, 5, 10]:
                    xgb_instance = xgb.XGBClassifier(n_estimators=numtrees,
                                                     learning_rate=LR,
                                                     subsample=Subsample,
                                                     colsample_bytree=colsample_bytree,
                                                     scale_pos_weight=scale_pos_weight)
                    model = xgb_instance.fit(X_train_final, Y_train)

                    table.loc[row, "#Trees"] = numtrees
                    table.loc[row, "LR"] = LR
                    table.loc[row, "Subsample"] = Subsample
                    table.loc[row, "% Features"] = colsample_bytree
                    table.loc[row, "Weight of Default"] = scale_pos_weight
                    table.loc[row, "AUC Train"] = roc_auc_score(Y_train, model.predict_proba(X_train_final)[:, 1])
                    table.loc[row, "AUC Test 1"] = roc_auc_score(Y_test1, model.predict_proba(X_test1_final)[:, 1])
                    table.loc[row, "AUC Test 2"] = roc_auc_score(Y_test2, model.predict_proba(X_test2_final)[:, 1])
                    row = row + 1

table
```

Python

We selected this parameters due the Average AUC (Among Train,Test1 and test2) being the highest at 0.9377.

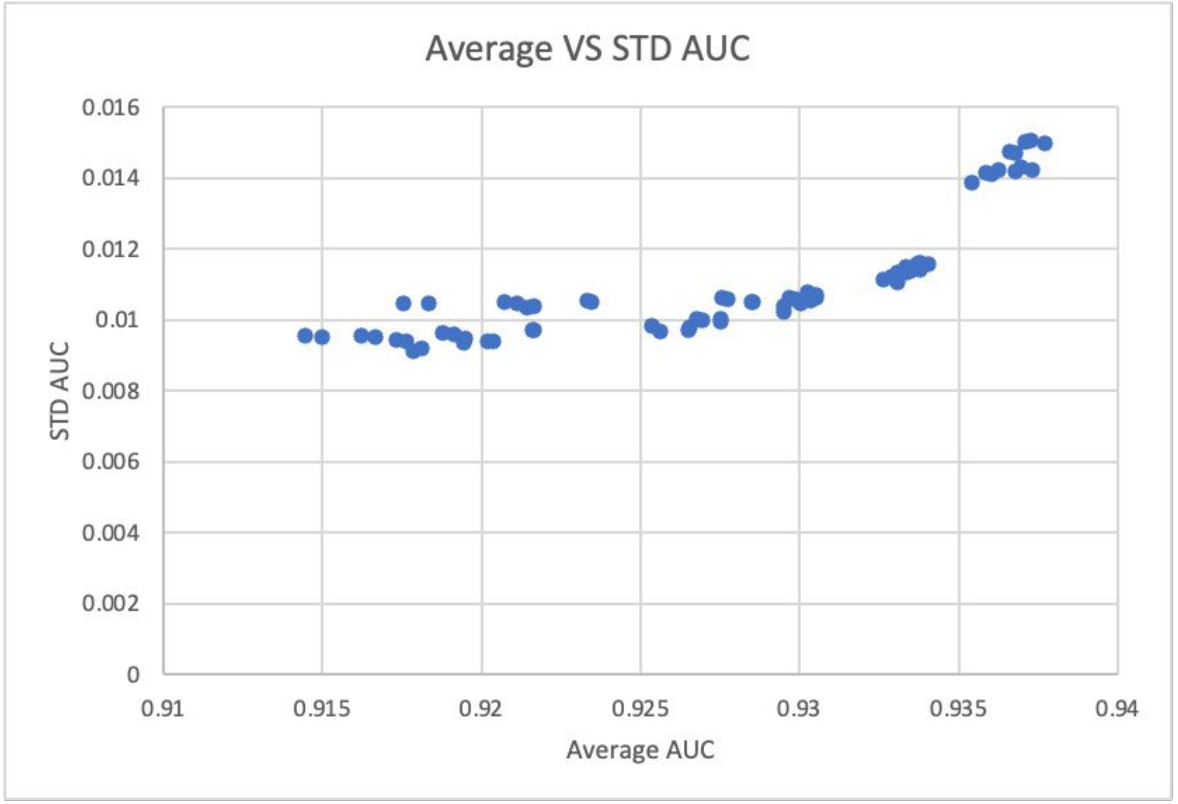
These were the parameters:

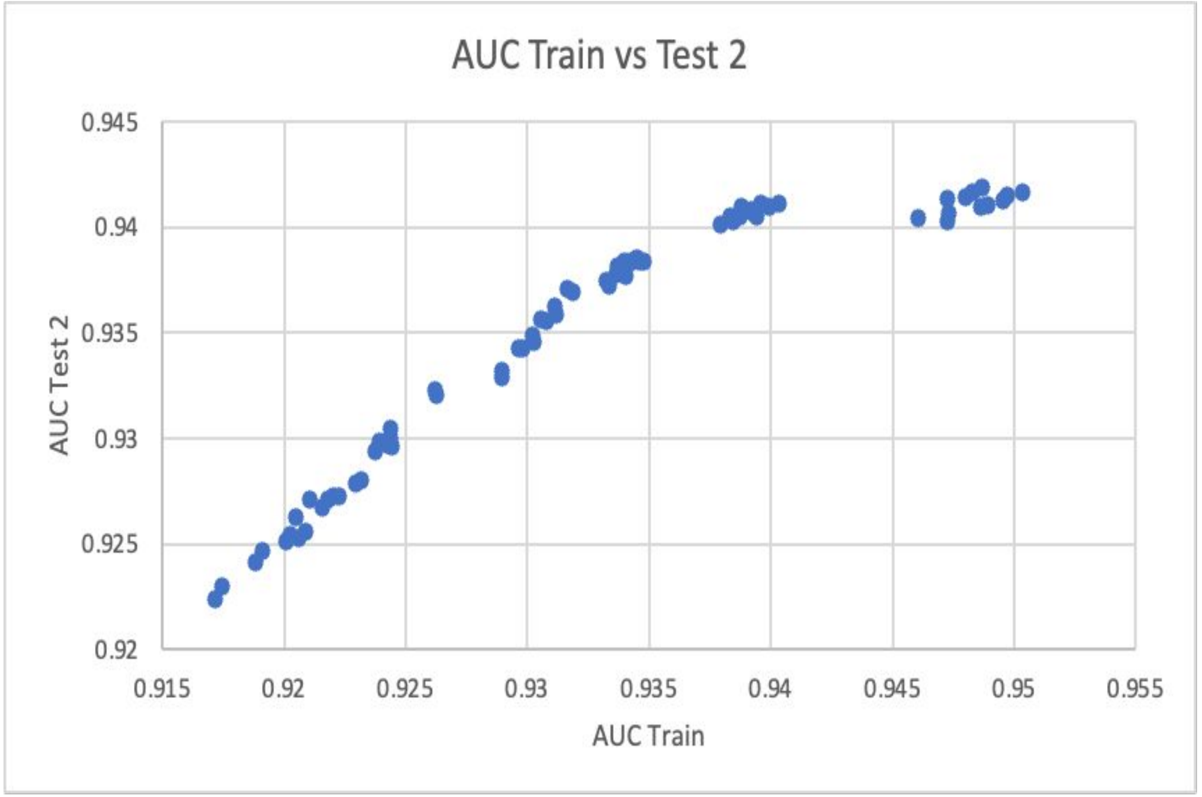
- Number of trees: 300
- Learning rate: 0.1
- Subsample:0.1
- % in each tree: 1
- Default weight: 1

The lessons we learned are :

- More the number of trees doesn't essentially mean better model.
- Learning rate might overfit the model as well.

8. XGBoost - Grid Search

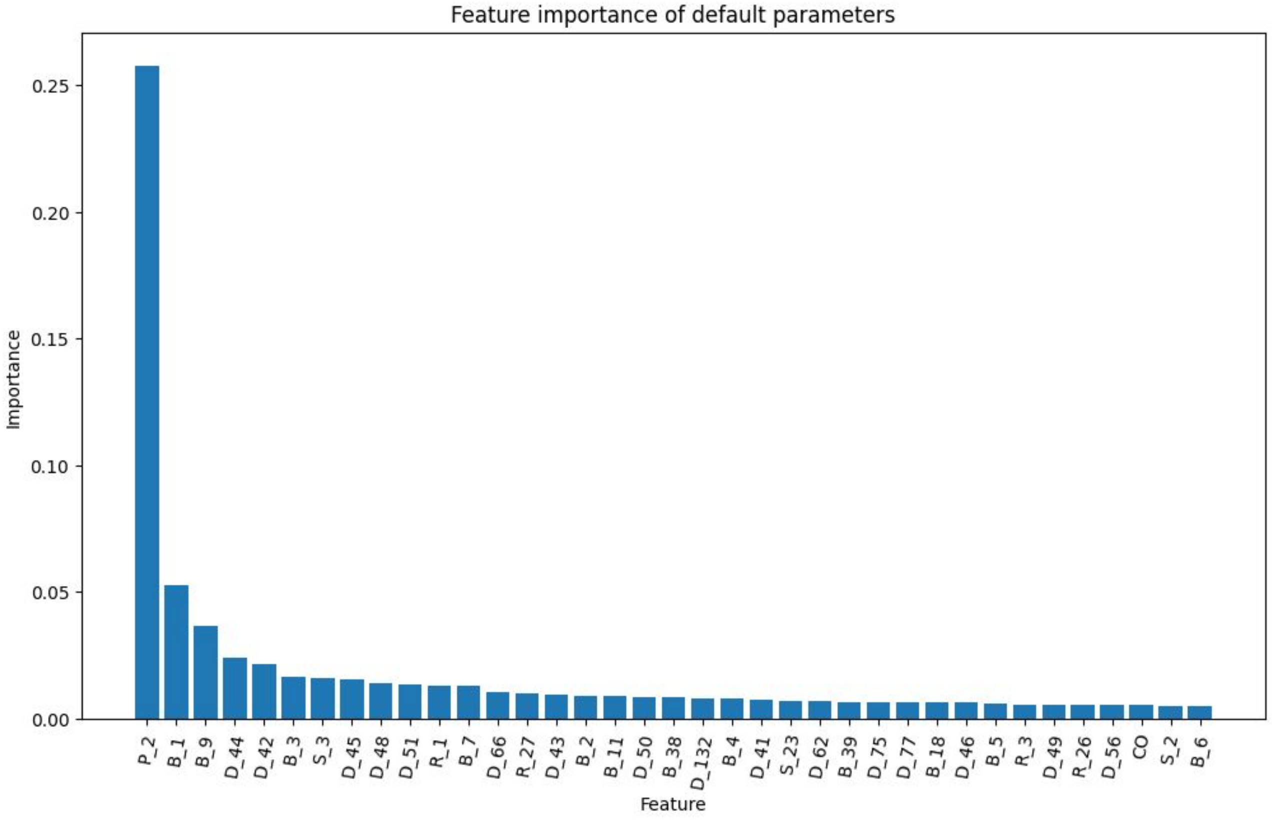


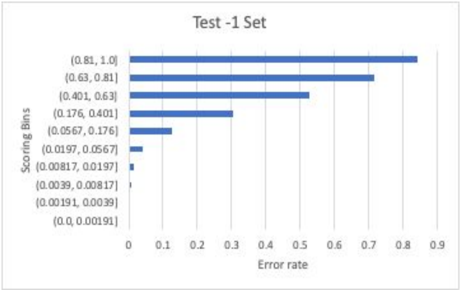


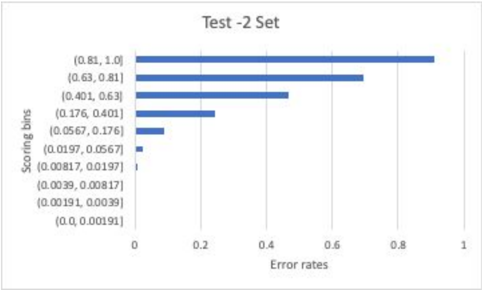
We should be choosing the model with highest AUC and least amount of Standard deviation. It shows the model is best representation of the data.

We see from the second graph that the models perform well in both test and train which signifies accurate representation of relations.

9. XGBoost – Final Model



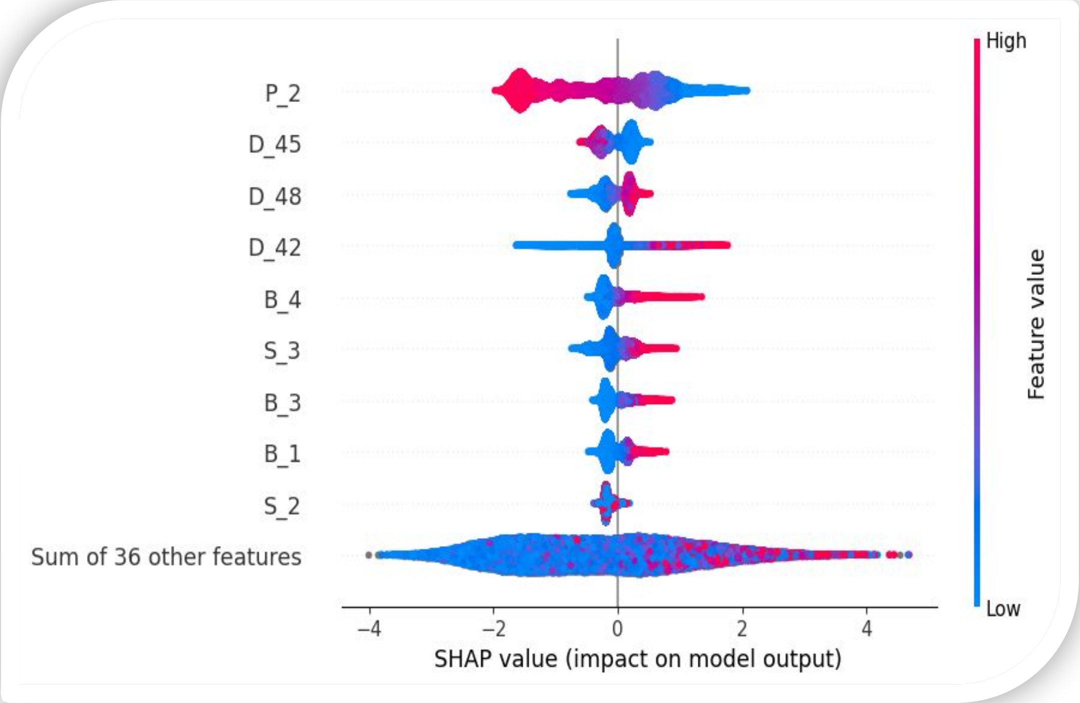




Scores	AUC train	AUC Test1	AUC Test2	Accuracy test 1	Accuracy test 2
Values	0.95	0.92	0.94	0.86	0.87

xgb_instance = xgb.XGBClassifier(n_estimators=300, learning_rate = 0.1, subsample = 0.8, colsample_bytree = 0.5, scale_pos_weight = 1, random_state = 42)

10. XGBoost – SHAP Analysis

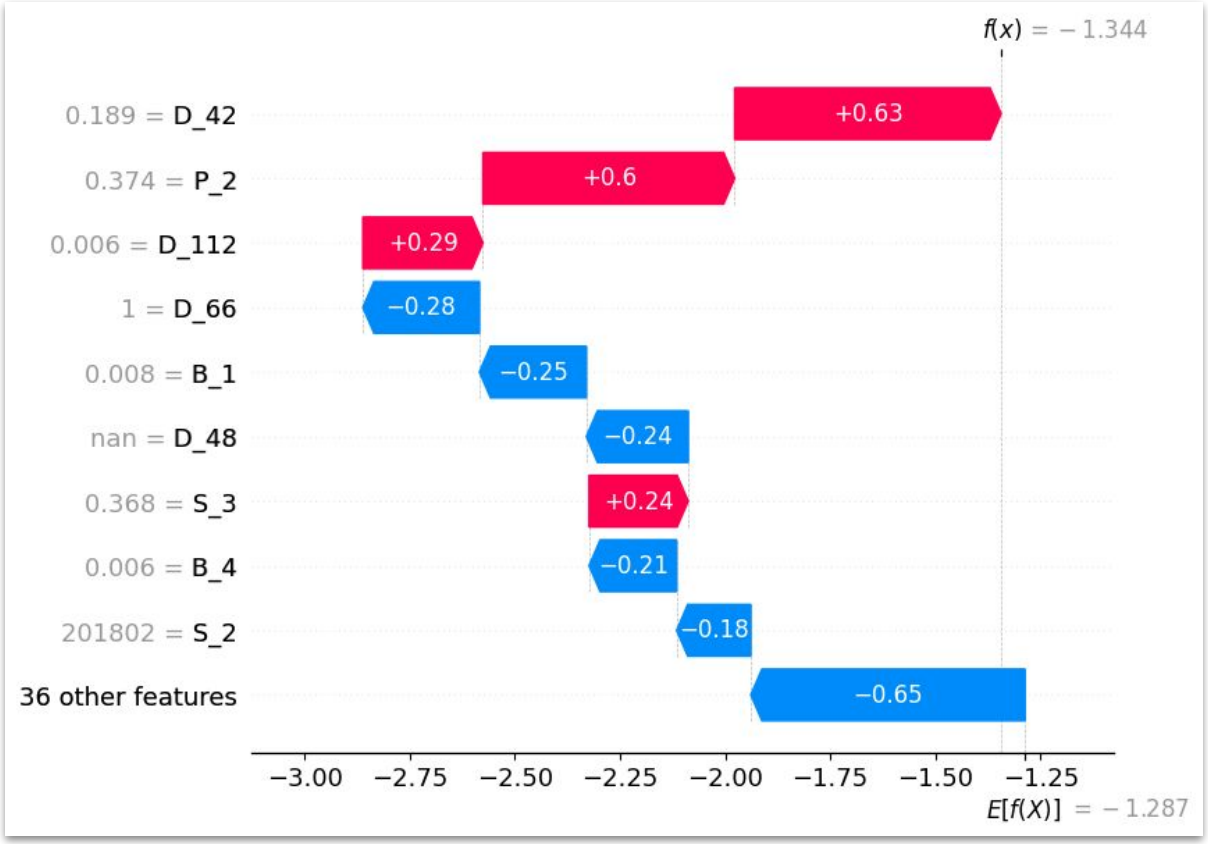



```
shap.plots.beeswarm(shap_values)
```

P_2 It's the most relevant feature and its negatively correlated
With a concentration of observations on the -2 (high value) and 1 (low value)

Delinquency variables:
Has the 3 places after with most of observations shap value close to 0 D45 negative and D48 & D42 with positive correlation

11. XGBoost – SHAP Analysis




```
shap.plots.waterfall(shap_values[123])
```

12. Neural Network – Data Processing

- We removed the top 99% percentile of values after using StandardScaler to scale the data. This ensure we are skewed by outlier data. Alternatives would be using IsolationForest to identify the same.
- Post that we imputed the null values with 0 as the StandardScalar function would normalise the values and adding 0 would add to the mean and wouldn’t skew the distribution.

[8]

```python
arr = ['B_9','S_3','D_48','D_43','D_50','D_132','S_23','D_62','D_77','D_46','B_5','R_3','D_49','R_26','D_56','B_6','B_10','D_61','D_41']
```

Python

[9]

```python
for i in arr:
    X_train_normalized[i] = np.where((X_train_normalized[i] > X_train_normalized[i].quantile(0.99)), X_train_normalized[i].quantile(0.99), X_train_normalized[i])
X_train_normalized['S_23'] = np.where((X_train_normalized['S_23'] < X_train_normalized['S_23'].quantile(0.01)), X_train_normalized['S_23'].quantile(0.01), X_train_normalized['S_23'
X_train_normalized['D_46'] = np.where((X_train_normalized['D_46'] < X_train_normalized['D_46'].quantile(0.01)), X_train_normalized['D_46'].quantile(0.01), X_train_normalized['D_46'
```

Python

```python
X_train_normalized = X_train_normalized.fillna(0)
X_test1_normalized = X_test1_normalized.fillna(0)
X_test2_normalized = X_test2_normalized.fillna(0)
```

Python