## Business Context

Renewable energy sources play an increasingly important role in the global energy mix, as the effort to reduce the environmental impact of energy production increases.

Out of all the renewable energy alternatives, wind energy is one of the most developed technologies worldwide. The U.S Department of Energy has put together a guide to achieving operational efficiency using predictive maintenance practices.

Predictive maintenance uses sensor information and analysis methods to measure and predict degradation and future component capability. The idea behind predictive maintenance is that failure patterns are predictable and if component failure can be predicted accurately and the component is replaced before it fails, the costs of operation and maintenance will be much lower.

The sensors fitted across different machines involved in the process of energy generation collect data related to various environmental factors (temperature, humidity, wind speed, etc.) and additional features related to various parts of the wind turbine (gearbox, tower, blades, break, etc.).

## Objective

“ReneWind” is a company working on improving the machinery/processes involved in the production of wind energy using machine learning and has collected data of generator failure of wind turbines using sensors. They have shared a ciphered version of the data, as the data collected through sensors is confidential (the type of data collected varies with companies). Data has 40 predictors, 20000 observations in the training set and 5000 in the test set.

The objective is to build various classification models, tune them, and find the best one that will help identify failures so that the generators could be repaired before failing/breaking to reduce the overall maintenance cost. The nature of predictions made by the classification model will translate as follows:

* True positives (TP) are failures correctly predicted by the model. These will result in repairing costs.
* False negatives (FN) are real failures where there is no detection by the model. These will result in replacement costs.
* False positives (FP) are detections where there is no failure. These will result in inspection costs. It is given that the cost of repairing a generator is much less than the cost of replacing it, and the cost of inspection is less than the cost of repair.

“1” in the target variables should be considered as “failure” and “0” represents “No failure”.

## Data Description

* The data provided is a transformed version of original data which was collected using sensors.
* Train.csv - To be used for training and tuning of models.
* Test.csv - To be used only for testing the performance of the final best model.
* Both the datasets consist of 40 predictor variables and 1 target variable

# Data manipulation and analysis libraries  
import pandas as pd  
import numpy as np  
  
# Data visualization libraries  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
# Preprocessing and model evaluation libraries  
from sklearn import metrics  
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder  
from sklearn.impute import SimpleImputer  
from sklearn.metrics import (  
 f1\_score, accuracy\_score, recall\_score, precision\_score,  
 confusion\_matrix, roc\_auc\_score  
)  
from sklearn.model\_selection import (  
 train\_test\_split, StratifiedKFold, cross\_val\_score, RandomizedSearchCV, GridSearchCV  
)  
  
# Sampling libraries  
from imblearn.over\_sampling import SMOTE  
from imblearn.under\_sampling import RandomUnderSampler  
  
# Pipeline and transformation libraries  
from sklearn.pipeline import Pipeline  
from sklearn.impute import SimpleImputer  
from sklearn.compose import ColumnTransformer  
  
# Model building libraries  
from sklearn.linear\_model import LogisticRegression  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.ensemble import (  
 AdaBoostClassifier, GradientBoostingClassifier, RandomForestClassifier, BaggingClassifier  
)  
from xgboost import XGBClassifier  
  
# Settings to enhance output readability and suppress warnings  
pd.set\_option("display.max\_columns", None)  
pd.set\_option("display.max\_rows", None)  
pd.set\_option("display.float\_format", lambda x: "%.3f" % x) # Suppress scientific notations in pandas  
  
# Suppress warnings  
import warnings  
warnings.filterwarnings("ignore")

train\_data = pd.read\_csv('./Train.csv')  
test\_data = pd.read\_csv('./Test.csv')

# Data Overview

# View top 5 rows of the data  
train\_data.head()

V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 \  
0 -4.465 -4.679 3.102 0.506 -0.221 -2.033 -2.911 0.051 -1.522 3.762   
1 3.366 3.653 0.910 -1.368 0.332 2.359 0.733 -4.332 0.566 -0.101   
2 -3.832 -5.824 0.634 -2.419 -1.774 1.017 -2.099 -3.173 -2.082 5.393   
3 1.618 1.888 7.046 -1.147 0.083 -1.530 0.207 -2.494 0.345 2.119   
4 -0.111 3.872 -3.758 -2.983 3.793 0.545 0.205 4.849 -1.855 -6.220   
  
 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20 \  
0 -5.715 0.736 0.981 1.418 -3.376 -3.047 0.306 2.914 2.270 4.395   
1 1.914 -0.951 -1.255 -2.707 0.193 -4.769 -2.205 0.908 0.757 -5.834   
2 -0.771 1.107 1.144 0.943 -3.164 -4.248 -4.039 3.689 3.311 1.059   
3 -3.053 0.460 2.705 -0.636 -0.454 -3.174 -3.404 -1.282 1.582 -1.952   
4 1.998 4.724 0.709 -1.989 -2.633 4.184 2.245 3.734 -6.313 -5.380   
  
 V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 \  
0 -2.388 0.646 -1.191 3.133 0.665 -2.511 -0.037 0.726 -3.982 -1.073   
1 -3.065 1.597 -1.757 1.766 -0.267 3.625 1.500 -0.586 0.783 -0.201   
2 -2.143 1.650 -1.661 1.680 -0.451 -4.551 3.739 1.134 -2.034 0.841   
3 -3.517 -1.206 -5.628 -1.818 2.124 5.295 4.748 -2.309 -3.963 -6.029   
4 -0.887 2.062 9.446 4.490 -3.945 4.582 -8.780 -3.383 5.107 6.788   
  
 V31 V32 V33 V34 V35 V36 V37 V38 V39 V40 \  
0 1.667 3.060 -1.690 2.846 2.235 6.667 0.444 -2.369 2.951 -3.480   
1 0.025 -1.795 3.033 -2.468 1.895 -2.298 -1.731 5.909 -0.386 0.616   
2 -1.600 -0.257 0.804 4.086 2.292 5.361 0.352 2.940 3.839 -4.309   
3 4.949 -3.584 -2.577 1.364 0.623 5.550 -1.527 0.139 3.101 -1.277   
4 2.044 8.266 6.629 -10.069 1.223 -3.230 1.687 -2.164 -3.645 6.510   
  
 Target   
0 0   
1 0   
2 0   
3 0   
4 0

# View top 5 rows of the data  
test\_data.head()

V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 \  
0 -0.613 -3.820 2.202 1.300 -1.185 -4.496 -1.836 4.723 1.206 -0.342   
1 0.390 -0.512 0.527 -2.577 -1.017 2.235 -0.441 -4.406 -0.333 1.967   
2 -0.875 -0.641 4.084 -1.590 0.526 -1.958 -0.695 1.347 -1.732 0.466   
3 0.238 1.459 4.015 2.534 1.197 -3.117 -0.924 0.269 1.322 0.702   
4 5.828 2.768 -1.235 2.809 -1.642 -1.407 0.569 0.965 1.918 -2.775   
  
 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20 \  
0 -5.123 1.017 4.819 3.269 -2.984 1.387 2.032 -0.512 -1.023 7.339   
1 1.797 0.410 0.638 -1.390 -1.883 -5.018 -3.827 2.418 1.762 -3.242   
2 -4.928 3.565 -0.449 -0.656 -0.167 -1.630 2.292 2.396 0.601 1.794   
3 -5.578 -0.851 2.591 0.767 -2.391 -2.342 0.572 -0.934 0.509 1.211   
4 -0.530 1.375 -0.651 -1.679 -0.379 -4.443 3.894 -0.608 2.945 0.367   
  
 V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 \  
0 -2.242 0.155 2.054 -2.772 1.851 -1.789 -0.277 -1.255 -3.833 -1.505 1.587   
1 -3.193 1.857 -1.708 0.633 -0.588 0.084 3.014 -0.182 0.224 0.865 -1.782   
2 -2.120 0.482 -0.841 1.790 1.874 0.364 -0.169 -0.484 -2.119 -2.157 2.907   
3 -3.260 0.105 -0.659 1.498 1.100 4.143 -0.248 -1.137 -5.356 -4.546 3.809   
4 -5.789 4.598 4.450 3.225 0.397 0.248 -2.362 1.079 -0.473 2.243 -3.591   
  
 V32 V33 V34 V35 V36 V37 V38 V39 V40 Target   
0 2.291 -5.411 0.870 0.574 4.157 1.428 -10.511 0.455 -1.448 0   
1 -2.475 2.494 0.315 2.059 0.684 -0.485 5.128 1.721 -1.488 0   
2 -1.319 -2.997 0.460 0.620 5.632 1.324 -1.752 1.808 1.676 0   
3 3.518 -3.074 -0.284 0.955 3.029 -1.367 -3.412 0.906 -2.451 0   
4 1.774 -1.502 -2.227 4.777 -6.560 -0.806 -0.276 -3.858 -0.538 0

# Check the dimensions of the data  
train\_data.shape

(20000, 41)

# Check the dimensions of the data  
test\_data.shape

(5000, 41)

# Check data types  
train\_data.info()

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 20000 entries, 0 to 19999  
Data columns (total 41 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 V1 19982 non-null float64  
 1 V2 19982 non-null float64  
 2 V3 20000 non-null float64  
 3 V4 20000 non-null float64  
 4 V5 20000 non-null float64  
 5 V6 20000 non-null float64  
 6 V7 20000 non-null float64  
 7 V8 20000 non-null float64  
 8 V9 20000 non-null float64  
 9 V10 20000 non-null float64  
 10 V11 20000 non-null float64  
 11 V12 20000 non-null float64  
 12 V13 20000 non-null float64  
 13 V14 20000 non-null float64  
 14 V15 20000 non-null float64  
 15 V16 20000 non-null float64  
 16 V17 20000 non-null float64  
 17 V18 20000 non-null float64  
 18 V19 20000 non-null float64  
 19 V20 20000 non-null float64  
 20 V21 20000 non-null float64  
 21 V22 20000 non-null float64  
 22 V23 20000 non-null float64  
 23 V24 20000 non-null float64  
 24 V25 20000 non-null float64  
 25 V26 20000 non-null float64  
 26 V27 20000 non-null float64  
 27 V28 20000 non-null float64  
 28 V29 20000 non-null float64  
 29 V30 20000 non-null float64  
 30 V31 20000 non-null float64  
 31 V32 20000 non-null float64  
 32 V33 20000 non-null float64  
 33 V34 20000 non-null float64  
 34 V35 20000 non-null float64  
 35 V36 20000 non-null float64  
 36 V37 20000 non-null float64  
 37 V38 20000 non-null float64  
 38 V39 20000 non-null float64  
 39 V40 20000 non-null float64  
 40 Target 20000 non-null int64   
dtypes: float64(40), int64(1)  
memory usage: 6.3 MB

# Check data types  
test\_data.info()

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5000 entries, 0 to 4999  
Data columns (total 41 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 V1 4995 non-null float64  
 1 V2 4994 non-null float64  
 2 V3 5000 non-null float64  
 3 V4 5000 non-null float64  
 4 V5 5000 non-null float64  
 5 V6 5000 non-null float64  
 6 V7 5000 non-null float64  
 7 V8 5000 non-null float64  
 8 V9 5000 non-null float64  
 9 V10 5000 non-null float64  
 10 V11 5000 non-null float64  
 11 V12 5000 non-null float64  
 12 V13 5000 non-null float64  
 13 V14 5000 non-null float64  
 14 V15 5000 non-null float64  
 15 V16 5000 non-null float64  
 16 V17 5000 non-null float64  
 17 V18 5000 non-null float64  
 18 V19 5000 non-null float64  
 19 V20 5000 non-null float64  
 20 V21 5000 non-null float64  
 21 V22 5000 non-null float64  
 22 V23 5000 non-null float64  
 23 V24 5000 non-null float64  
 24 V25 5000 non-null float64  
 25 V26 5000 non-null float64  
 26 V27 5000 non-null float64  
 27 V28 5000 non-null float64  
 28 V29 5000 non-null float64  
 29 V30 5000 non-null float64  
 30 V31 5000 non-null float64  
 31 V32 5000 non-null float64  
 32 V33 5000 non-null float64  
 33 V34 5000 non-null float64  
 34 V35 5000 non-null float64  
 35 V36 5000 non-null float64  
 36 V37 5000 non-null float64  
 37 V38 5000 non-null float64  
 38 V39 5000 non-null float64  
 39 V40 5000 non-null float64  
 40 Target 5000 non-null int64   
dtypes: float64(40), int64(1)  
memory usage: 1.6 MB

# Double check for null values per column  
train\_data.isnull().sum()

V1 18  
V2 18  
V3 0  
V4 0  
V5 0  
V6 0  
V7 0  
V8 0  
V9 0  
V10 0  
V11 0  
V12 0  
V13 0  
V14 0  
V15 0  
V16 0  
V17 0  
V18 0  
V19 0  
V20 0  
V21 0  
V22 0  
V23 0  
V24 0  
V25 0  
V26 0  
V27 0  
V28 0  
V29 0  
V30 0  
V31 0  
V32 0  
V33 0  
V34 0  
V35 0  
V36 0  
V37 0  
V38 0  
V39 0  
V40 0  
Target 0  
dtype: int64

# Checking for null values per column  
test\_data.isnull().sum()

V1 5  
V2 6  
V3 0  
V4 0  
V5 0  
V6 0  
V7 0  
V8 0  
V9 0  
V10 0  
V11 0  
V12 0  
V13 0  
V14 0  
V15 0  
V16 0  
V17 0  
V18 0  
V19 0  
V20 0  
V21 0  
V22 0  
V23 0  
V24 0  
V25 0  
V26 0  
V27 0  
V28 0  
V29 0  
V30 0  
V31 0  
V32 0  
V33 0  
V34 0  
V35 0  
V36 0  
V37 0  
V38 0  
V39 0  
V40 0  
Target 0  
dtype: int64

# Check data duplicate values  
train\_data.duplicated().sum()

np.int64(0)

# Check data duplicate values  
test\_data.duplicated().sum()

np.int64(0)

train\_data.describe().T

count mean std min 25% 50% 75% max  
V1 19982.000 -0.272 3.442 -11.876 -2.737 -0.748 1.840 15.493  
V2 19982.000 0.440 3.151 -12.320 -1.641 0.472 2.544 13.089  
V3 20000.000 2.485 3.389 -10.708 0.207 2.256 4.566 17.091  
V4 20000.000 -0.083 3.432 -15.082 -2.348 -0.135 2.131 13.236  
V5 20000.000 -0.054 2.105 -8.603 -1.536 -0.102 1.340 8.134  
V6 20000.000 -0.995 2.041 -10.227 -2.347 -1.001 0.380 6.976  
V7 20000.000 -0.879 1.762 -7.950 -2.031 -0.917 0.224 8.006  
V8 20000.000 -0.548 3.296 -15.658 -2.643 -0.389 1.723 11.679  
V9 20000.000 -0.017 2.161 -8.596 -1.495 -0.068 1.409 8.138  
V10 20000.000 -0.013 2.193 -9.854 -1.411 0.101 1.477 8.108  
V11 20000.000 -1.895 3.124 -14.832 -3.922 -1.921 0.119 11.826  
V12 20000.000 1.605 2.930 -12.948 -0.397 1.508 3.571 15.081  
V13 20000.000 1.580 2.875 -13.228 -0.224 1.637 3.460 15.420  
V14 20000.000 -0.951 1.790 -7.739 -2.171 -0.957 0.271 5.671  
V15 20000.000 -2.415 3.355 -16.417 -4.415 -2.383 -0.359 12.246  
V16 20000.000 -2.925 4.222 -20.374 -5.634 -2.683 -0.095 13.583  
V17 20000.000 -0.134 3.345 -14.091 -2.216 -0.015 2.069 16.756  
V18 20000.000 1.189 2.592 -11.644 -0.404 0.883 2.572 13.180  
V19 20000.000 1.182 3.397 -13.492 -1.050 1.279 3.493 13.238  
V20 20000.000 0.024 3.669 -13.923 -2.433 0.033 2.512 16.052  
V21 20000.000 -3.611 3.568 -17.956 -5.930 -3.533 -1.266 13.840  
V22 20000.000 0.952 1.652 -10.122 -0.118 0.975 2.026 7.410  
V23 20000.000 -0.366 4.032 -14.866 -3.099 -0.262 2.452 14.459  
V24 20000.000 1.134 3.912 -16.387 -1.468 0.969 3.546 17.163  
V25 20000.000 -0.002 2.017 -8.228 -1.365 0.025 1.397 8.223  
V26 20000.000 1.874 3.435 -11.834 -0.338 1.951 4.130 16.836  
V27 20000.000 -0.612 4.369 -14.905 -3.652 -0.885 2.189 17.560  
V28 20000.000 -0.883 1.918 -9.269 -2.171 -0.891 0.376 6.528  
V29 20000.000 -0.986 2.684 -12.579 -2.787 -1.176 0.630 10.722  
V30 20000.000 -0.016 3.005 -14.796 -1.867 0.184 2.036 12.506  
V31 20000.000 0.487 3.461 -13.723 -1.818 0.490 2.731 17.255  
V32 20000.000 0.304 5.500 -19.877 -3.420 0.052 3.762 23.633  
V33 20000.000 0.050 3.575 -16.898 -2.243 -0.066 2.255 16.692  
V34 20000.000 -0.463 3.184 -17.985 -2.137 -0.255 1.437 14.358  
V35 20000.000 2.230 2.937 -15.350 0.336 2.099 4.064 15.291  
V36 20000.000 1.515 3.801 -14.833 -0.944 1.567 3.984 19.330  
V37 20000.000 0.011 1.788 -5.478 -1.256 -0.128 1.176 7.467  
V38 20000.000 -0.344 3.948 -17.375 -2.988 -0.317 2.279 15.290  
V39 20000.000 0.891 1.753 -6.439 -0.272 0.919 2.058 7.760  
V40 20000.000 -0.876 3.012 -11.024 -2.940 -0.921 1.120 10.654  
Target 20000.000 0.056 0.229 0.000 0.000 0.000 0.000 1.000

test\_data.describe().T

count mean std min 25% 50% 75% max  
V1 4995.000 -0.278 3.466 -12.382 -2.744 -0.765 1.831 13.504  
V2 4994.000 0.398 3.140 -10.716 -1.649 0.427 2.444 14.079  
V3 5000.000 2.552 3.327 -9.238 0.315 2.260 4.587 15.315  
V4 5000.000 -0.049 3.414 -14.682 -2.293 -0.146 2.166 12.140  
V5 5000.000 -0.080 2.111 -7.712 -1.615 -0.132 1.341 7.673  
V6 5000.000 -1.042 2.005 -8.924 -2.369 -1.049 0.308 5.068  
V7 5000.000 -0.908 1.769 -8.124 -2.054 -0.940 0.212 7.616  
V8 5000.000 -0.575 3.332 -12.253 -2.642 -0.358 1.713 10.415  
V9 5000.000 0.030 2.174 -6.785 -1.456 -0.080 1.450 8.851  
V10 5000.000 0.019 2.145 -8.171 -1.353 0.166 1.511 6.599  
V11 5000.000 -2.009 3.112 -13.152 -4.050 -2.043 0.044 9.956  
V12 5000.000 1.576 2.907 -8.164 -0.450 1.488 3.563 12.984  
V13 5000.000 1.622 2.883 -11.548 -0.126 1.719 3.465 12.620  
V14 5000.000 -0.921 1.803 -7.814 -2.111 -0.896 0.272 5.734  
V15 5000.000 -2.452 3.387 -15.286 -4.479 -2.417 -0.433 11.673  
V16 5000.000 -3.019 4.264 -20.986 -5.648 -2.774 -0.178 13.976  
V17 5000.000 -0.104 3.337 -13.418 -2.228 0.047 2.112 19.777  
V18 5000.000 1.196 2.586 -12.214 -0.409 0.881 2.604 13.642  
V19 5000.000 1.210 3.385 -14.170 -1.026 1.296 3.526 12.428  
V20 5000.000 0.138 3.657 -13.720 -2.325 0.193 2.540 13.871  
V21 5000.000 -3.664 3.578 -16.341 -5.944 -3.663 -1.330 11.047  
V22 5000.000 0.962 1.640 -6.740 -0.048 0.986 2.029 7.505  
V23 5000.000 -0.422 4.057 -14.422 -3.163 -0.279 2.426 13.181  
V24 5000.000 1.089 3.968 -12.316 -1.623 0.913 3.537 17.806  
V25 5000.000 0.061 2.010 -6.770 -1.298 0.077 1.428 6.557  
V26 5000.000 1.847 3.400 -11.414 -0.242 1.917 4.156 17.528  
V27 5000.000 -0.552 4.403 -13.177 -3.663 -0.872 2.247 17.290  
V28 5000.000 -0.868 1.926 -7.933 -2.160 -0.931 0.421 7.416  
V29 5000.000 -1.096 2.655 -9.988 -2.861 -1.341 0.522 14.039  
V30 5000.000 -0.119 3.023 -12.438 -1.997 0.112 1.946 10.315  
V31 5000.000 0.469 3.446 -11.263 -1.822 0.486 2.779 12.559  
V32 5000.000 0.233 5.586 -17.244 -3.556 -0.077 3.752 26.539  
V33 5000.000 -0.080 3.539 -14.904 -2.348 -0.160 2.099 13.324  
V34 5000.000 -0.393 3.166 -14.700 -2.010 -0.172 1.465 12.146  
V35 5000.000 2.211 2.948 -12.261 0.322 2.112 4.032 13.489  
V36 5000.000 1.595 3.775 -12.736 -0.866 1.703 4.104 17.116  
V37 5000.000 0.023 1.785 -5.079 -1.241 -0.110 1.238 6.810  
V38 5000.000 -0.406 3.969 -15.335 -2.984 -0.381 2.288 13.065  
V39 5000.000 0.939 1.717 -5.451 -0.208 0.959 2.131 7.182  
V40 5000.000 -0.932 2.978 -10.076 -2.987 -1.003 1.080 8.698  
Target 5000.000 0.056 0.231 0.000 0.000 0.000 0.000 1.000

# Check for class distribution of the target variable by looking at the count of observations for each class  
class\_distribution = train\_data['Target'].value\_counts(normalize=True) \* 100  
print(class\_distribution)

Target  
0 94.450  
1 5.550  
Name: proportion, dtype: float64

# Check for class distribution of the target variable by looking at the count of observations for each class  
class\_distribution = test\_data['Target'].value\_counts(normalize=True) \* 100  
print(class\_distribution)

Target  
0 94.360  
1 5.640  
Name: proportion, dtype: float64

Observations:

**Dataset Size**:

* The training set contains 20,000 observations, each with 41 columns (40 predictors and 1 target variable).
* The test set includes 5,000 observations, also with 41 columns.

**Data Types**:

* Both datasets primarily consist of numerical features (float64), with the target variable being an integer (int64).

**Missing Values**:

* In the training set, two variables (V1 and V2) have 18 missing values each.
* In the test set, V1 has 5 missing values, and V2 has 6 missing values.
* No missing values are present in the target variable of both datasets.

**Data Integrity**:

* There are no duplicate rows in either the training or the test dataset, indicating good data integrity.

**Class Imbalance**: Bot training and testing target variables have imbalanced distribution of the target variables classes where:

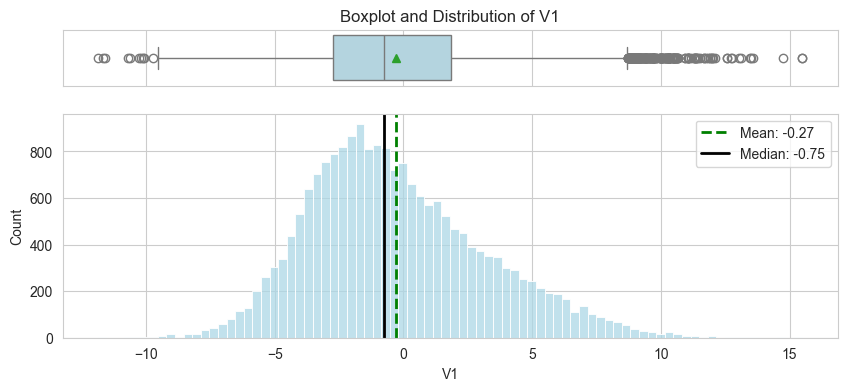
* About 94% of the observations belong to the negative class (0), which represents "Non-failure" cases.
* About 5% of the observations belong to the positive class (1), representing "Failure" cases.

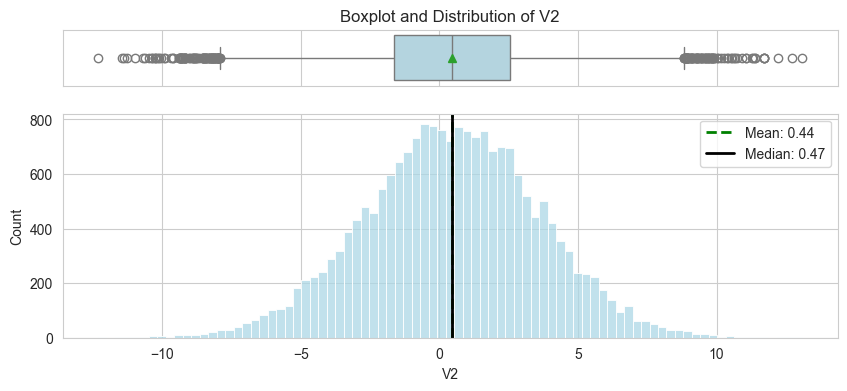
# Exploratory Data analysis (EDA)

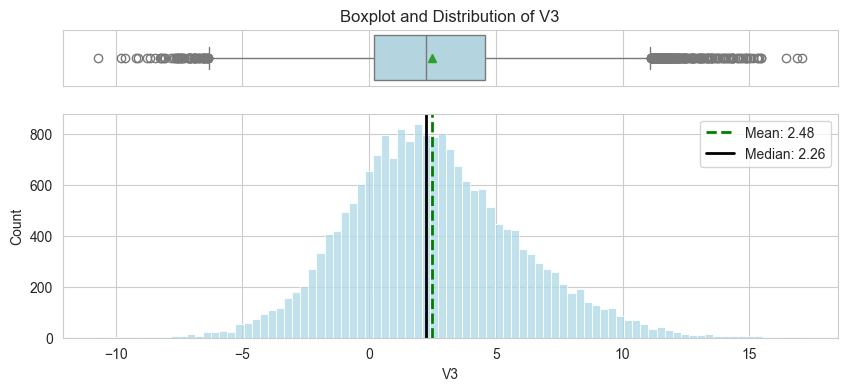
## Univariate Analysis

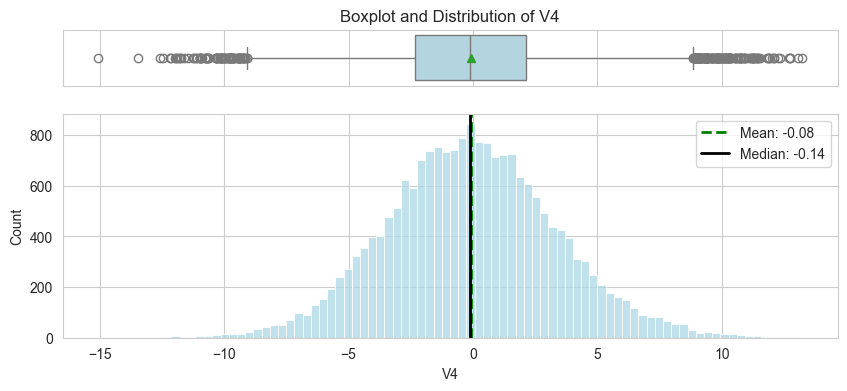
def hist\_and\_boxplot(data, variable, figsize=(12, 4), kde=False, bins=None):  
 """  
 Creates a plot with both a histogram and boxplot for a specified numerical variable.  
  
 Args:  
 - data: The DataFrame containing the data.  
 - variable: Column name of the numerical variable (feature) to be plotted.  
 - figsize: A tuple representing the size of the figure.  
 - density\_curve: A boolean indicating whether to overlay a density curve curve on the histogram.  
 - bins: An integer representing the number of bins for the histogram, or None for automatic bin size.  
  
 Returns:  
 None  
 """  
 # Set up the matplotlib figure with two rows and one column  
 fig, (ax1, ax2) = plt.subplots(2, 1, figsize=figsize, sharex=True, gridspec\_kw={'height\_ratios': [0.2, 0.8]})  
  
 # Plot the boxplot on the first row  
 sns.boxplot(x=variable, data=data, ax=ax1, showmeans=True, color="lightblue")  
 ax1.set(xlabel='', title=f'Boxplot and Distribution of {variable}')  
  
 # Plot the histogram on the second row  
 if bins:  
 sns.histplot(data[variable], kde=kde, bins=bins, ax=ax2, color="lightblue")  
 else:  
 sns.histplot(data[variable], kde=kde, ax=ax2, color="lightblue")  
  
 # Draw lines for mean and median  
 mean\_val = data[variable].mean()  
 median\_val = data[variable].median()  
 ax2.axvline(mean\_val, color='green', linestyle='--', linewidth=2, label=f'Mean: {mean\_val:.2f}')  
 ax2.axvline(median\_val, color='black', linestyle='-', linewidth=2, label=f'Median: {median\_val:.2f}')  
  
 # Add legend to the histogram  
 ax2.legend()  
  
 plt.show()

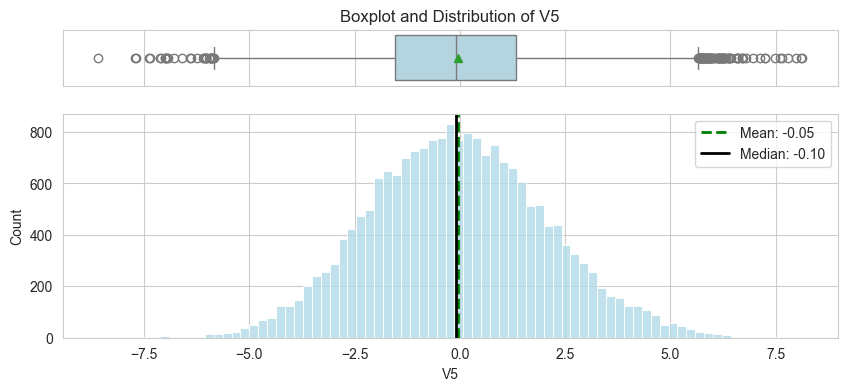
# Looping through the variables in the train dataset and creating a histogram and boxplot for each variable  
for variable in train\_data.columns:  
 hist\_and\_boxplot(train\_data, variable, figsize=(10, 4), kde=False, bins=None)

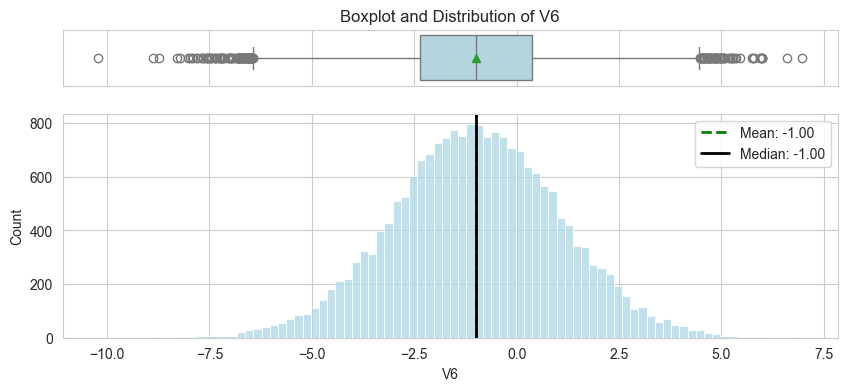


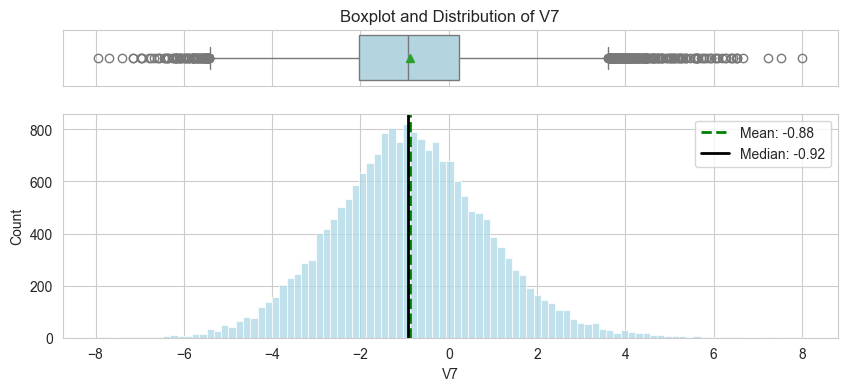


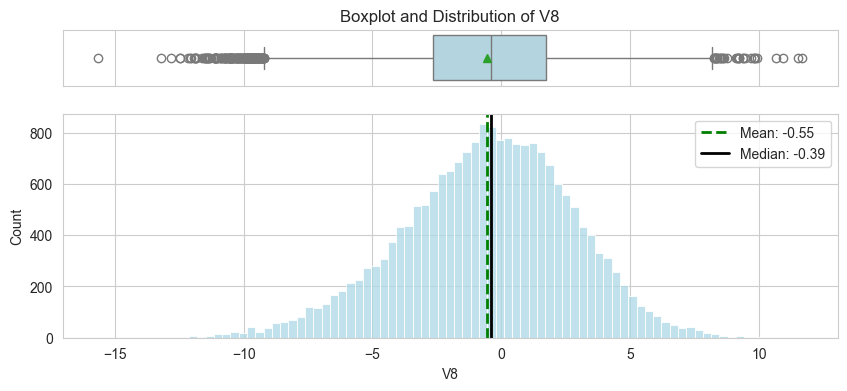


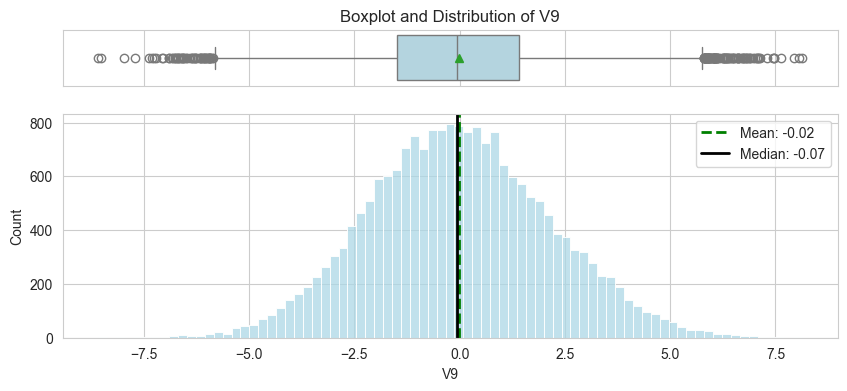


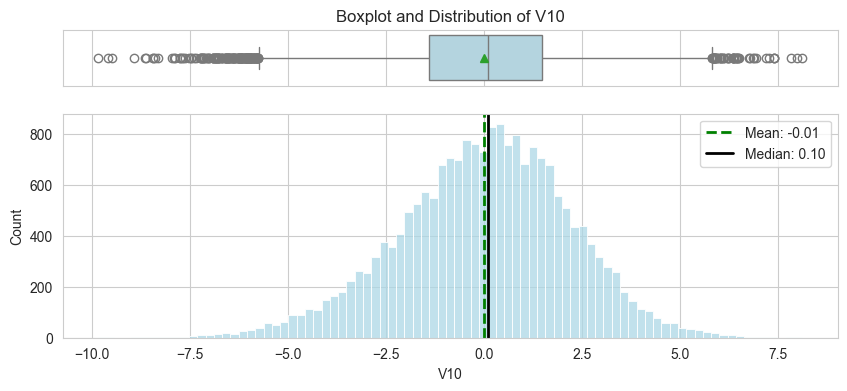


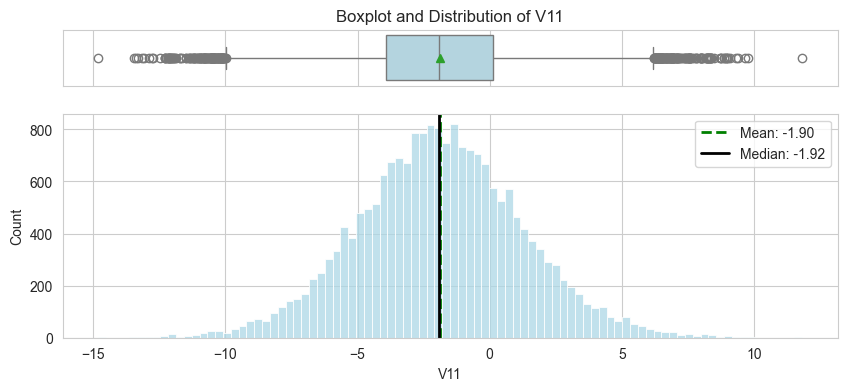


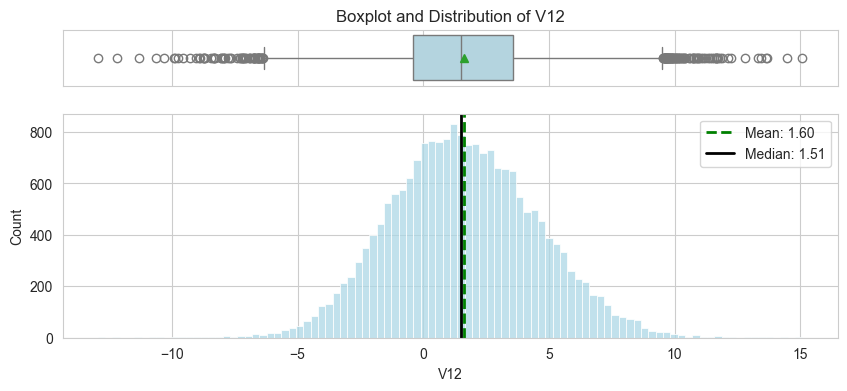


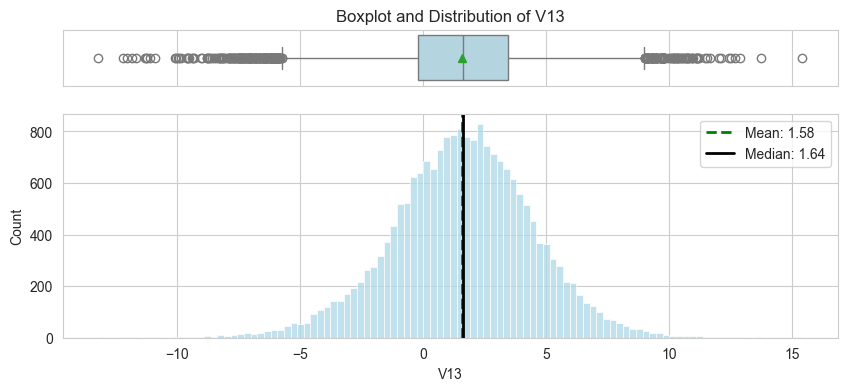


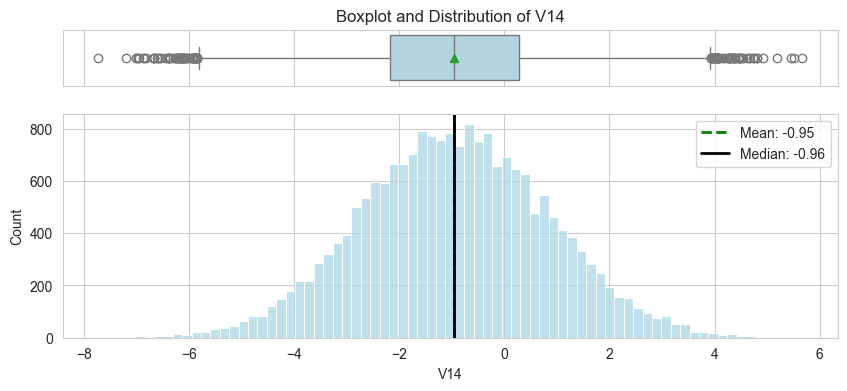


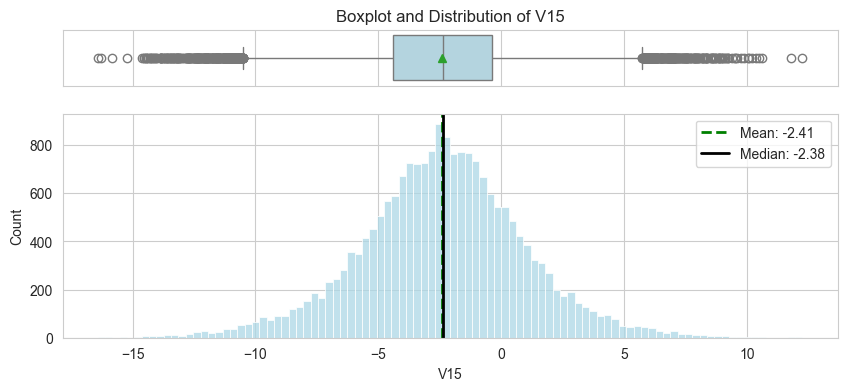


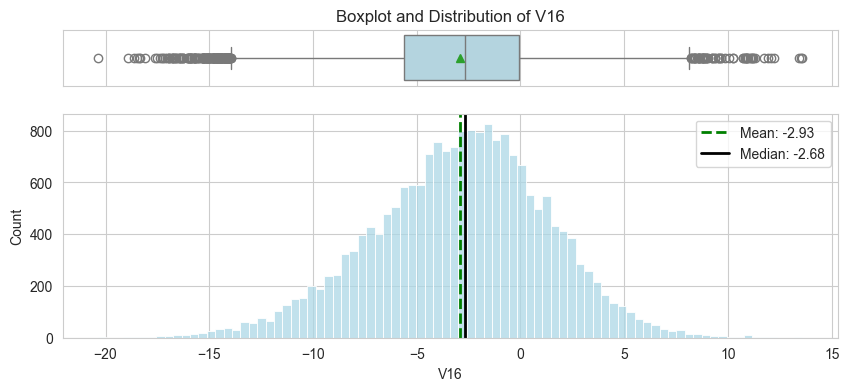


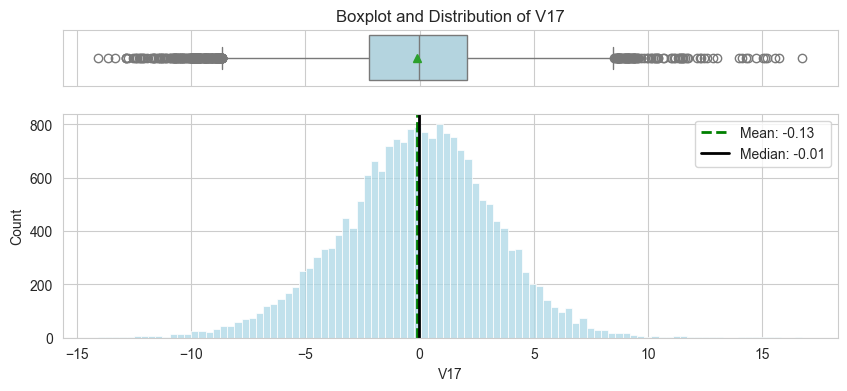


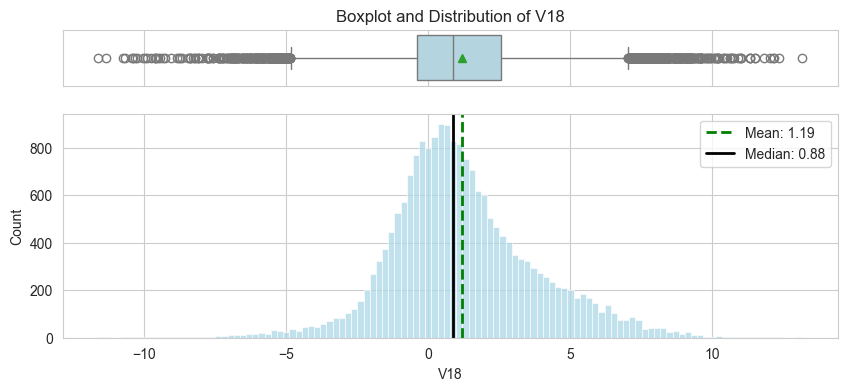


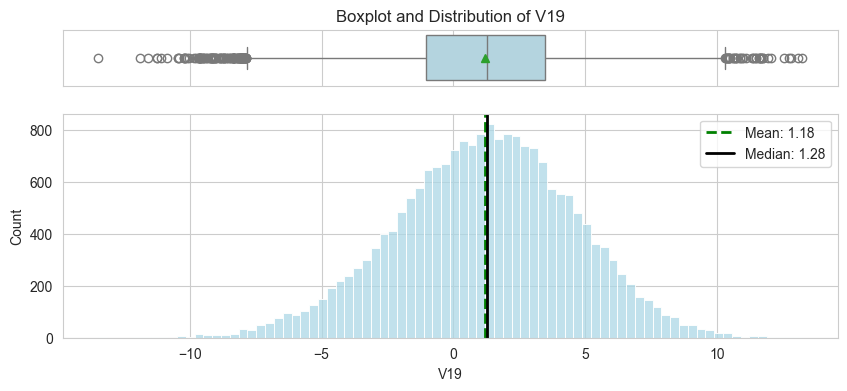


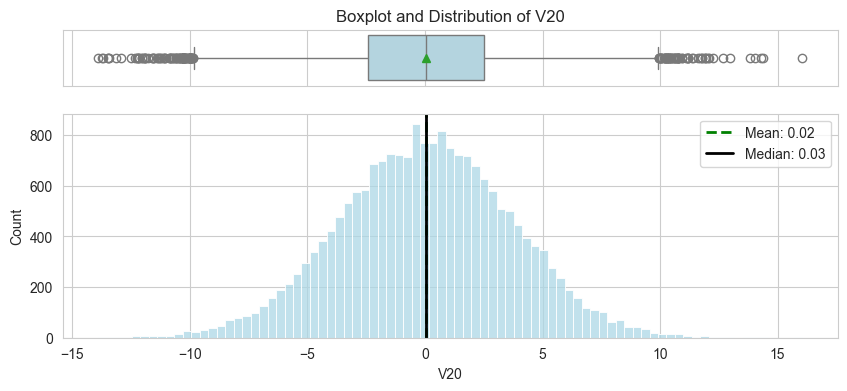


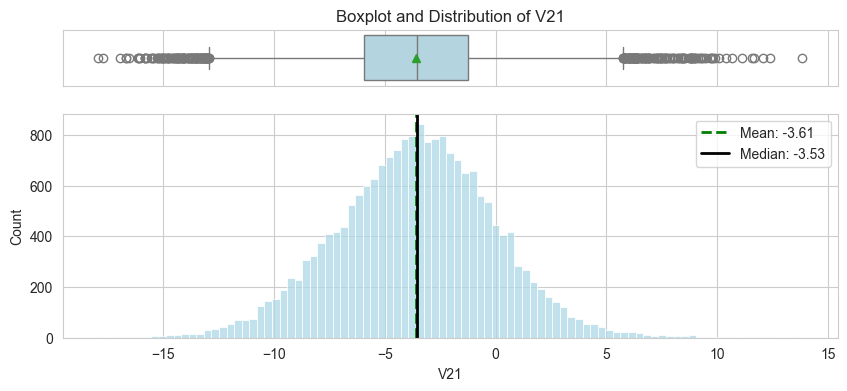


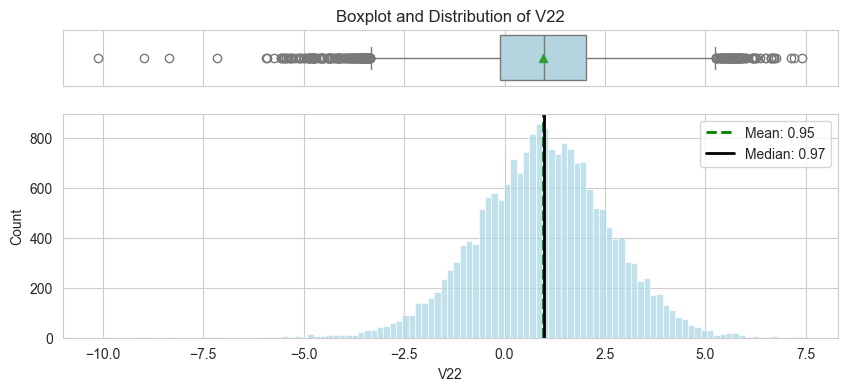


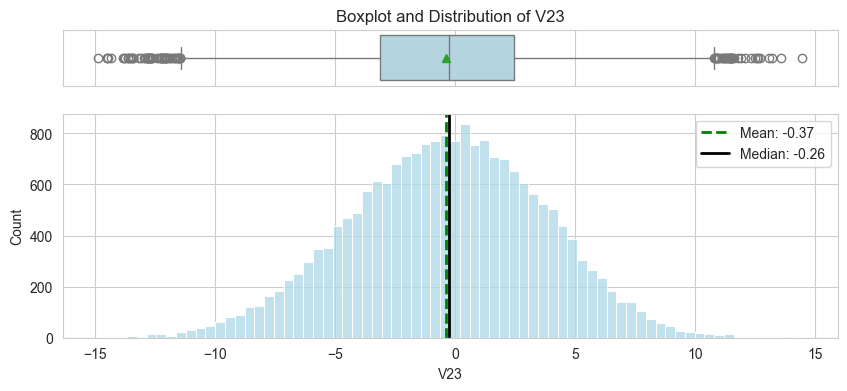


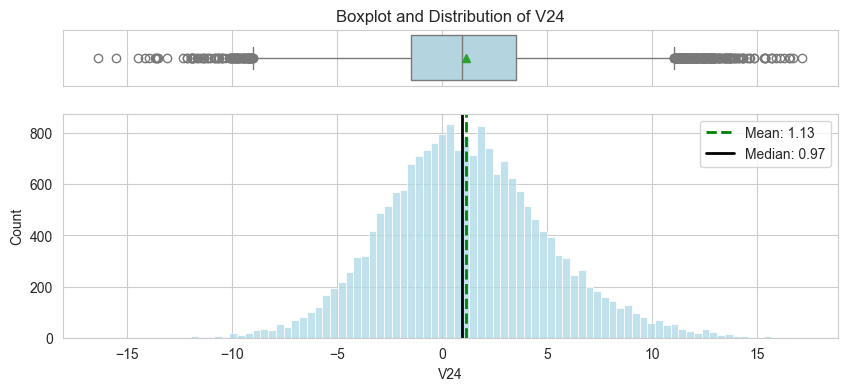


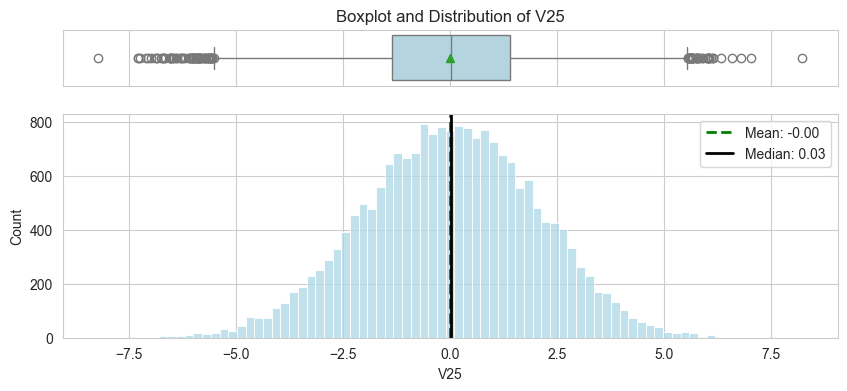


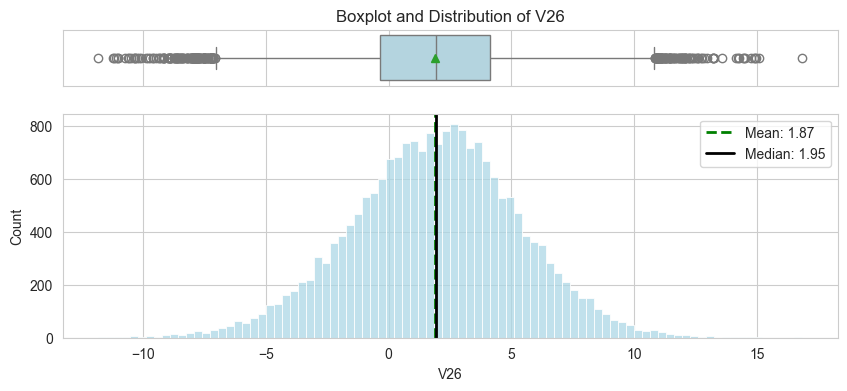


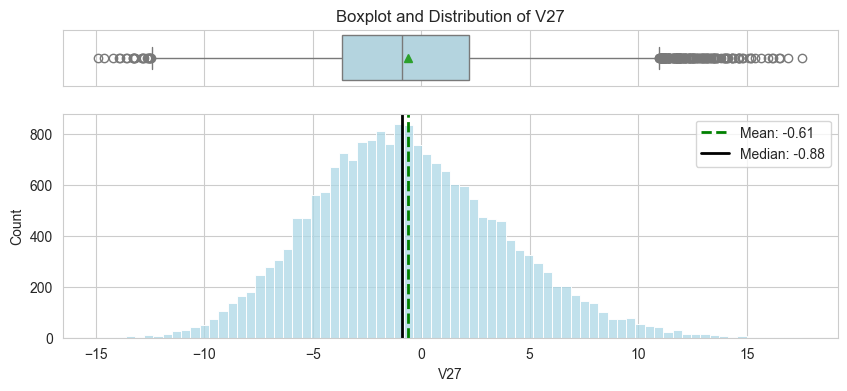


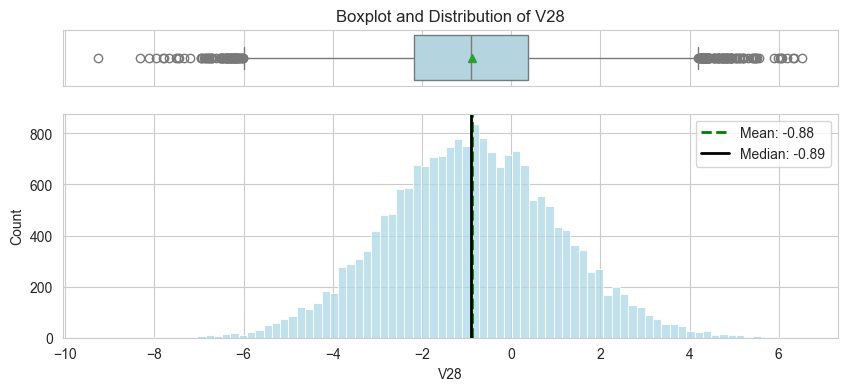


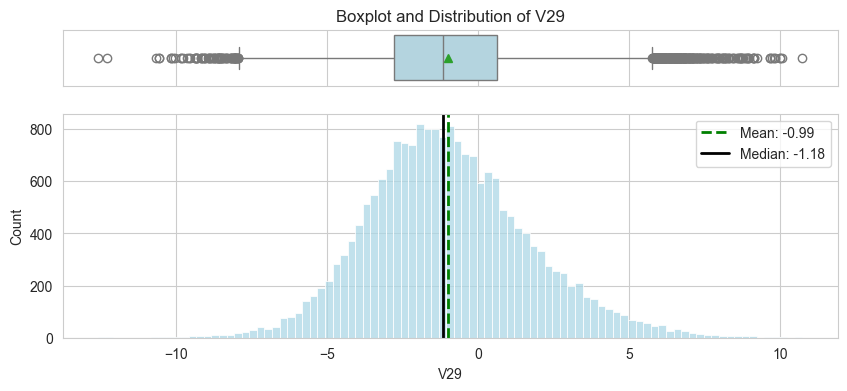


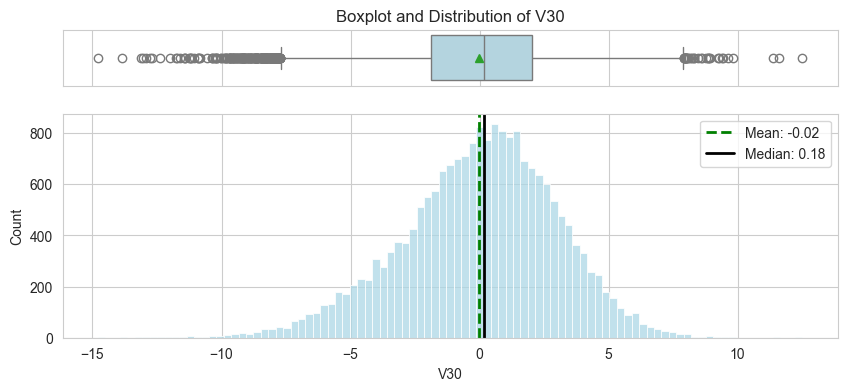


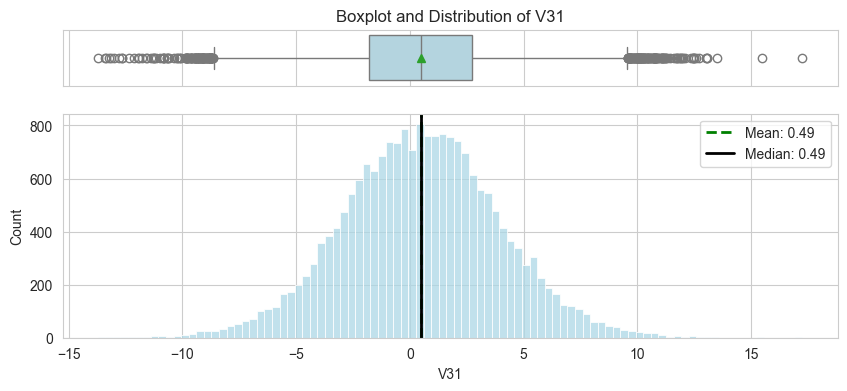


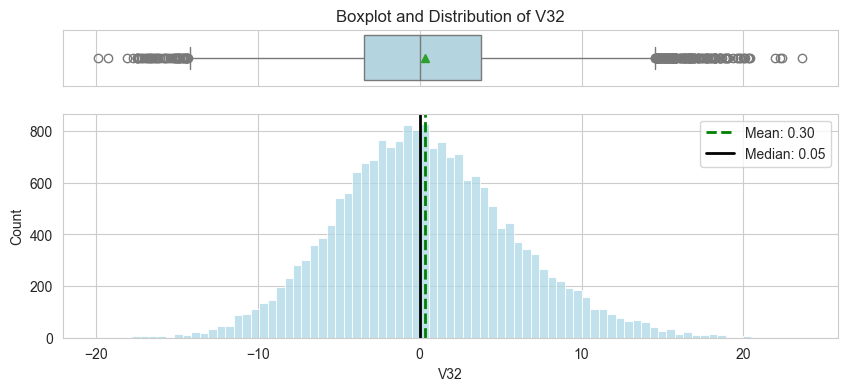


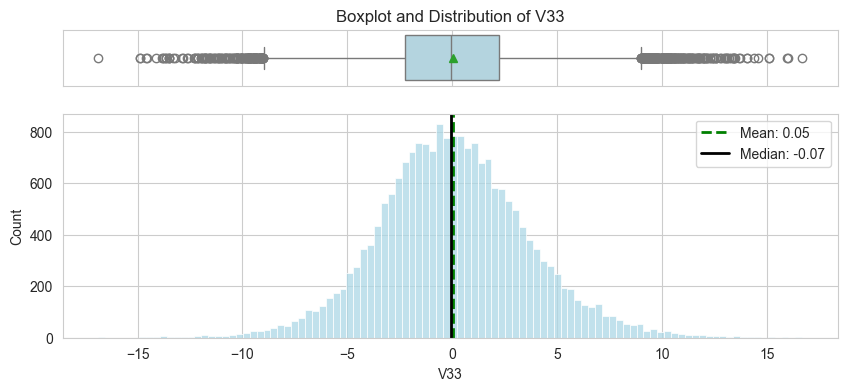


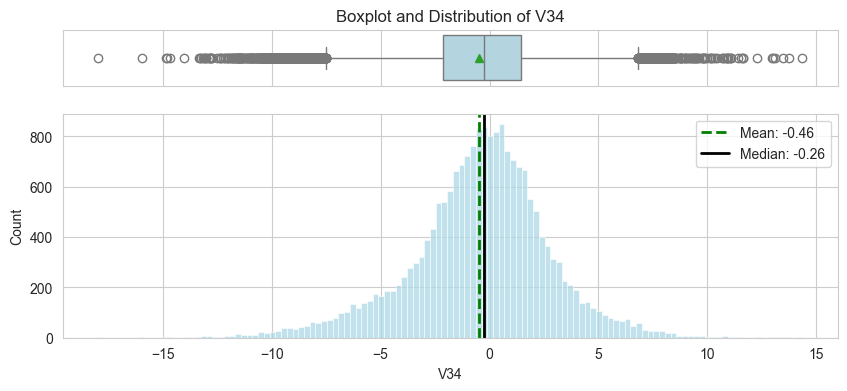


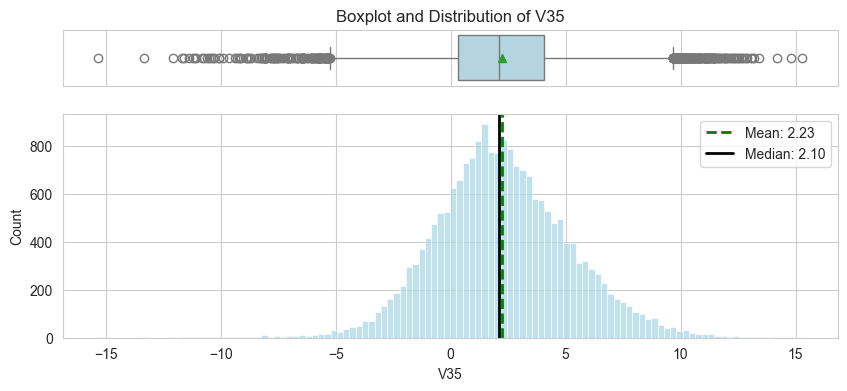


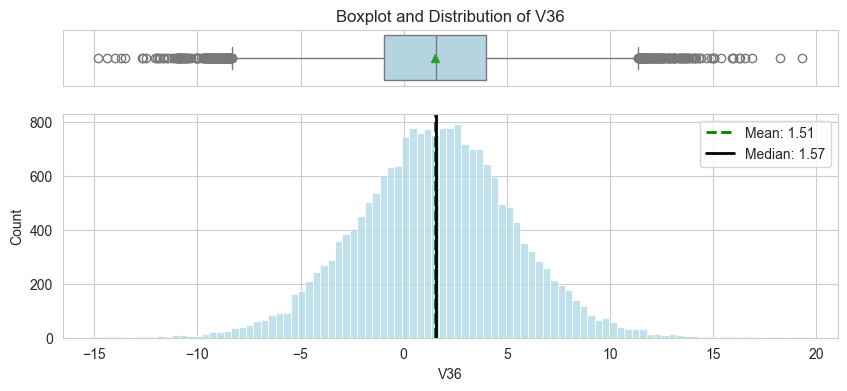


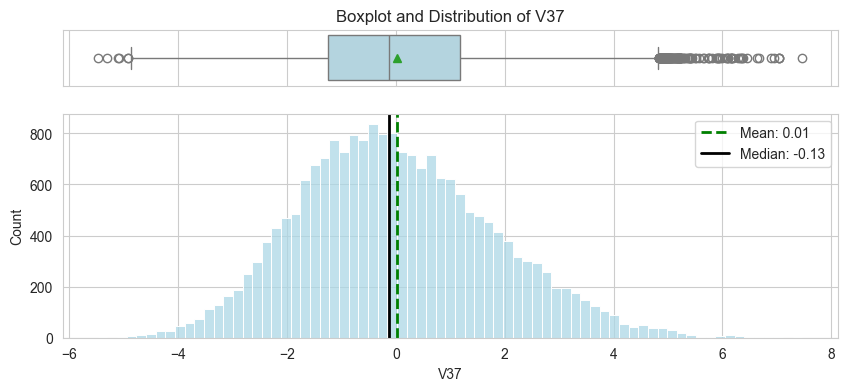


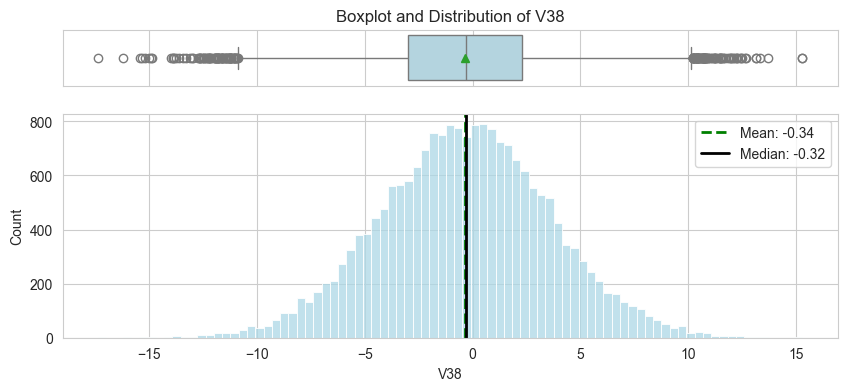


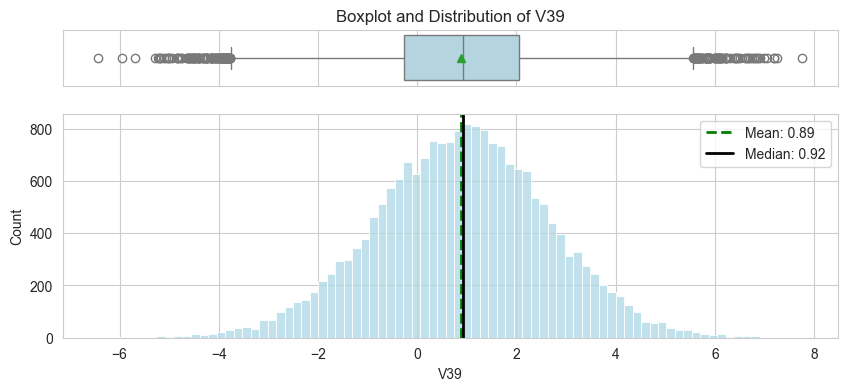


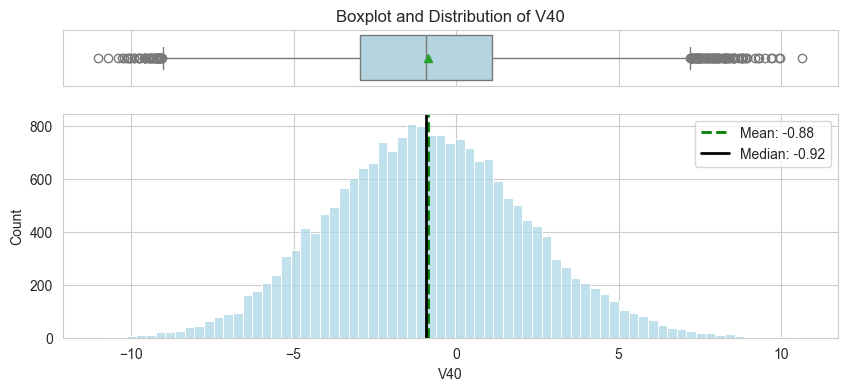


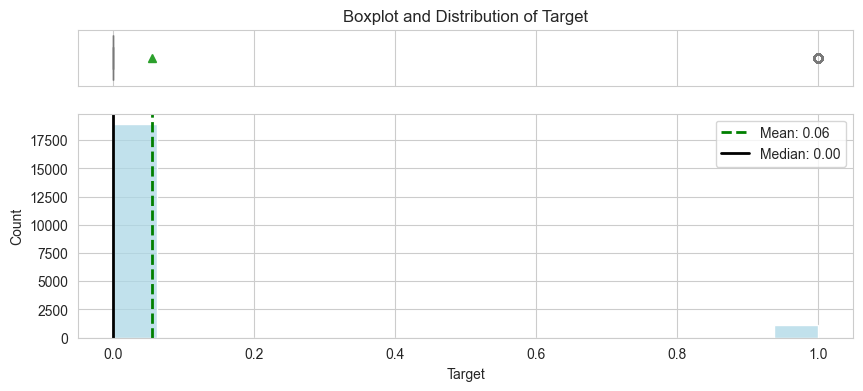












Observations:

**Independent variables**:

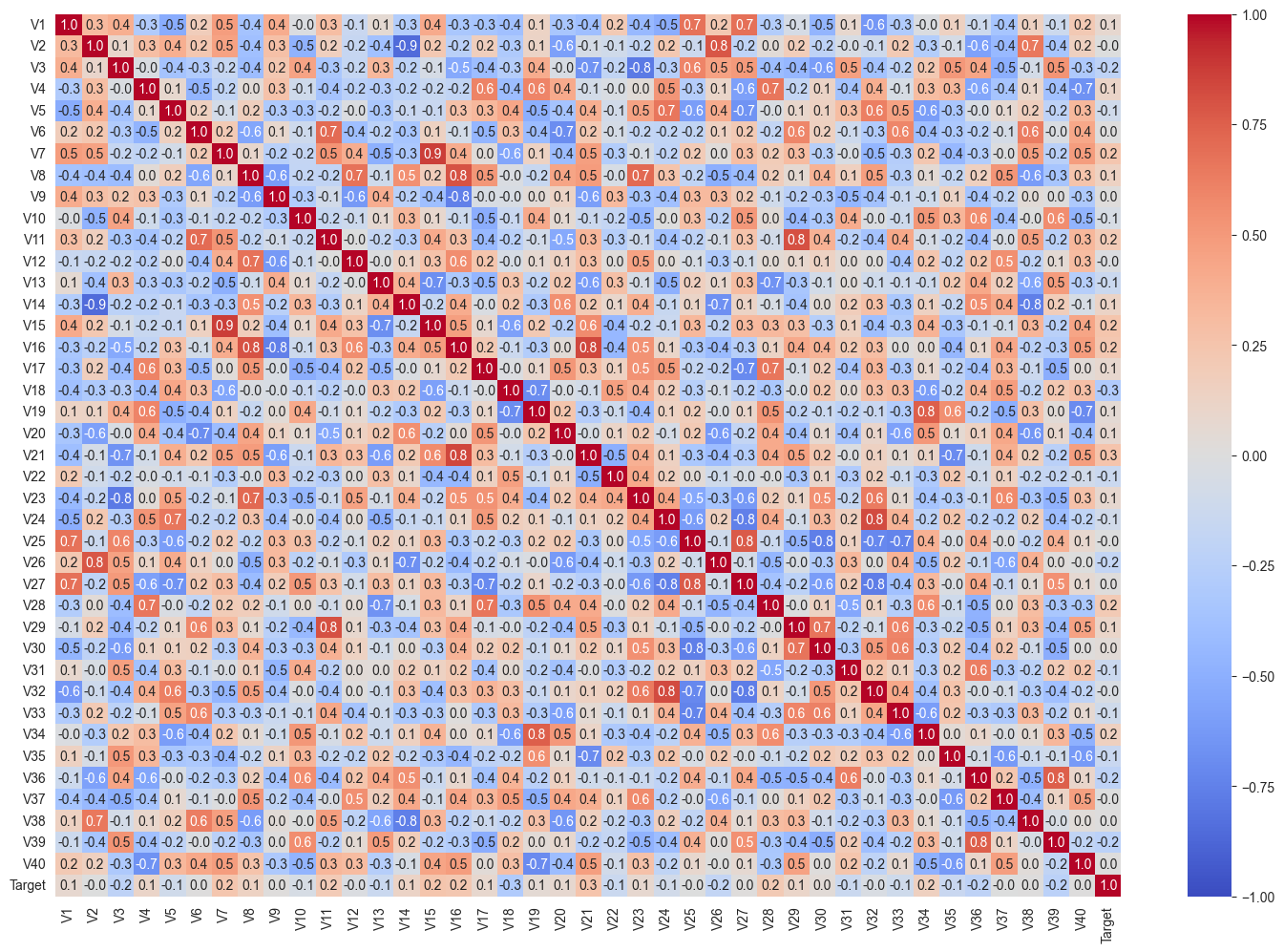
* The histograms indicate that the data for these variables is approximately normally distributed with mean and median fairly close for most, further suggesting a normal distribution.
* The boxplots show dots outside of the whiskers, which represent outliers. These are observations that fall significantly higher or lower than the majority of the data.

**The target variable**:

* Highly imbalanced, with a much larger count of the 0 class compared to the 1 class consistnat with the numerical class distribution observation noted earlier.

## Bivariate Analysis

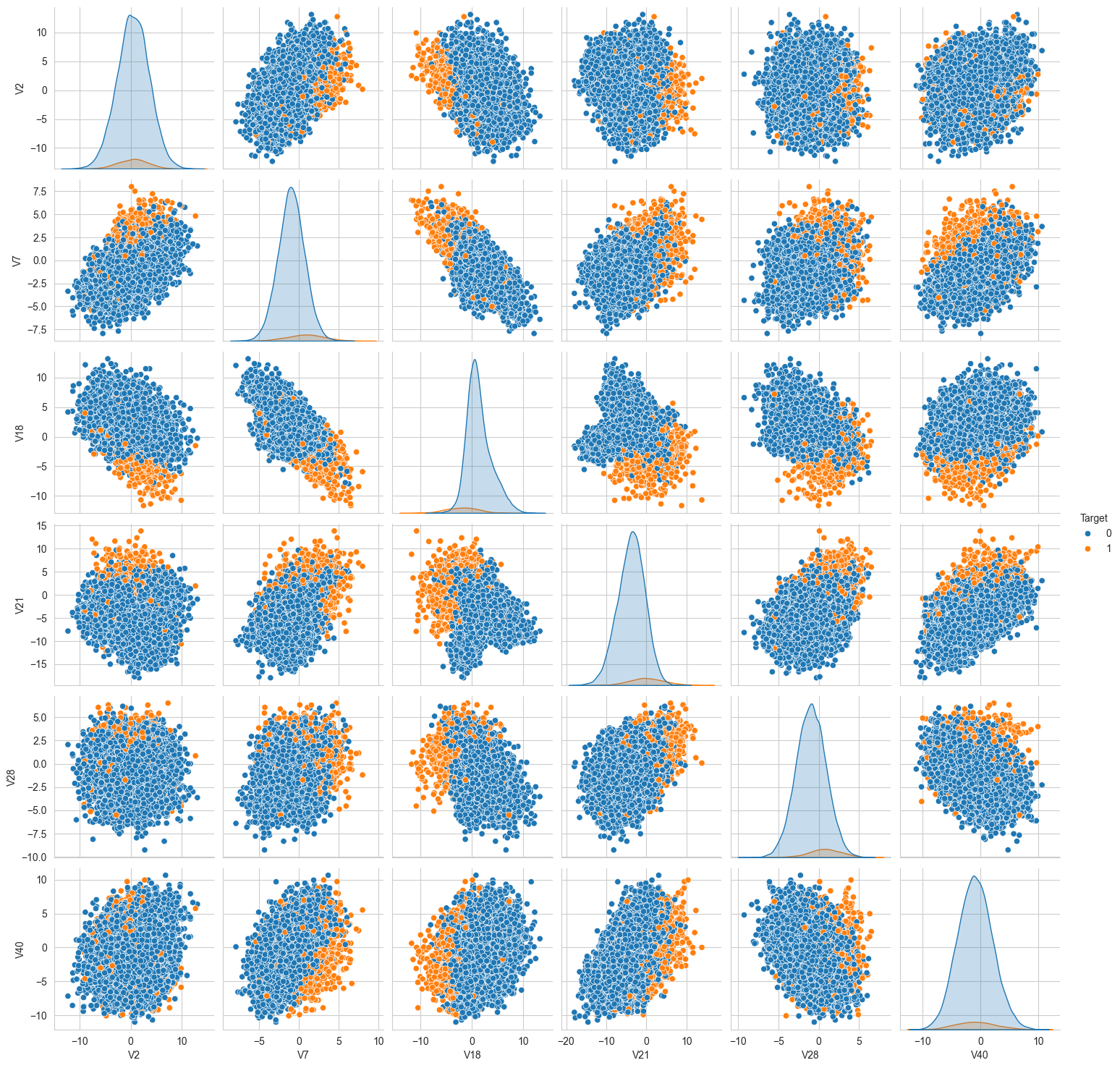
# Heatmap of the correlation coefficients between numerical variables in the dataset  
cols\_list = train\_data.select\_dtypes(include=np.number).columns.tolist()  
  
plt.figure(figsize=(18, 12))  
sns.heatmap(  
 train\_data[cols\_list].corr(), annot=True, vmin=-1, vmax=1, fmt=".1f", cmap="coolwarm"  
)  
plt.show()



Observations:

* Highly Correlated Features:lots of them (both positive and negative). These may represent redundancy within the dataset which means that one feature can be predicted from the other with a high degree of accuracy. In other words, these features carry similar information or signals about the data.
  + Multicollinearity in liner models
  + Overfitting the ti the training dataset due to redundant features
  + Redundant features increase computational complexity
  + Consider dimensionality reduction techniques like PCA
* Relationship with Target: features that have a stronger correlation with the target variable (V18, V21). These features may be particularly important for predicting the target and should be examined more closely in subsequent analyses.

# Selected a subset of features for the pair plot for better performance and clarity  
selected\_features = ['V2', 'V7', 'V18', 'V21', 'V28', 'V40', 'Target']  
  
# Create the pair plot using the selected features  
sns.pairplot(train\_data[selected\_features], hue='Target', diag\_kind='kde')  
plt.show()



train\_df = train\_data.copy()  
test\_df = test\_data.copy()

# Data Pre-Processing

# Split train data into X and y to separate the features from the target  
X = train\_df.drop(["Target"], axis=1)  
y = train\_df["Target"]

# Split the X training set into train and validate  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(  
 X, y, test\_size=0.25, random\_state=42)

# Check the number of rows and columns in the X\_train data  
X\_train.shape

(15000, 40)

# Check the number of rows and columns in the X\_val data  
X\_val.shape

(5000, 40)

# Split test data into X and y to separate the features from the target  
X\_test = test\_df.drop(["Target"], axis=1)  
y\_test = test\_df["Target"]

# Check the number of rows and columns in the X\_test data  
X\_test.shape

(5000, 40)

* X\_train and y\_train are now the subsets for training the model.
* X\_val and y\_val are for validating the model during the hyperparameter tuning and model selection process.
* X\_test and y\_test (from the original test dataset) are used strictly for the final evaluation of the model to assess its performance on unseen data.

### Missing Value Imputation

# Create an instance of the imputer  
imputer = SimpleImputer(strategy="median")

# Fit on the training data and transform it  
X\_train = pd.DataFrame(imputer.fit\_transform(X\_train), columns=X\_train.columns)  
  
# Transform the validation data based on the fit from the training data  
X\_val = pd.DataFrame(imputer.transform(X\_val), columns=X\_val.columns)  
  
# Transform the test data based on the fit from the training data  
X\_test = pd.DataFrame(imputer.transform(X\_test), columns=X\_test.columns)

# Check that it worked  
print(X\_train.isna().sum())  
print("-"\*15)  
print(X\_val.isna().sum())  
print("-"\*15)  
print(X\_test.isna().sum())  
print("-"\*15)

V1 0  
V2 0  
V3 0  
V4 0  
V5 0  
V6 0  
V7 0  
V8 0  
V9 0  
V10 0  
V11 0  
V12 0  
V13 0  
V14 0  
V15 0  
V16 0  
V17 0  
V18 0  
V19 0  
V20 0  
V21 0  
V22 0  
V23 0  
V24 0  
V25 0  
V26 0  
V27 0  
V28 0  
V29 0  
V30 0  
V31 0  
V32 0  
V33 0  
V34 0  
V35 0  
V36 0  
V37 0  
V38 0  
V39 0  
V40 0  
dtype: int64  
---------------  
V1 0  
V2 0  
V3 0  
V4 0  
V5 0  
V6 0  
V7 0  
V8 0  
V9 0  
V10 0  
V11 0  
V12 0  
V13 0  
V14 0  
V15 0  
V16 0  
V17 0  
V18 0  
V19 0  
V20 0  
V21 0  
V22 0  
V23 0  
V24 0  
V25 0  
V26 0  
V27 0  
V28 0  
V29 0  
V30 0  
V31 0  
V32 0  
V33 0  
V34 0  
V35 0  
V36 0  
V37 0  
V38 0  
V39 0  
V40 0  
dtype: int64  
---------------  
V1 0  
V2 0  
V3 0  
V4 0  
V5 0  
V6 0  
V7 0  
V8 0  
V9 0  
V10 0  
V11 0  
V12 0  
V13 0  
V14 0  
V15 0  
V16 0  
V17 0  
V18 0  
V19 0  
V20 0  
V21 0  
V22 0  
V23 0  
V24 0  
V25 0  
V26 0  
V27 0  
V28 0  
V29 0  
V30 0  
V31 0  
V32 0  
V33 0  
V34 0  
V35 0  
V36 0  
V37 0  
V38 0  
V39 0  
V40 0  
dtype: int64  
---------------

### Scale/Normalize Features

# Create a scaler instance  
scaler = StandardScaler()

# Fit on the training data and transform it  
X\_train = pd.DataFrame(scaler.fit\_transform(X\_train), columns=X\_train.columns)  
  
# Transform the validation and test data  
X\_val = pd.DataFrame(scaler.transform(X\_val), columns=X\_val.columns)  
  
X\_test = pd.DataFrame(scaler.transform(X\_test), columns=X\_test.columns)

# Model Building

# defining a function to compute different metrics to check performance of a classification model built using sklearn  
def model\_performance\_classification\_sklearn(model, predictors, target):  
 """  
 Function to compute different metrics to check classification model performance  
  
 model: classifier  
 predictors: independent variables  
 target: dependent variable  
 """  
 TP = confusion\_matrix(target, model.predict(predictors))[1,1]  
 FP = confusion\_matrix(target, model.predict(predictors))[0,1]  
 FN = confusion\_matrix(target, model.predict(predictors))[1,0]  
   
 # predicting using the independent variables  
 pred = model.predict(predictors)  
  
 acc = accuracy\_score(target, pred) # to compute Accuracy  
 recall = recall\_score(target, pred) # to compute Recall  
 precision = precision\_score(target, pred) # to compute Precision  
 f1 = f1\_score(target, pred) # to compute F1-score  
  
 # creating a dataframe of metrics  
 df\_perf = pd.DataFrame(  
 {  
 "Accuracy": acc,  
 "Recall": recall,  
 "Precision": precision,  
 "F1": f1  
   
 },  
 index=[0],  
 )  
  
 return df\_perf

### Defining scorer for cross-validation and hyperparameter tuning

* The goal heare is to reduce FN and maximize "Recall"
* So, use Recall as a scorer

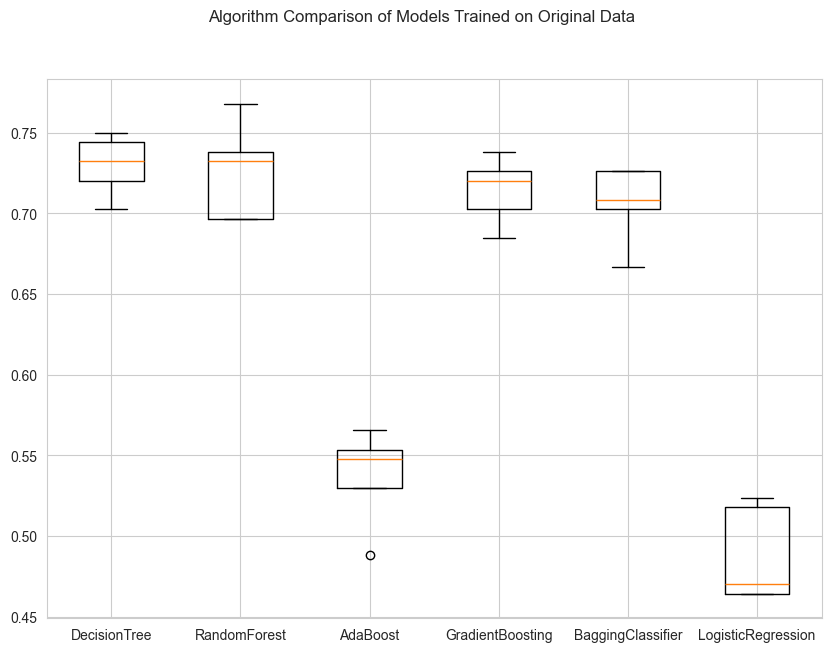
# Defining the scorer based on recall  
scorer = metrics.make\_scorer(metrics.recall\_score)

## Model Building on Original Data

# Models to evaluate  
models = []  
models.append(("DecisionTree", DecisionTreeClassifier(random\_state=1)))  
models.append(("RandomForest", RandomForestClassifier(random\_state=1)))  
models.append(("AdaBoost", AdaBoostClassifier(random\_state=1)))  
models.append(("GradientBoosting", GradientBoostingClassifier(random\_state=1)))  
models.append(("BaggingClassifier", BaggingClassifier(random\_state=1)))  
models.append(("LogisticRegression", LogisticRegression(random\_state=1, max\_iter=10000))) # Increased max\_iter for convergence  
  
results\_original = [] # To store cross-validation results  
names\_original = [] # To store model names  
  
# Cross-validation across all models for Original Data  
print("\nCross-Validation on Original Data:\n")  
  
  
# StratifiedKFold setup  
kfold = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=1)  
  
# Cross-validation across all models  
for name, model in models:  
 cv\_result = cross\_val\_score(model, X\_train, y\_train, scoring=scorer, cv=kfold)  
 results\_original.append(cv\_result)  
 names\_original.append(name)  
 print(f"{name}: Mean Recall Score = {cv\_result.mean()}")  
  
  
print("\nValidation Performance on Original Data:\n")  
  
# Fit models on the original training set and evaluate on the original validation set  
for name, model in models:  
 model.fit(X\_train, y\_train) # Use the training data  
 scores = recall\_score(y\_val, model.predict(X\_val)) # Evaluate against the validation set  
 print(f"{name}: Validation Recall Score = {scores}")

Cross-Validation on Original Data:  
  
DecisionTree: Mean Recall Score = 0.7297619047619047  
RandomForest: Mean Recall Score = 0.7261904761904762  
AdaBoost: Mean Recall Score = 0.536904761904762  
GradientBoosting: Mean Recall Score = 0.7142857142857142  
BaggingClassifier: Mean Recall Score = 0.705952380952381  
LogisticRegression: Mean Recall Score = 0.48809523809523814  
  
Validation Performance on Original Data:  
  
DecisionTree: Validation Recall Score = 0.7111111111111111  
RandomForest: Validation Recall Score = 0.6962962962962963  
AdaBoost: Validation Recall Score = 0.5333333333333333  
GradientBoosting: Validation Recall Score = 0.6888888888888889  
BaggingClassifier: Validation Recall Score = 0.7222222222222222  
LogisticRegression: Validation Recall Score = 0.48148148148148145

# Plotting boxplots for CV scores of all models defined above  
fig = plt.figure(figsize=(10, 7))  
  
fig.suptitle("Algorithm Comparison of Models Trained on Original Data")  
ax = fig.add\_subplot(111)  
  
plt.boxplot(results\_original)  
ax.set\_xticklabels(names\_original)  
  
plt.show()



Recall is highest for Random Forest followed by Decision Tree and Gradient Boosting.

## Model Building with Oversampled Data

# Fit Synthetic Minority Over Sampling Technique (SMOTE) on train data, which uses kNN to generate the data  
sm = SMOTE(sampling\_strategy=1, k\_neighbors=5, random\_state=1)  
X\_train\_over, y\_train\_over = sm.fit\_resample(X\_train, y\_train)

print("Before OverSampling, count of label '1': {}".format(sum(y\_train == 1)))  
print("Before OverSampling, count of label '0': {} \n".format(sum(y\_train == 0)))  
  
print("After OverSampling, count of label '1': {}".format(sum(y\_train\_over == 1)))  
print("After OverSampling, count of label '0': {} \n".format(sum(y\_train\_over == 0)))  
  
print("After OverSampling, the shape of train\_X: {}".format(X\_train\_over.shape))  
print("After OverSampling, the shape of train\_y: {} \n".format(y\_train\_over.shape))

Before OverSampling, count of label '1': 840  
Before OverSampling, count of label '0': 14160   
  
After OverSampling, count of label '1': 14160  
After OverSampling, count of label '0': 14160   
  
After OverSampling, the shape of train\_X: (28320, 40)  
After OverSampling, the shape of train\_y: (28320,)

Observations:

SMOTE has successfully balanced your dataset by generating synthetic examples of the minority class (1) using k-nearest neighbors.

Before applying SMOTE: the training dataset was significantly imbalanced with only 833 instances of the minority class (1) compared to 14,167 instances of the majority class (0). This imbalance leads to models that are biased towards the majority class and ignoring the minority class which is of greater interest to us.

After applying SMOTE: both classes now have an equal count of 14,167 instances, doubling the size of the training dataset to 28,334 instances. This balanced dataset is expected to improve model performance on the minority class by providing it with more examples to learn from, thus helping the model to generalize better and be less biased towards the majority class.

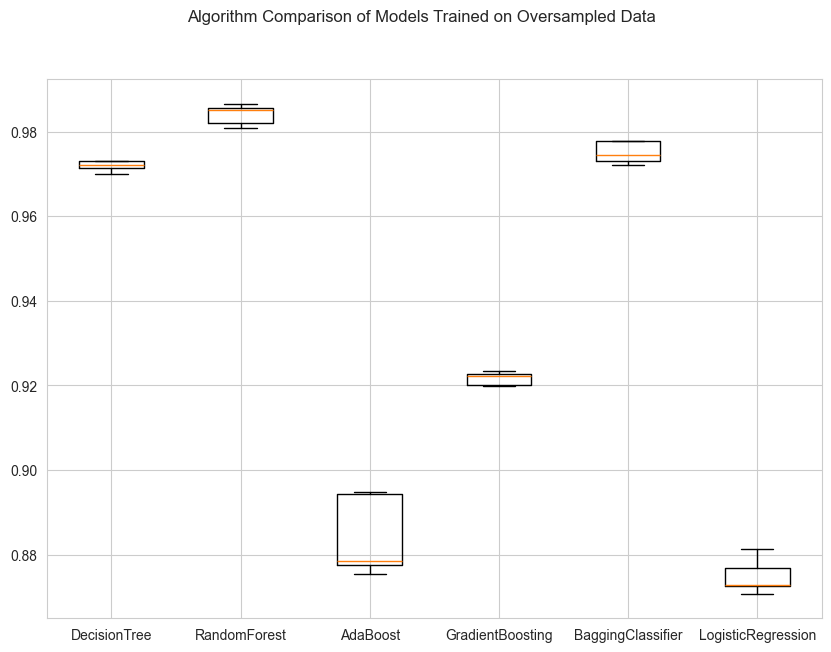
# Models to evaluate on Oversampled Data  
models = []  
models.append(("DecisionTree", DecisionTreeClassifier(random\_state=1)))  
models.append(("RandomForest", RandomForestClassifier(random\_state=1)))  
models.append(("AdaBoost", AdaBoostClassifier(random\_state=1)))  
models.append(("GradientBoosting", GradientBoostingClassifier(random\_state=1)))  
models.append(("BaggingClassifier", BaggingClassifier(random\_state=1)))  
models.append(("LogisticRegression", LogisticRegression(random\_state=1, max\_iter=10000))) # Increased max\_iter for convergence  
  
# To store cross-validation results  
results\_oversampled = []  
# To store model names  
names\_oversampled= []  
  
# Cross-validation across all models for Oversampled Data  
print("\nCross-Validation on Oversampled Data:\n")  
  
# StratifiedKFold setup  
kfold\_oversampled = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=1)  
  
# Cross-validation across all models  
for name, model in models:  
 cv\_result\_oversampled = cross\_val\_score(model, X\_train\_over, y\_train\_over, scoring=scorer, cv=kfold\_oversampled)  
 results\_oversampled.append(cv\_result\_oversampled)  
 names\_oversampled.append(name)  
 print(f"{name}: Mean Recall Score = {cv\_result\_oversampled.mean()}")  
  
print("\nValidation Performance on Oversampled Data:\n")  
  
# Fit models on the oversampled training set and evaluate on the original validation set  
for name, model in models:  
 model.fit(X\_train\_over, y\_train\_over) # Use the oversampled training data  
 scores\_oversampled = recall\_score(y\_val, model.predict(X\_val)) # Evaluate against the validation set  
 print(f"{name}: Validation Recall Score = {scores\_oversampled}")

Cross-Validation on Oversampled Data:  
  
DecisionTree: Mean Recall Score = 0.9719632768361581  
RandomForest: Mean Recall Score = 0.984039548022599  
AdaBoost: Mean Recall Score = 0.8841101694915254  
GradientBoosting: Mean Recall Score = 0.9216807909604519  
BaggingClassifier: Mean Recall Score = 0.9750706214689266  
LogisticRegression: Mean Recall Score = 0.8748587570621469  
  
Validation Performance on Oversampled Data:  
  
DecisionTree: Validation Recall Score = 0.7814814814814814  
RandomForest: Validation Recall Score = 0.8296296296296296  
AdaBoost: Validation Recall Score = 0.8444444444444444  
GradientBoosting: Validation Recall Score = 0.8814814814814815  
BaggingClassifier: Validation Recall Score = 0.8111111111111111  
LogisticRegression: Validation Recall Score = 0.8518518518518519

Observations:

As expected, the balancing of the data with SMOTE improved model performance accross the board. However, it also introduced overfitting issues (ratio of Cross-Validation to Validation scores). Except for AdaBoost and Logistic Regression.

# Plotting boxplots for CV scores of all models evaluated on oversampled data  
fig = plt.figure(figsize=(10, 7))  
  
fig.suptitle("Algorithm Comparison of Models Trained on Oversampled Data")  
ax = fig.add\_subplot(111)  
  
plt.boxplot(results\_oversampled)  
ax.set\_xticklabels(names\_oversampled)  
  
plt.show()



Observations:

Same trend, jsut more exagerated.

## Model Building with Undersampled Data

# Random undersampler for under sampling the data  
rus = RandomUnderSampler(random\_state=1, sampling\_strategy=1)  
X\_train\_un, y\_train\_un = rus.fit\_resample(X\_train, y\_train)

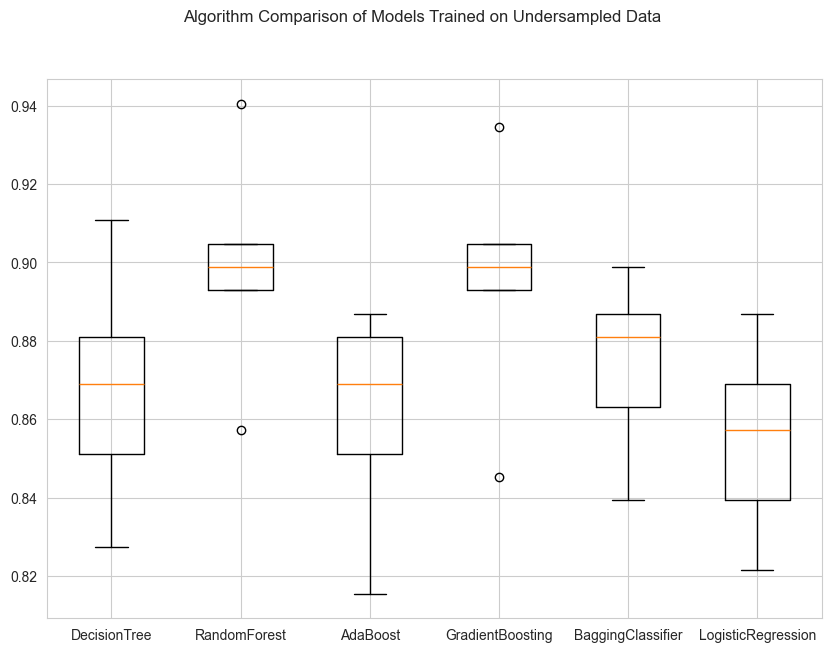
print("Before Under Sampling, count of label '1': {}".format(sum(y\_train== 1)))  
print("Before Under Sampling, count of label '0': {} \n".format(sum(y\_train == 0)))  
  
print("After Under Sampling, count of label '1': {}".format(sum(y\_train\_un == 1)))  
print("After Under Sampling, count of label '0': {} \n".format(sum(y\_train\_un == 0)))  
  
print("After Under Sampling, the shape of train\_X: {}".format(X\_train\_un.shape))  
print("After Under Sampling, the shape of train\_y: {} \n".format(y\_train\_un.shape))

Before Under Sampling, count of label '1': 840  
Before Under Sampling, count of label '0': 14160   
  
After Under Sampling, count of label '1': 840  
After Under Sampling, count of label '0': 840   
  
After Under Sampling, the shape of train\_X: (1680, 40)  
After Under Sampling, the shape of train\_y: (1680,)

# Models to evaluate on Undersampled Data  
models\_undersampled = []  
models\_undersampled.append(("DecisionTree", DecisionTreeClassifier(random\_state=1)))  
models\_undersampled.append(("RandomForest", RandomForestClassifier(random\_state=1)))  
models\_undersampled.append(("AdaBoost", AdaBoostClassifier(random\_state=1)))  
models\_undersampled.append(("GradientBoosting", GradientBoostingClassifier(random\_state=1)))  
models\_undersampled.append(("BaggingClassifier", BaggingClassifier(random\_state=1)))  
models\_undersampled.append(("LogisticRegression", LogisticRegression(random\_state=1, max\_iter=10000))) # Increased max\_iter for convergence  
  
# To store cross-validation results for undersampled data  
results\_undersampled = []  
# To store model names for undersampled data  
names\_undersampled = []  
  
# Cross-validation across all models for Undersampled Data  
print("\nCross-Validation on Undersampled Data:\n")  
  
# StratifiedKFold setup for undersampled data  
kfold\_undersampled = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=1)  
  
# Cross-validation across all models  
for name, model in models\_undersampled:  
 cv\_result\_undersampled = cross\_val\_score(model, X\_train\_un, y\_train\_un, scoring=scorer, cv=kfold\_undersampled)  
 results\_undersampled.append(cv\_result\_undersampled)  
 names\_undersampled.append(name)  
 print(f"{name}: Mean Recall Score = {cv\_result\_undersampled.mean()}")  
  
print("\nValidation Performance on Undersampled Data:\n")  
  
# Fit models on the undersampled training set and evaluate on the original validation set  
for name, model in models\_undersampled:  
 model.fit(X\_train\_un, y\_train\_un) # Use the undersampled training data  
 scores\_undersampled = recall\_score(y\_val, model.predict(X\_val)) # Evaluate against the validation set  
 print(f"{name}: Validation Recall Score = {scores\_undersampled}")

Cross-Validation on Undersampled Data:  
  
DecisionTree: Mean Recall Score = 0.8678571428571427  
RandomForest: Mean Recall Score = 0.8988095238095237  
AdaBoost: Mean Recall Score = 0.8607142857142858  
GradientBoosting: Mean Recall Score = 0.8952380952380953  
BaggingClassifier: Mean Recall Score = 0.8738095238095237  
LogisticRegression: Mean Recall Score = 0.8547619047619047  
  
Validation Performance on Undersampled Data:  
  
DecisionTree: Validation Recall Score = 0.837037037037037  
RandomForest: Validation Recall Score = 0.8777777777777778  
AdaBoost: Validation Recall Score = 0.8555555555555555  
GradientBoosting: Validation Recall Score = 0.8888888888888888  
BaggingClassifier: Validation Recall Score = 0.8481481481481481  
LogisticRegression: Validation Recall Score = 0.8555555555555555

# Plotting boxplots for CV scores of all models evaluated on undersampled data  
fig = plt.figure(figsize=(10, 7))  
  
fig.suptitle("Algorithm Comparison of Models Trained on Undersampled Data")  
ax = fig.add\_subplot(111)  
  
plt.boxplot(results\_undersampled)  
ax.set\_xticklabels(names\_undersampled)  
  
plt.show()



Observations:

Fast. Random Forest still highest performace. Less overfitting accross the board.

## Hyperparameter Tuning

### Model Selection for Hyperparameter Tuning

Top 3 models providing the highest Recall scores on both cross-validation and validation datasets :

1. **RandomForest**
2. **GradientBoosting**
3. **AdaBoost**

### Hyperparameter Tuning of Random Forest on Original Data

# Defining the Random Forest model for original data  
rf\_model\_orig = RandomForestClassifier(random\_state=1)  
  
# Parameter grid for Random Forest  
param\_grid\_rf\_orig = {  
 "n\_estimators": [200, 250], # Reduced for simplicity  
 "min\_samples\_leaf": [1, 2], # Simplified range  
 "max\_features": ['sqrt'], # Simplified choice  
 "max\_samples": [0.5] # Simplified choice  
}  
  
# Setting up RandomizedSearchCV for the original data with reduced n\_iter and cv  
randomized\_rf\_orig = RandomizedSearchCV(estimator=rf\_model\_orig, param\_distributions=param\_grid\_rf\_orig,  
 scoring= scorer, n\_iter=10, cv=3, random\_state=1)  
  
# Fitting RandomizedSearchCV on the original data  
randomized\_rf\_orig.fit(X\_train, y\_train)  
  
# Extracting best parameters and creating a new Random Forest Classifier  
best\_params\_rf\_orig = randomized\_rf\_orig.best\_params\_  
tuned\_rf\_orig = RandomForestClassifier(\*\*best\_params\_rf\_orig, random\_state=1)  
  
# Fitting the tuned model and evaluating  
tuned\_rf\_orig.fit(X\_train, y\_train)

RandomForestClassifier(max\_samples=0.5, n\_estimators=200, random\_state=1)

# Evaluating the tuned model on original data  
rf\_train\_perf\_orig = model\_performance\_classification\_sklearn(tuned\_rf\_orig, X\_train, y\_train)  
print("Performance on Train Set:")  
rf\_train\_perf\_orig

Performance on Train Set:

Accuracy Recall Precision F1  
0 0.993 0.879 0.999 0.935

rf\_val\_perf\_orig = model\_performance\_classification\_sklearn(tuned\_rf\_orig, X\_val, y\_val)  
print("Performance on Validation Set:")  
rf\_val\_perf\_orig

Performance on Validation Set:

Accuracy Recall Precision F1  
0 0.982 0.670 0.989 0.799

### Hyperparameter Tuning of Random Forest on Oversampled Data

# Defining the Random Forest model for original data  
rf\_model\_over = RandomForestClassifier(random\_state=1)  
  
# Parameter grid for Random Forest  
param\_grid\_rf\_orig = {  
 "n\_estimators": [200, 250], # Reduced for simplicity  
 "min\_samples\_leaf": [1, 2], # Simplified range  
 "max\_features": ['sqrt'], # Simplified choice  
 "max\_samples": [0.5] # Simplified choice  
}  
  
# Setting up RandomizedSearchCV for the original data  
randomized\_rf\_over = RandomizedSearchCV(estimator=rf\_model\_orig, param\_distributions=param\_grid\_rf\_orig,  
 scoring=scorer, n\_iter=50, cv=5, random\_state=1)  
  
# Fitting RandomizedSearchCV on the oversampled data  
randomized\_rf\_over.fit(X\_train\_over, y\_train\_over)  
  
# Extracting best parameters for oversampled data  
best\_params\_rf\_over = randomized\_rf\_over.best\_params\_  
  
# Creating a new Random Forest Classifier with the best parameters for oversampled data  
tuned\_rf\_over = RandomForestClassifier(\*\*best\_params\_rf\_over, random\_state=1)  
  
# Fitting the tuned model on the oversampled training data  
tuned\_rf\_over.fit(X\_train\_over, y\_train\_over)

RandomForestClassifier(max\_samples=0.5, n\_estimators=250, random\_state=1)

# Evaluating the tuned model on oversampled train data  
rf\_train\_perf\_over = model\_performance\_classification\_sklearn(tuned\_rf\_over, X\_train\_over, y\_train\_over)  
rf\_train\_perf\_over

Accuracy Recall Precision F1  
0 0.999 0.998 1.000 0.999

# Evaluating the tuned model on oversampled validation data  
rf\_val\_perf\_over = model\_performance\_classification\_sklearn(tuned\_rf\_over, X\_val, y\_val)  
rf\_val\_perf\_over

Accuracy Recall Precision F1  
0 0.988 0.859 0.917 0.887

### Hyperparameter Tuning of Random Forest on Undersampled Data

# Defining the Random Forest model for original data  
rf\_model\_un = RandomForestClassifier(random\_state=1)  
  
# Parameter grid for Random Forest  
param\_grid\_rf\_orig = {  
 "n\_estimators": [200, 250], # Reduced for simplicity  
 "min\_samples\_leaf": [1, 2], # Simplified range  
 "max\_features": ['sqrt'], # Simplified choice  
 "max\_samples": [0.5] # Simplified choice  
}  
  
# Setting up RandomizedSearchCV for the original data  
randomized\_rf\_un = RandomizedSearchCV(estimator=rf\_model\_orig, param\_distributions=param\_grid\_rf\_orig,  
 scoring= scorer, n\_iter=50, cv=5, random\_state=1)  
  
# Fitting RandomizedSearchCV on the undersampled data  
randomized\_rf\_un.fit(X\_train\_un, y\_train\_un)  
  
# Extracting best parameters for undersampled data  
best\_params\_rf\_un = randomized\_rf\_un.best\_params\_  
print("Best parameters are {} with CV score={}:" .format(randomized\_rf\_un.best\_params\_,randomized\_rf\_un.best\_score\_))

Best parameters are {'n\_estimators': 250, 'min\_samples\_leaf': 2, 'max\_samples': 0.5, 'max\_features': 'sqrt'} with CV score=0.8928571428571429:

# Creating a new Random Forest Classifier with the best parameters for undersampled data  
tuned\_rf\_un = RandomForestClassifier(\*\*best\_params\_rf\_un, random\_state=1)  
  
# Fitting the tuned model on the undersampled training data  
tuned\_rf\_un.fit(X\_train\_un, y\_train\_un)

RandomForestClassifier(max\_samples=0.5, min\_samples\_leaf=2, n\_estimators=250,  
 random\_state=1)

# Evaluating the tuned model on undersampled data  
rf\_train\_perf\_un = model\_performance\_classification\_sklearn(tuned\_rf\_un, X\_train\_un, y\_train\_un)  
rf\_train\_perf\_un

Accuracy Recall Precision F1  
0 0.966 0.940 0.991 0.965

rf\_val\_perf\_un = model\_performance\_classification\_sklearn(tuned\_rf\_un, X\_val, y\_val)  
rf\_val\_perf\_un

Accuracy Recall Precision F1  
0 0.935 0.878 0.450 0.595

### Hyperparameter Tuning of Gradient Boost on Original Data

# Defining the Gradient Boosting model  
gb\_model\_orig = GradientBoostingClassifier(random\_state=1)  
  
# Parameter grid remains the same as the oversampling case  
param\_grid\_orig = {  
 "n\_estimators": np.arange(100, 150, 25),  
 "learning\_rate": [0.2, 0.05, 1],  
 "subsample": [0.5, 0.7],  
 "max\_features": [0.5, 0.7]  
}  
  
# Setting up RandomizedSearchCV for the original data  
randomized\_cv\_orig = RandomizedSearchCV(estimator=gb\_model\_orig, param\_distributions=param\_grid\_orig,  
 scoring=scorer, n\_iter=50, n\_jobs=-1, cv=5, random\_state=1)  
  
# Fitting RandomizedSearchCV on the original data  
randomized\_cv\_orig.fit(X\_train, y\_train)  
  
# Best parameters  
# print("Best parameters for original data are {} with CV score={}:".format(randomized\_cv\_orig.best\_params\_, randomized\_cv\_orig.best\_score\_))  
  
# Extracting best parameters for the original data  
best\_params\_orig = randomized\_cv\_orig.best\_params\_  
  
# Creating a new Gradient Boosting Classifier with the best parameters for original data  
tuned\_gbm\_orig = GradientBoostingClassifier(  
 max\_features=best\_params\_orig['max\_features'],  
 random\_state=1,  
 learning\_rate=best\_params\_orig['learning\_rate'],  
 n\_estimators=best\_params\_orig['n\_estimators'],  
 subsample=best\_params\_orig['subsample']  
)  
  
# Fitting the tuned model on the original training data  
tuned\_gbm\_orig.fit(X\_train, y\_train)

GradientBoostingClassifier(learning\_rate=0.2, max\_features=0.7,  
 n\_estimators=np.int64(125), random\_state=1,  
 subsample=0.7)

# Checking performance   
gbm\_train\_perf\_orig = model\_performance\_classification\_sklearn(tuned\_gbm\_orig, X\_train\_over, y\_train\_over)  
print("Performance on Train Set:")  
gbm\_train\_perf\_orig

Performance on Train Set:

Accuracy Recall Precision F1  
0 0.912 0.825 1.000 0.904

gbm\_val\_perf\_orig = model\_performance\_classification\_sklearn(tuned\_gbm\_orig, X\_val, y\_val)  
print("Performance on Validation Set:")  
gbm\_val\_perf\_orig

Performance on Validation Set:

Accuracy Recall Precision F1  
0 0.979 0.715 0.869 0.785

### Hyperparameter Tuning of Gradient Boost on Oversampled Data

# Defining the Gradient Boosting model for oversampled data  
gb\_model\_over = GradientBoostingClassifier(random\_state=1)  
  
# Parameter grid for oversampled data (you can adjust or use the same)  
param\_grid\_over = {  
 "n\_estimators": np.arange(100, 150, 25),  
 "learning\_rate": [0.2, 0.05, 1],  
 "subsample": [0.5, 0.7],  
 "max\_features": [0.5, 0.7]  
}  
  
# Setting up RandomizedSearchCV for the oversampled data  
randomized\_cv\_over = RandomizedSearchCV(estimator=gb\_model\_over, param\_distributions=param\_grid\_over,  
 scoring=scorer, n\_iter=50, n\_jobs=-1, cv=5, random\_state=1)  
  
# Fitting RandomizedSearchCV on the oversampled data  
randomized\_cv\_over.fit(X\_train\_over, y\_train\_over)  
  
# Extracting best parameters for the oversampled data  
best\_params\_over = randomized\_cv\_over.best\_params\_  
  
# Creating a new Gradient Boosting Classifier with the best parameters for oversampled data  
tuned\_gbm\_over = GradientBoostingClassifier(  
 max\_features=best\_params\_over['max\_features'],  
 random\_state=1,  
 learning\_rate=best\_params\_over['learning\_rate'],  
 n\_estimators=best\_params\_over['n\_estimators'],  
 subsample=best\_params\_over['subsample']  
)  
  
# Fitting the tuned model on the oversampled training data  
tuned\_gbm\_over.fit(X\_train\_over, y\_train\_over)

GradientBoostingClassifier(learning\_rate=1, max\_features=0.5,  
 n\_estimators=np.int64(125), random\_state=1,  
 subsample=0.7)

# Checking performance   
gbm\_train\_perf\_over = model\_performance\_classification\_sklearn(tuned\_gbm\_over, X\_train\_over, y\_train\_over)  
gbm\_train\_perf\_over

Accuracy Recall Precision F1  
0 0.993 0.992 0.993 0.993

gbm\_val\_perf\_over = model\_performance\_classification\_sklearn(tuned\_gbm\_over, X\_val, y\_val)  
gbm\_val\_perf\_over

Accuracy Recall Precision F1  
0 0.968 0.867 0.657 0.748

### Hyperparameter Tuning of Gradient Boost on Undersampled Data

# Defining the Gradient Boosting model for undersampled data  
gb\_model\_un = GradientBoostingClassifier(random\_state=1)  
  
# Parameter grid for undersampled data (you can adjust or use the same)  
param\_grid\_un = {  
 "n\_estimators": np.arange(100, 150, 25),  
 "learning\_rate": [0.2, 0.05, 1],  
 "subsample": [0.5, 0.7],  
 "max\_features": [0.5, 0.7]  
}  
  
# Setting up RandomizedSearchCV for the undersampled data  
randomized\_cv\_un = RandomizedSearchCV(estimator=gb\_model\_un, param\_distributions=param\_grid\_un,  
 scoring=scorer, n\_iter=50, n\_jobs=-1, cv=5, random\_state=1)  
  
# Fitting RandomizedSearchCV on the undersampled data  
randomized\_cv\_un.fit(X\_train\_un, y\_train\_un)  
  
# Extracting best parameters for the undersampled data  
best\_params\_un = randomized\_cv\_un.best\_params\_  
  
# Creating a new Gradient Boosting Classifier with the best parameters for undersampled data  
tuned\_gbm\_un = GradientBoostingClassifier(  
 max\_features=best\_params\_un['max\_features'],  
 random\_state=1,  
 learning\_rate=best\_params\_un['learning\_rate'],  
 n\_estimators=best\_params\_un['n\_estimators'],  
 subsample=best\_params\_un['subsample']  
)  
  
# Fitting the tuned model on the undersampled training data  
tuned\_gbm\_un.fit(X\_train\_un, y\_train\_un)

GradientBoostingClassifier(learning\_rate=0.2, max\_features=0.5,  
 n\_estimators=np.int64(100), random\_state=1,  
 subsample=0.5)

# Checking performance   
gbm\_train\_perf\_un = model\_performance\_classification\_sklearn(tuned\_gbm\_un, X\_train\_un, y\_train\_un)  
gbm\_train\_perf\_un

Accuracy Recall Precision F1  
0 0.984 0.975 0.993 0.984

gbm\_val\_perf\_un = model\_performance\_classification\_sklearn(tuned\_gbm\_un, X\_val, y\_val)  
gbm\_val\_perf\_un

Accuracy Recall Precision F1  
0 0.909 0.878 0.361 0.511

### Hyperparameter Tuning of AdaBoost on Original Data

# Defining the AdaBoost model for original data  
ada\_model\_orig = AdaBoostClassifier(random\_state=1)  
  
# Parameter grid for AdaBoost  
param\_grid\_ada\_orig = {  
 "n\_estimators": [50, 100, 150],  
 "learning\_rate": [0.01, 0.05],  
}  
  
  
# Setting up RandomizedSearchCV for the original data  
randomized\_ada\_orig = RandomizedSearchCV(estimator=ada\_model\_orig, param\_distributions=param\_grid\_ada\_orig,  
 scoring= scorer, n\_iter=10, cv=5, random\_state=1)  
  
# Fitting RandomizedSearchCV on the original data  
randomized\_ada\_orig.fit(X\_train, y\_train)  
  
# Extracting best parameters  
best\_params\_ada\_orig = randomized\_ada\_orig.best\_params\_  
  
# Creating a new pipline of AdaBoost Classifier with the best parameters  
tuned\_ada\_orig = AdaBoostClassifier(\*\*best\_params\_ada\_orig, random\_state=1)  
  
# Fitting the tuned model on the original training data  
tuned\_ada\_orig.fit(X\_train, y\_train)

AdaBoostClassifier(learning\_rate=0.05, n\_estimators=150, random\_state=1)

# Evaluating the tuned model  
ada\_train\_perf\_orig = model\_performance\_classification\_sklearn(tuned\_ada\_orig, X\_train, y\_train)  
ada\_train\_perf\_orig

Accuracy Recall Precision F1  
0 0.951 0.140 0.915 0.244

ada\_val\_perf\_orig = model\_performance\_classification\_sklearn(tuned\_ada\_orig, X\_val, y\_val)  
ada\_val\_perf\_orig

Accuracy Recall Precision F1  
0 0.952 0.137 0.881 0.237

### Hyperparameter Tuning of AdaBoost on Oversample Data

# Defining the AdaBoost model for oversampled data  
ada\_model\_over = AdaBoostClassifier(random\_state=1)  
  
# Parameter grid for AdaBoost  
param\_grid\_ada\_over = {  
 "n\_estimators": [50, 100, 150],  
 "learning\_rate": [0.01, 0.05],  
}  
# Setting up RandomizedSearchCV for the oversampled data  
randomized\_ada\_over = RandomizedSearchCV(estimator=ada\_model\_orig, param\_distributions=param\_grid\_ada\_orig,  
 scoring=scorer, n\_iter=10, cv=5, random\_state=1)  
  
# Fitting RandomizedSearchCV on the oversampled data  
randomized\_ada\_over.fit(X\_train\_over, y\_train\_over)  
  
# Extracting best parameters for the oversampled data  
best\_params\_ada\_over = randomized\_ada\_over.best\_params\_  
  
# Creating a new AdaBoost Classifier with the best parameters for oversampled data  
tuned\_ada\_over = AdaBoostClassifier(\*\*best\_params\_ada\_over, random\_state=1)  
  
# Fitting the tuned model on the oversampled training data  
tuned\_ada\_over.fit(X\_train\_over, y\_train\_over)

AdaBoostClassifier(learning\_rate=0.01, random\_state=1)

print("Best parameters are {} with CV score={}:" .format(randomized\_ada\_over.best\_params\_,randomized\_ada\_over.best\_score\_))

Best parameters are {'n\_estimators': 50, 'learning\_rate': 0.01} with CV score=0.8516949152542374:

# Evaluating the tuned model on oversampled data  
ada\_train\_perf\_over = model\_performance\_classification\_sklearn(tuned\_ada\_over, X\_train\_over, y\_train\_over)  
ada\_train\_perf\_over

Accuracy Recall Precision F1  
0 0.759 0.864 0.714 0.782

ada\_val\_perf\_over = model\_performance\_classification\_sklearn(tuned\_ada\_over, X\_val, y\_val)  
ada\_val\_perf\_over

Accuracy Recall Precision F1  
0 0.664 0.804 0.118 0.205

### Hyperparameter Tuning of AdaBoost on Undersampled Data

# Defining the AdaBoost model for oversampled data  
ada\_model\_un = AdaBoostClassifier(random\_state=1)  
  
# Parameter grid for AdaBoost  
param\_grid\_ada\_un = {  
 "n\_estimators": [50, 100, 150],  
 "learning\_rate": [0.01, 0.05],  
}  
# Setting up RandomizedSearchCV for the undersampled data  
randomized\_ada\_un = RandomizedSearchCV(estimator=ada\_model\_orig, param\_distributions=param\_grid\_ada\_orig,  
 scoring=scorer, n\_iter=10, cv=5, random\_state=1)  
  
# Fitting RandomizedSearchCV on the undersampled data  
randomized\_ada\_un.fit(X\_train\_un, y\_train\_un)  
  
# Extracting best parameters for the undersampled data  
best\_params\_ada\_un = randomized\_ada\_un.best\_params\_  
  
# Creating a new AdaBoost Classifier with the best parameters for undersampled data  
tuned\_ada\_un = AdaBoostClassifier(\*\*best\_params\_ada\_un, random\_state=1)  
  
# Fitting the tuned model on the undersampled training data  
tuned\_ada\_un.fit(X\_train\_un, y\_train\_un)

AdaBoostClassifier(learning\_rate=0.05, n\_estimators=150, random\_state=1)

# Evaluating the tuned model on undersampled data  
ada\_train\_perf\_un = model\_performance\_classification\_sklearn(tuned\_ada\_un, X\_train\_un, y\_train\_un)  
ada\_train\_perf\_un

Accuracy Recall Precision F1  
0 0.854 0.781 0.915 0.843

ada\_val\_perf\_un = model\_performance\_classification\_sklearn(tuned\_ada\_un, X\_val, y\_val)  
ada\_val\_perf\_un

Accuracy Recall Precision F1  
0 0.898 0.733 0.312 0.438

# Model Performance Comparison

# Training performance comparison  
  
models\_train\_comp\_df = pd.concat(  
 [  
 rf\_train\_perf\_orig.T,  
 gbm\_train\_perf\_orig.T,  
 ada\_train\_perf\_orig.T,  
   
 rf\_train\_perf\_over.T,  
 gbm\_train\_perf\_over.T,  
 ada\_train\_perf\_over.T,  
  
 rf\_train\_perf\_un.T,   
 gbm\_train\_perf\_un.T,  
 ada\_train\_perf\_un.T,  
   
 ],  
 axis=1,  
)  
models\_train\_comp\_df.columns = [  
 'rf\_train\_perf\_orig',  
 'gbm\_train\_perf\_orig',  
 'ada\_train\_perf\_orig',  
   
 'rf\_train\_perf\_over',  
 'gbm\_train\_perf\_over',  
 'ada\_train\_perf\_over',  
   
 'rf\_train\_perf\_un',  
 'gbm\_train\_perf\_un',  
 'ada\_train\_perf\_un',  
]  
print("Training performance comparison:")  
models\_train\_comp\_df

Training performance comparison:

rf\_train\_perf\_orig gbm\_train\_perf\_orig ada\_train\_perf\_orig \  
Accuracy 0.993 0.912 0.951   
Recall 0.879 0.825 0.140   
Precision 0.999 1.000 0.915   
F1 0.935 0.904 0.244   
  
 rf\_train\_perf\_over gbm\_train\_perf\_over ada\_train\_perf\_over \  
Accuracy 0.999 0.993 0.759   
Recall 0.998 0.992 0.864   
Precision 1.000 0.993 0.714   
F1 0.999 0.993 0.782   
  
 rf\_train\_perf\_un gbm\_train\_perf\_un ada\_train\_perf\_un   
Accuracy 0.966 0.984 0.854   
Recall 0.940 0.975 0.781   
Precision 0.991 0.993 0.915   
F1 0.965 0.984 0.843

# Validation performance comparison  
  
models\_val\_comp\_df = pd.concat(  
 [  
 rf\_val\_perf\_orig.T,  
 gbm\_val\_perf\_orig.T,  
 ada\_val\_perf\_orig.T,  
   
 rf\_val\_perf\_over.T,  
 gbm\_val\_perf\_over.T,  
 ada\_val\_perf\_over.T,  
  
 rf\_val\_perf\_un.T,   
 gbm\_val\_perf\_un.T,  
 ada\_val\_perf\_un.T,  
 ],  
 axis=1,  
)  
models\_val\_comp\_df.columns = [  
 'rf\_val\_perf\_orig',  
 'gbm\_val\_perf\_orig',  
 'ada\_val\_perf\_orig',  
   
 'rf\_val\_perf\_over',  
 'gbm\_val\_perf\_over',  
 'ada\_val\_perf\_over',  
   
 'rf\_val\_perf\_un',  
 'gbm\_val\_perf\_un',  
 'ada\_val\_perf\_un',  
]  
print("Validation performance comparison:")  
models\_val\_comp\_df

Validation performance comparison:

rf\_val\_perf\_orig gbm\_val\_perf\_orig ada\_val\_perf\_orig \  
Accuracy 0.982 0.979 0.952   
Recall 0.670 0.715 0.137   
Precision 0.989 0.869 0.881   
F1 0.799 0.785 0.237   
  
 rf\_val\_perf\_over gbm\_val\_perf\_over ada\_val\_perf\_over \  
Accuracy 0.988 0.968 0.664   
Recall 0.859 0.867 0.804   
Precision 0.917 0.657 0.118   
F1 0.887 0.748 0.205   
  
 rf\_val\_perf\_un gbm\_val\_perf\_un ada\_val\_perf\_un   
Accuracy 0.935 0.909 0.898   
Recall 0.878 0.878 0.733   
Precision 0.450 0.361 0.312   
F1 0.595 0.511 0.438

print("Training performance comparison:")  
models\_train\_comp\_df

Training performance comparison:

rf\_train\_perf\_orig gbm\_train\_perf\_orig ada\_train\_perf\_orig \  
Accuracy 0.993 0.912 0.951   
Recall 0.879 0.825 0.140   
Precision 0.999 1.000 0.915   
F1 0.935 0.904 0.244   
  
 rf\_train\_perf\_over gbm\_train\_perf\_over ada\_train\_perf\_over \  
Accuracy 0.999 0.993 0.759   
Recall 0.998 0.992 0.864   
Precision 1.000 0.993 0.714   
F1 0.999 0.993 0.782   
  
 rf\_train\_perf\_un gbm\_train\_perf\_un ada\_train\_perf\_un   
Accuracy 0.966 0.984 0.854   
Recall 0.940 0.975 0.781   
Precision 0.991 0.993 0.915   
F1 0.965 0.984 0.843

print("Validation performance comparison:")  
models\_val\_comp\_df

Validation performance comparison:

rf\_val\_perf\_orig gbm\_val\_perf\_orig ada\_val\_perf\_orig \  
Accuracy 0.982 0.979 0.952   
Recall 0.670 0.715 0.137   
Precision 0.989 0.869 0.881   
F1 0.799 0.785 0.237   
  
 rf\_val\_perf\_over gbm\_val\_perf\_over ada\_val\_perf\_over \  
Accuracy 0.988 0.968 0.664   
Recall 0.859 0.867 0.804   
Precision 0.917 0.657 0.118   
F1 0.887 0.748 0.205   
  
 rf\_val\_perf\_un gbm\_val\_perf\_un ada\_val\_perf\_un   
Accuracy 0.935 0.909 0.898   
Recall 0.878 0.878 0.733   
Precision 0.450 0.361 0.312   
F1 0.595 0.511 0.438

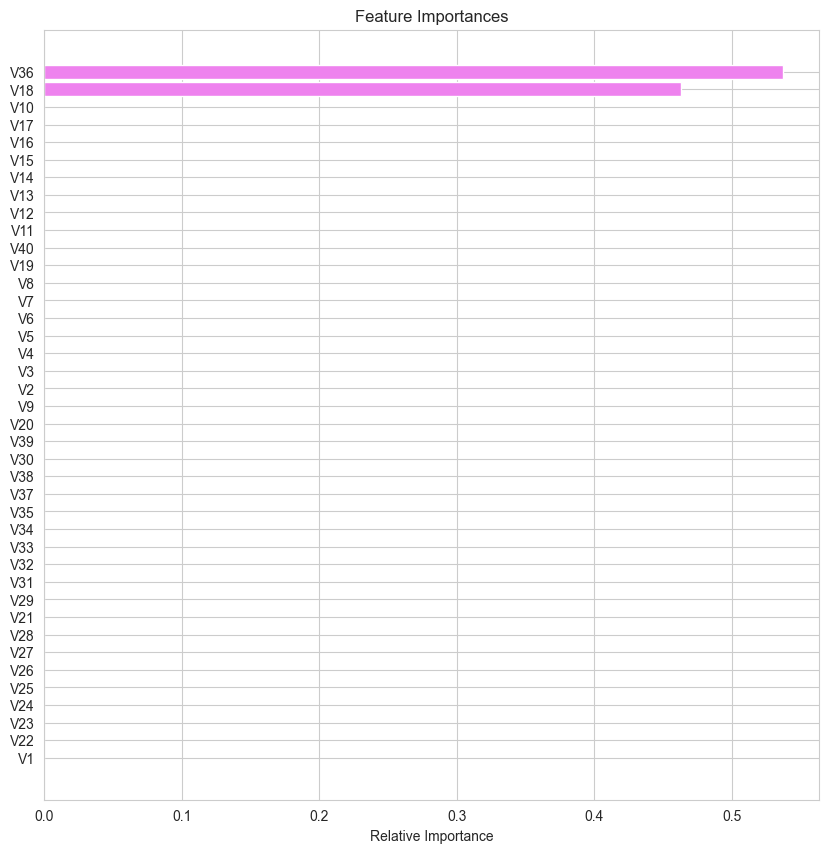
AdaBoost on oversampled data emerges as a strong candidate for the final model due to its excellent performance across accuracy, **recall**, precision, and F1 score in the validation set.

# Final Model Selection

# Check the performance of the final model on the test data  
ada\_test = model\_performance\_classification\_sklearn(tuned\_ada\_over, X\_test, y\_test)  
ada\_test

Accuracy Recall Precision F1  
0 0.675 0.840 0.131 0.226

feature\_names = X\_train.columns  
importances = tuned\_ada\_over.feature\_importances\_  
indices = np.argsort(importances)  
  
plt.figure(figsize=(10, 10))  
plt.title("Feature Importances")  
plt.barh(range(len(indices)), importances[indices], color="violet", align="center")  
plt.yticks(range(len(indices)), [feature\_names[i] for i in indices])  
plt.xlabel("Relative Importance")  
plt.show()



## Pipelines to build the final model

Since we have only one datatype in the data we don't need to use column transofrmer which can be used to personalize the pipeline to perform different preprocessing steps on different columns. Note: see case study how to use this.

The steps followed in implementing a pipeline are -

* Pipeline() - A pipeline is defined as a list of tuples
* Pipeline.fit() - Pipeline is fitted on the train set
* Pipeline.score() - Pipeline objects checks the performance.

pipe = Pipeline(  
 steps=[  
 ("imputer", SimpleImputer(strategy="median")),  
 ("scaler", StandardScaler()),  
 ("AdaBoost", AdaBoostClassifier  
 (  
 n\_estimators= 50,   
 learning\_rate= 0.05,  
 ),  
 ),  
 ]  
)

# Check pipline steps   
pipe.steps

[('imputer', SimpleImputer(strategy='median')),  
 ('scaler', StandardScaler()),  
 ('AdaBoost', AdaBoostClassifier(learning\_rate=0.05))]

# Fit the pipeline on training data as if it were a model since the model is included in it  
pipe.fit(X\_train, y\_train)  
# See pipeline structure below

Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),  
 ('scaler', StandardScaler()),  
 ('AdaBoost', AdaBoostClassifier(learning\_rate=0.05))])

The above means a model is built which becomes part of pipe

# pipe object's accuracy on the train set  
pipe.score(X\_train, y\_train)

0.944

Instead of model.score calling pipe.score fucntion does a predict then uses those results with the actual results to give the score.

# pipe object's accuracy on the test set  
pipe.score(X\_test, y\_test)

0.9436

The data called by pipe.score() undergoes transformation process in the first two steps (imputer and scaler) and the predict unction in the 3rd step ("AdaBoost").

Questions I still have:

* Am I suppose to fit the pipeline into the X\_train, y\_train or X and y?
* What about X\_train\_over, y\_train\_over?
* Is SMOTE incorporated into the pipeline?

# Business Insights and Conclusions

Features V18, V39, and V12 have significant influence on the model's predictions.

The company can focus on further analyzing these important features to better understand their underlying relationships with the target variable.