Problem Statement

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

Importing necessary libraries

```
# Installing the libraries with the specified version.
#!pip install pandas==1.5.3 numpy==1.25.2 matplotlib==3.7.1 seaborn==0.13.1 scikit-learn==1.2.2 imbalanced-learn==0.10.1 xgboost==2.0.3 threadpoolctl==3.3.0 -q --user
```

Note: After running the above cell, kindly restart the notebook kernel and run all cells sequentially from the start again.

```
# Libraries to help with reading and manipulating data
import pandas as pd
import numpy as np

# Libaries to help with data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# To tune model, get different metric scores, and split data
from sklearn.metrics import (
    f1_score,
    accuracy_score,
    recall_score,
    precision_score,
    confusion_matrix,
    roc_auc_score,
    ConfusionMatrixDisplay,
)
from sklearn import metrics

from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score

# To be used for data scaling and one hot encoding
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder

# To impute missing values
from sklearn.impute import SimpleImputer

# To oversample and undersample data
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler

# To do hyperparameter tuning
```

```
from sklearn.model_selection import RandomizedSearchCV

# To be used for creating pipelines and personalizing them
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

# To define maximum number of columns to be displayed in a dataframe
pd.set_option("display.max_columns", None)
pd.set_option("display.max_rows", None)

# To supress scientific notations for a dataframe
pd.set_option("display.float_format", lambda x: "%.3f" % x)

# To help with model building
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import (
    AdaBoostClassifier,
    GradientBoostingClassifier,
    RandomForestClassifier,
    BaggingClassifier,
)
from xgboost import XGBClassifier

# To suppress scientific notations
pd.set_option("display.float_format", lambda x: "%.3f" % x)

# To suppress warnings
import warnings

warnings.filterwarnings("ignore")
```

Loading the dataset

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# original train data
df_train = pd.read_csv('/content/drive/My Drive/Colab Notebooks/6 - Model Tuning/Final Project/Train.csv')
# original test data
df_test = pd.read_csv('/content/drive/My Drive/Colab Notebooks/6 - Model Tuning/Final Project/Test.csv')
```

Data Overview

- Observations
- Sanity checks

Data Overview - TRAIN dataset

```
df_train.head()
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V2
0	-4.465	-4.679	3.102	0.506	-0.221	-2.033	-2.911	0.051	-1.522	3.762	-5.715	0.736	0.981	1.418	-3.376	-3.047	0.306	2.914	2.270	4.395	-2.388	0.646	-1.191	3.133	0.66
1	3.366	3.653	0.910	-1.368	0.332	2.359	0.733	-4.332	0.566	-0.101	1.914	-0.951	-1.255	-2.707	0.193	-4.769	-2.205	0.908	0.757	-5.834	-3.065	1.597	-1.757	1.766	-0.26
2	-3.832	-5.824	0.634	-2.419	-1.774	1.017	-2.099	-3.173	-2.082	5.393	-0.771	1.107	1.144	0.943	-3.164	-4.248	-4.039	3.689	3.311	1.059	-2.143	1.650	-1.661	1.680	-0.45
3	1.618	1.888	7.046	-1.147	0.083	-1.530	0.207	-2.494	0.345	2.119	-3.053	0.460	2.705	-0.636	-0.454	-3.174	-3.404	-1.282	1.582	-1.952	-3.517	-1.206	-5.628	-1.818	2.12
4	-0.111	3.872	-3.758	-2.983	3.793	0.545	0.205	4.849	-1.855	-6.220	1.998	4.724	0.709	-1.989	-2.633	4.184	2.245	3.734	-6.313	-5.380	-0.887	2.062	9.446	4.490	-3.94

```
df_train.shape
```

```
(20000, 41)
```

Observations - There are 20,000 rows and 41 columns in the train dataset

```
df_train.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
```

Observations - There are 41 numerical (1 int64 & 40 float64) the train dataset

```
df_train.describe()
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16
count	19982.000	19982.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000	20000.000
mean	-0.272	0.440	2.485	-0.083	-0.054	-0.995	-0.879	-0.548	-0.017	-0.013	-1.895	1.605	1.580	-0.951	-2.415	-2.925
std	3.442	3.151	3.389	3.432	2.105	2.041	1.762	3.296	2.161	2.193	3.124	2.930	2.875	1.790	3.355	4.222
min	-11.876	-12.320	-10.708	-15.082	-8.603	-10.227	-7.950	-15.658	-8.596	-9.854	-14.832	-12.948	-13.228	-7.739	-16.417	-20.374
25%	-2.737	-1.641	0.207	-2.348	-1.536	-2.347	-2.031	-2.643	-1.495	-1.411	-3.922	-0.397	-0.224	-2.171	-4.415	-5.634
50%	-0.748	0.472	2.256	-0.135	-0.102	-1.001	-0.917	-0.389	-0.068	0.101	-1.921	1.508	1.637	-0.957	-2.383	-2.683
75%	1.840	2.544	4.566	2.131	1.340	0.380	0.224	1.723	1.409	1.477	0.119	3.571	3.460	0.271	-0.359	-0.095
max	15.493	13.089	17.091	13.236	8.134	6.976	8.006	11.679	8.138	8.108	11.826	15.081	15.420	5.671	12.246	13.583

```
df_train.isnull().sum()
```

	0
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

Observations - There is no missing values in the data

```
print("There are",df_train.duplicated().sum(),"duplicated rows")
```

```
There are 0 duplicated rows
```

Data Overview - TEST dataset

```
df_test.head()
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V2
0	-0.613	-3.820	2.202	1.300	-1.185	-4.496	-1.836	4.723	1.206	-0.342	-5.123	1.017	4.819	3.269	-2.984	1.387	2.032	-0.512	-1.023	7.339	-2.242	0.155	2.054	-2.772	1.85
1	0.390	-0.512	0.527	-2.577	-1.017	2.235	-0.441	-4.406	-0.333	1.967	1.797	0.410	0.638	-1.390	-1.883	-5.018	-3.827	2.418	1.762	-3.242	-3.193	1.857	-1.708	0.633	-0.58
2	-0.875	-0.641	4.084	-1.590	0.526	-1.958	-0.695	1.347	-1.732	0.466	-4.928	3.565	-0.449	-0.656	-0.167	-1.630	2.292	2.396	0.601	1.794	-2.120	0.482	-0.841	1.790	1.87
3	0.238	1.459	4.015	2.534	1.197	-3.117	-0.924	0.269	1.322	0.702	-5.578	-0.851	2.591	0.767	-2.391	-2.342	0.572	-0.934	0.509	1.211	-3.260	0.105	-0.659	1.498	1.10
4	5.828	2.768	-1.235	2.809	-1.642	-1.407	0.569	0.965	1.918	-2.775	-0.530	1.375	-0.651	-1.679	-0.379	-4.443	3.894	-0.608	2.945	0.367	-5.789	4.598	4.450	3.225	0.39

```
df_test.shape
```

```
(5000, 41)
```

Observations - There are 5,000 rows and 41 columns in the test dataset

```
df_test.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
```

Observations - There are 41 numerical (1 int64 & 40 float64) the test dataset

```
df_test.describe()
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18
count	4995.000	4994.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000	5000.000
mean	-0.278	0.398	2.552	-0.049	-0.080	-1.042	-0.908	-0.575	0.030	0.019	-2.009	1.576	1.622	-0.921	-2.452	-3.019	-0.104	1.196
std	3.466	3.140	3.327	3.414	2.111	2.005	1.769	3.332	2.174	2.145	3.112	2.907	2.883	1.803	3.387	4.264	3.337	2.586
min	-12.382	-10.716	-9.238	-14.682	-7.712	-8.924	-8.124	-12.253	-6.785	-8.171	-13.152	-8.164	-11.548	-7.814	-15.286	-20.986	-13.418	-12.214
25%	-2.744	-1.649	0.315	-2.293	-1.615	-2.369	-2.054	-2.642	-1.456	-1.353	-4.050	-0.450	-0.126	-2.111	-4.479	-5.648	-2.228	-0.409
50%	-0.765	0.427	2.260	-0.146	-0.132	-1.049	-0.940	-0.358	-0.080	0.166	-2.043	1.488	1.719	-0.896	-2.417	-2.774	0.047	0.881
75%	1.831	2.444	4.587	2.166	1.341	0.308	0.212	1.713	1.450	1.511	0.044	3.563	3.465	0.272	-0.433	-0.178	2.112	2.604
max	13.504	14.079	15.315	12.140	7.673	5.068	7.616	10.415	8.851	6.599	9.956	12.984	12.620	5.734	11.673	13.976	19.777	13.642

```
df_test.isnull().sum()
```

	0
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

Observations - There is no missing values in the data

```
print("There are",df_test.duplicated().sum(),"duplicated rows")
```

```
There are 0 duplicated rows
```

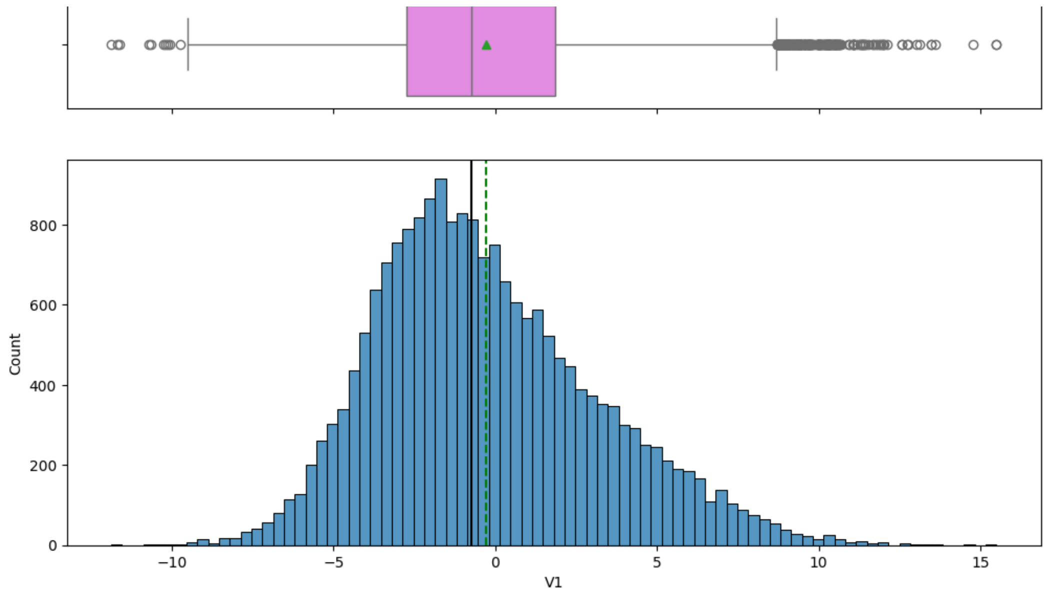
Exploratory Data Analysis (EDA)

Plotting histograms and boxplots for all the variables

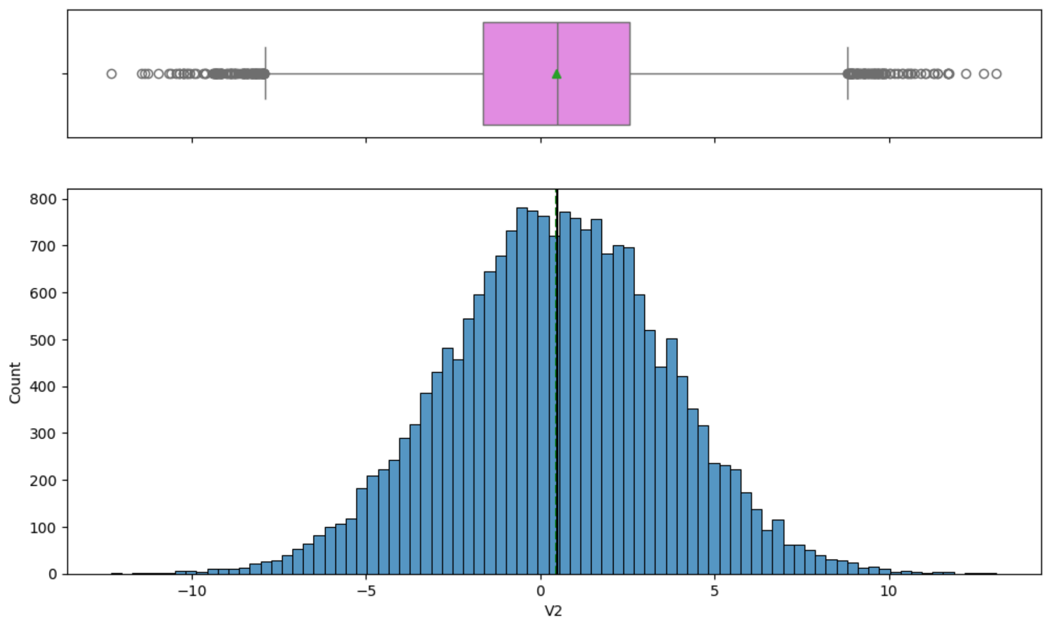
[ ] ↳ 1 cell hidden

Plotting all the features at one go

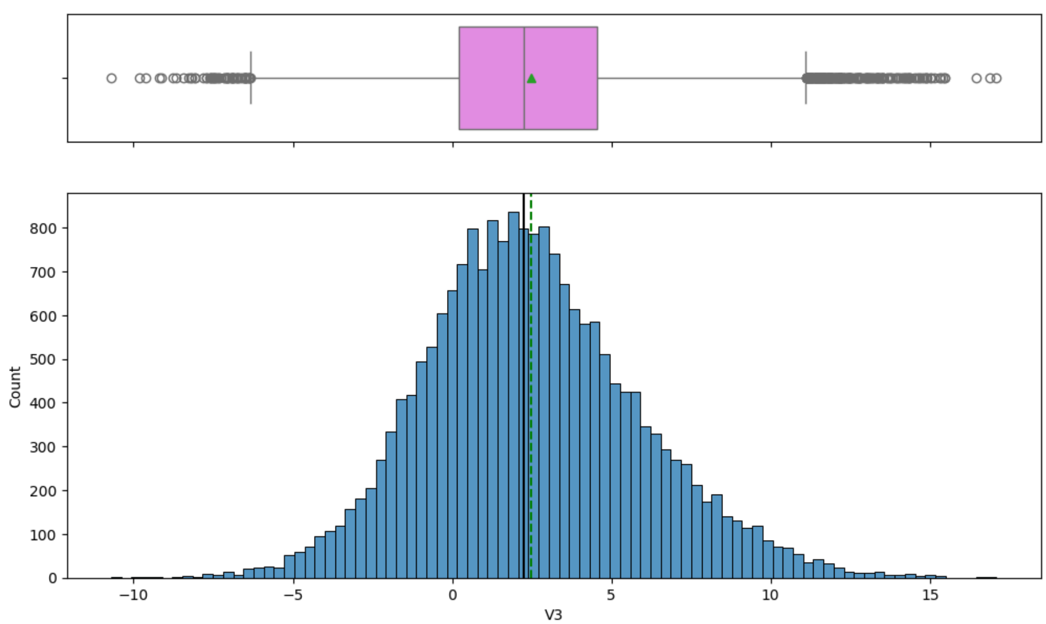
```
for feature in df_train.columns:
    histogram_boxplot(df_train, feature, figsize=(12, 7), kde=False, bins=None)
```



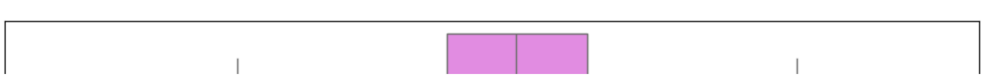
800
600
400
200
0
Count
−10
−5
0
5
10
15
V1

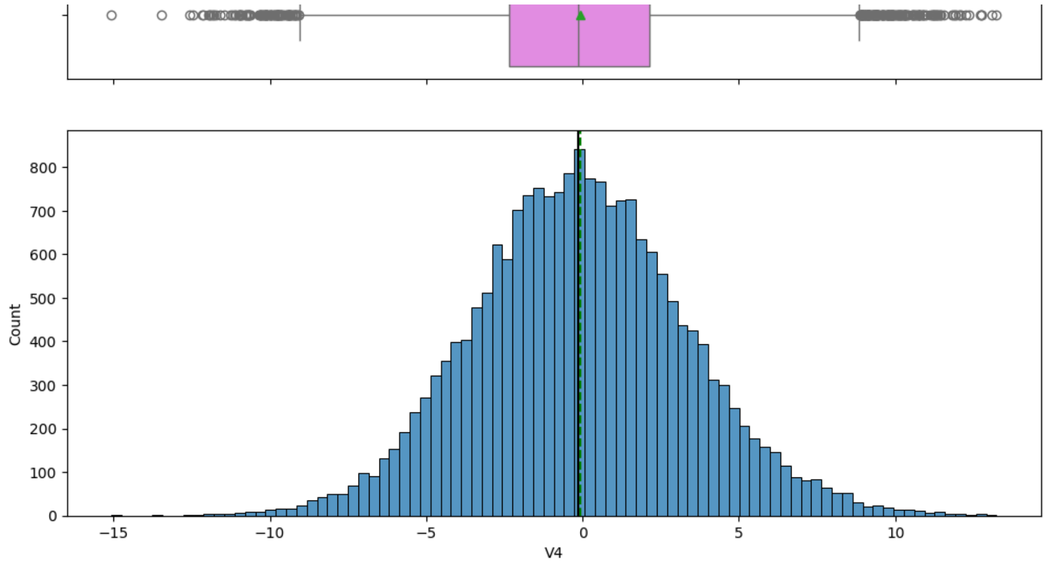


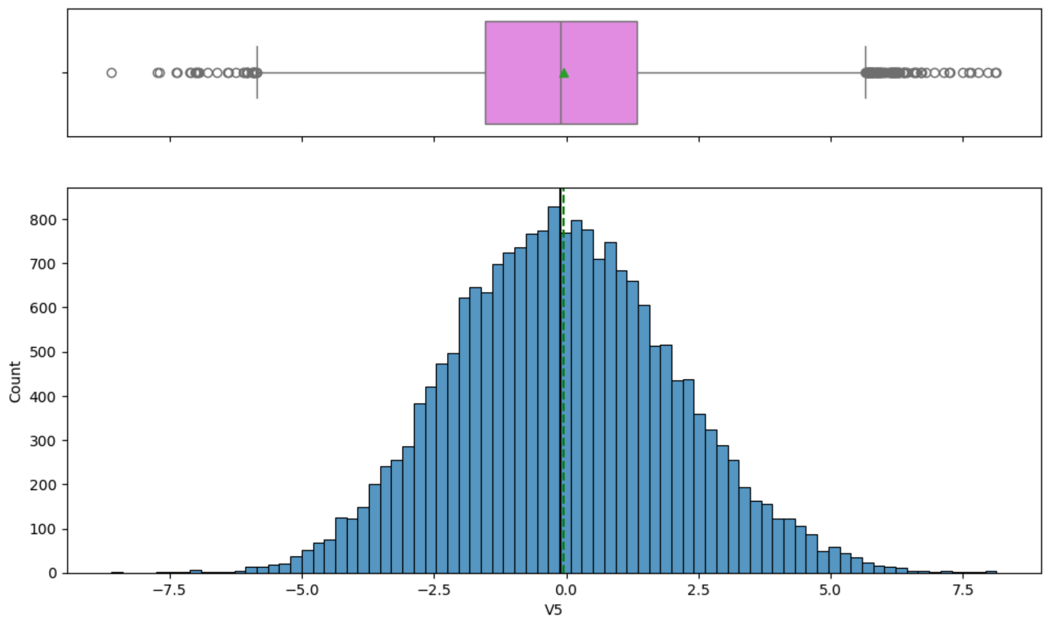
800
700
600
500
400
300
200
100
0
Count
−10
−5
0
5
10
V2

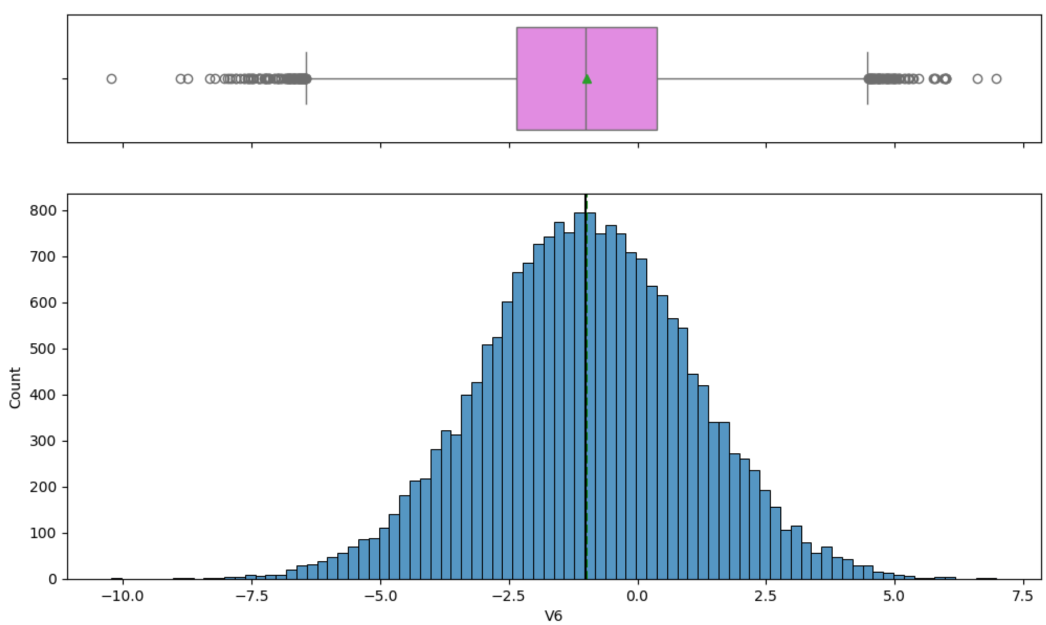


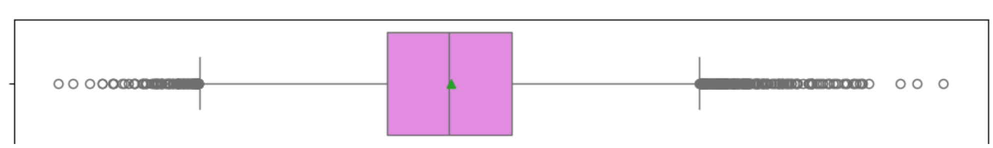
800
700
600
500
400
300
200
100
0
Count
−10
−5
0
5
10
15
V3

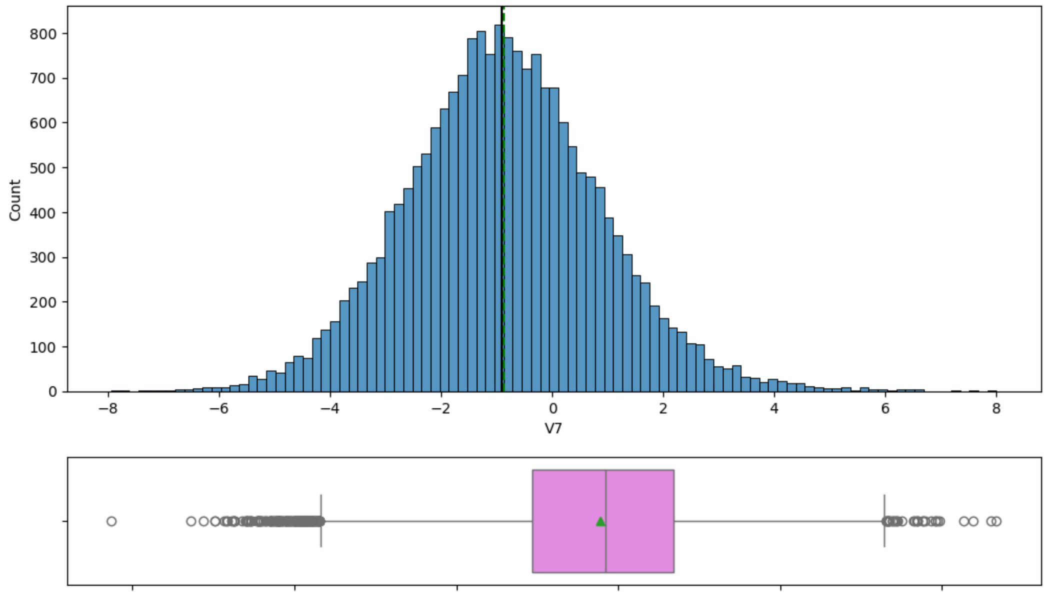


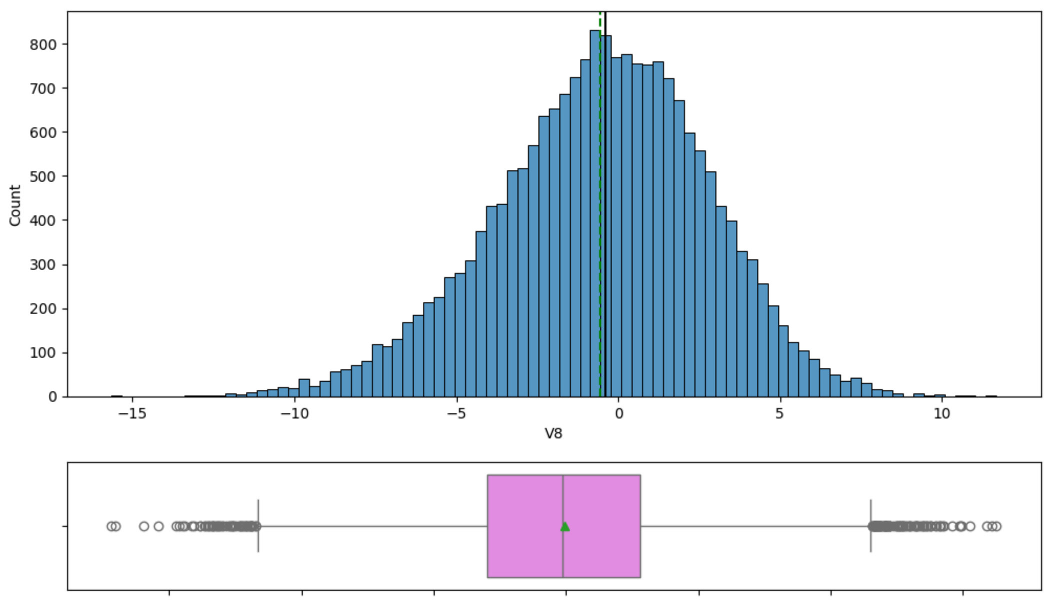

Count
V4

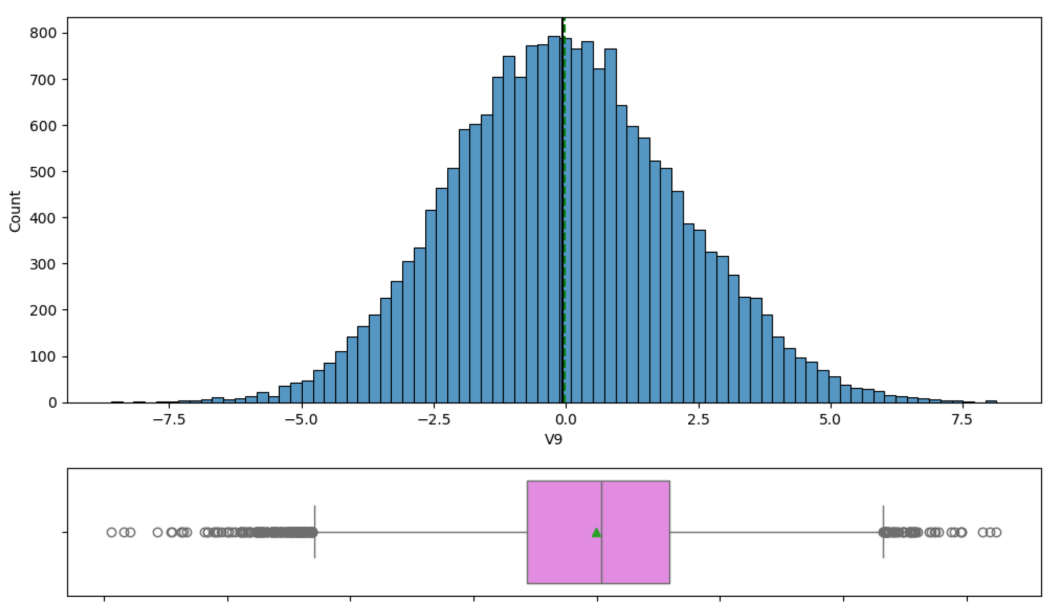

Count
V5

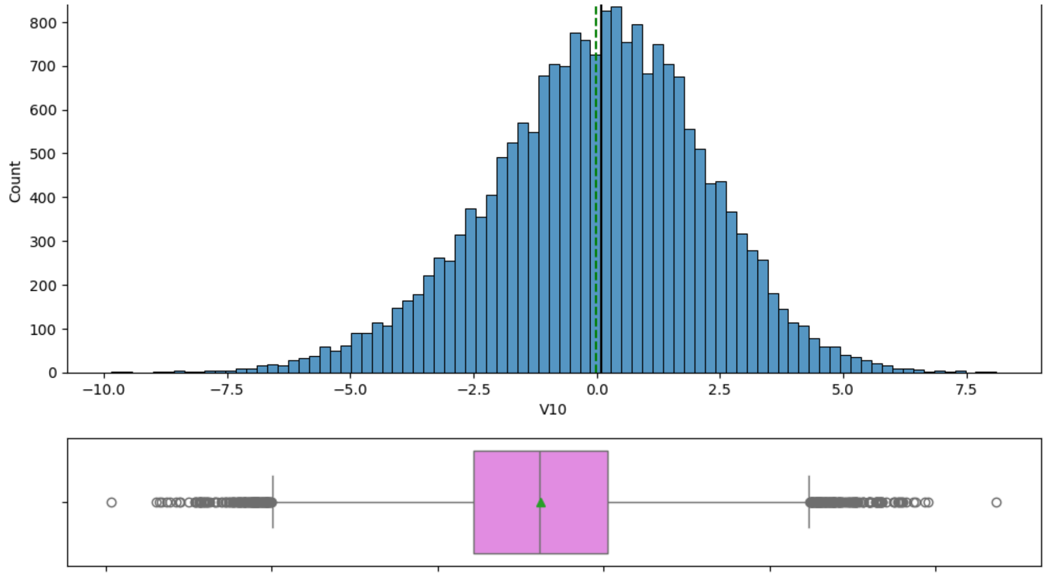

Count
V6

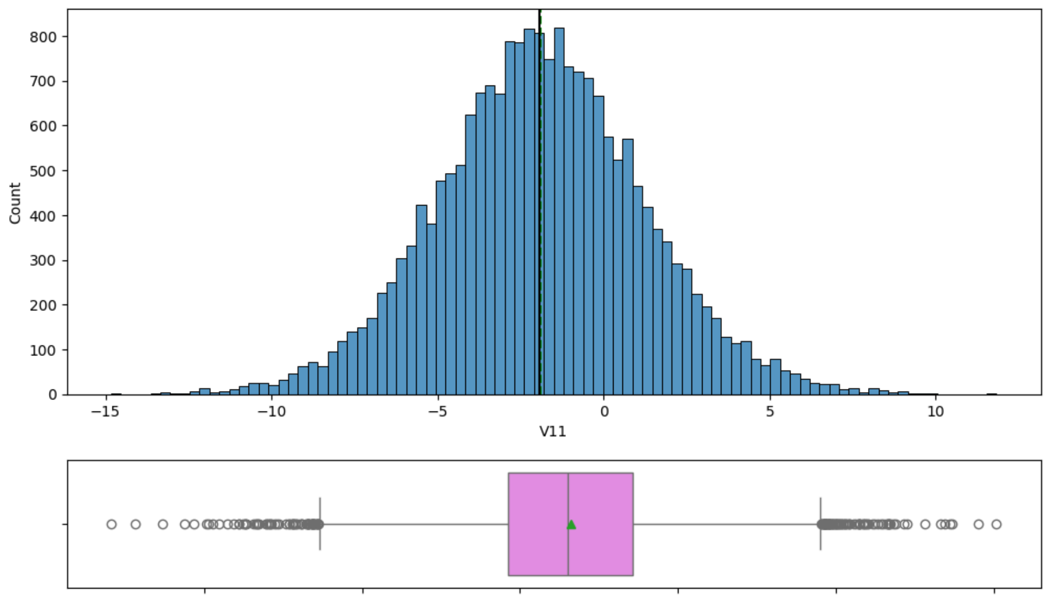


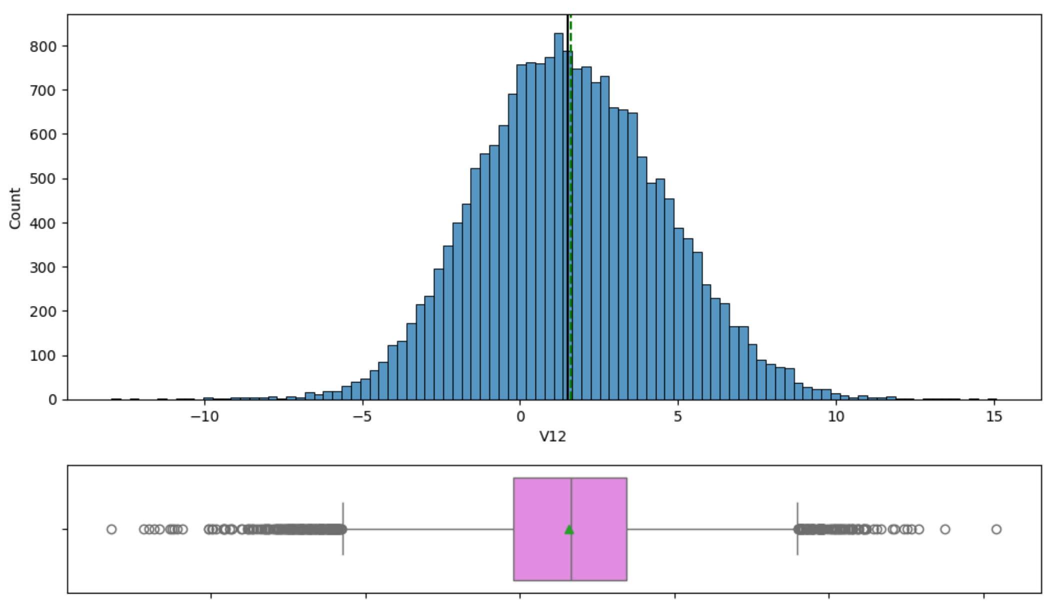

Count
V7
-8 -6 -4 -2 0 2 4 6 8
0 100 200 300 400 500 600 700 800

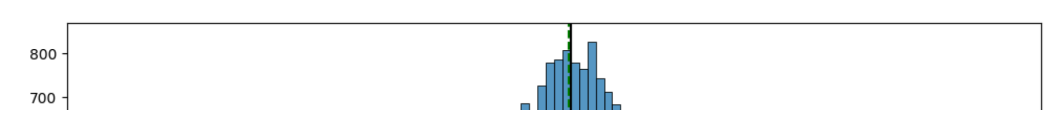

Count
V8
-15 -10 -5 0 5 10
0 100 200 300 400 500 600 700 800

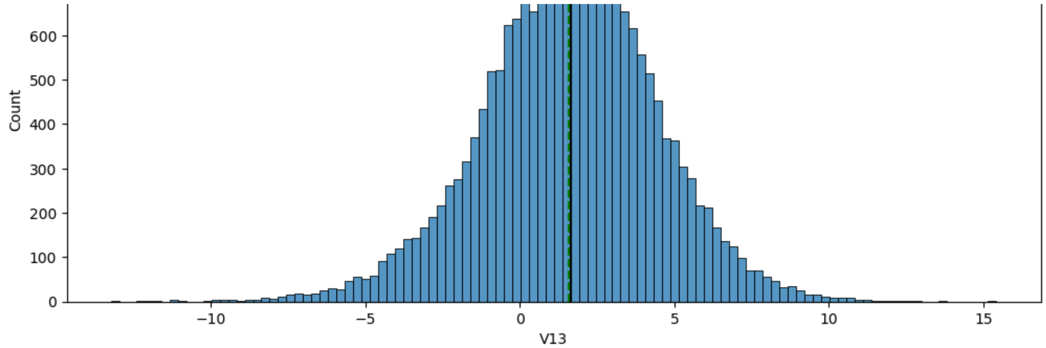

Count
V9
-7.5 -5.0 -2.5 0.0 2.5 5.0 7.5
0 100 200 300 400 500 600 700 800

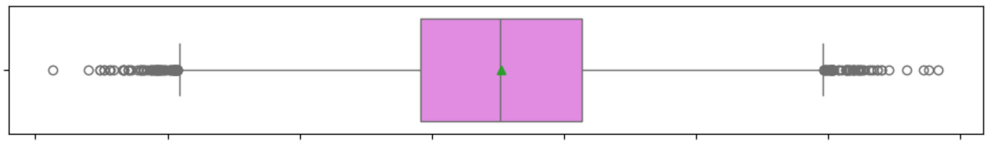

Count
V10

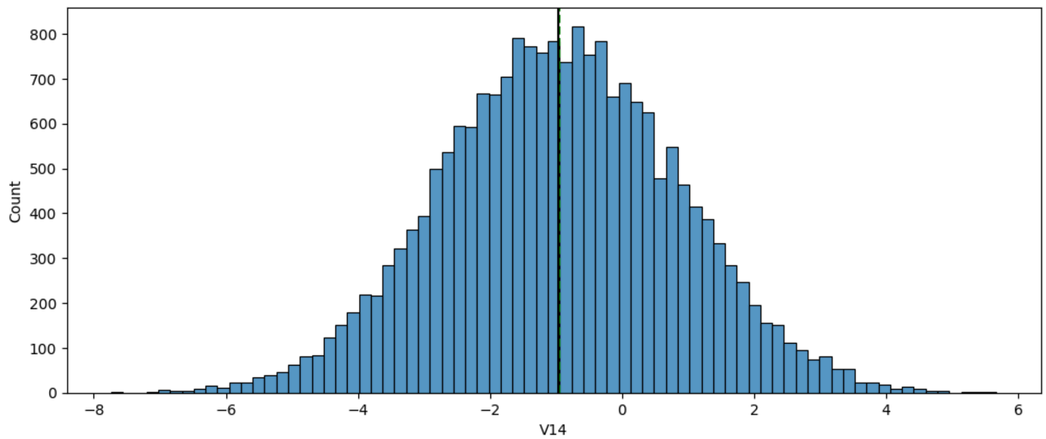

Count
V11

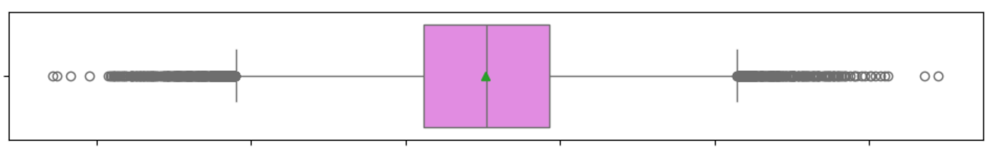

Count
V12

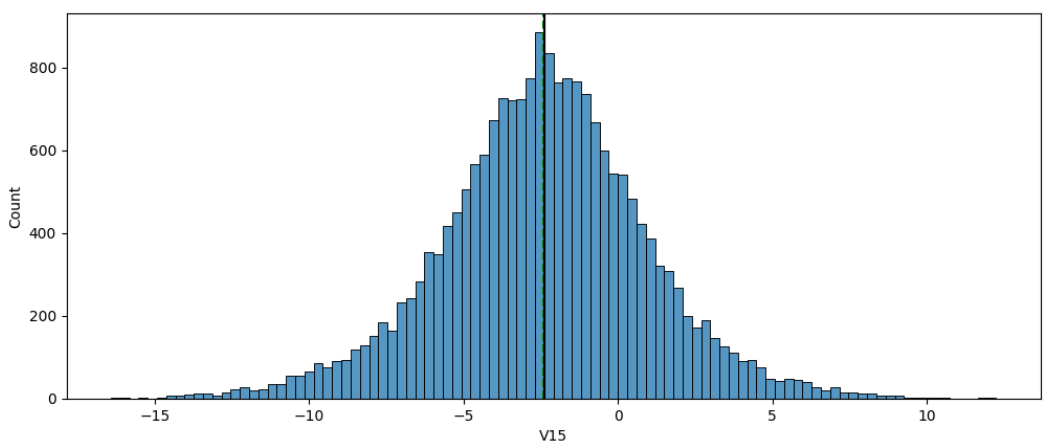


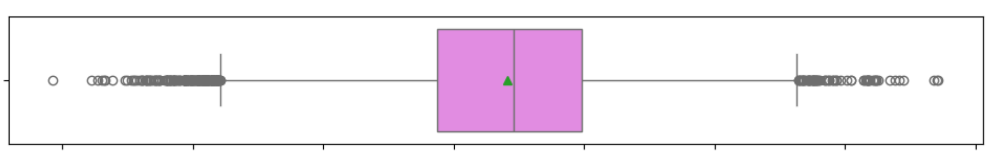

Count
V13
−10
−5
0
5
10
15

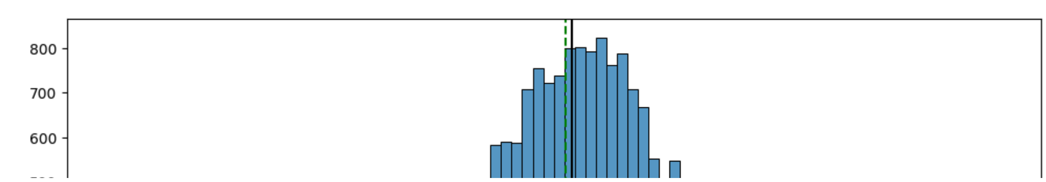


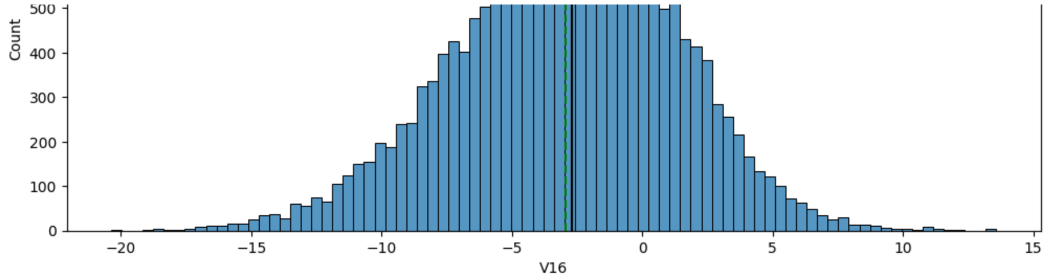

Count
V14
−8
−6
−4
−2
0
2
4
6

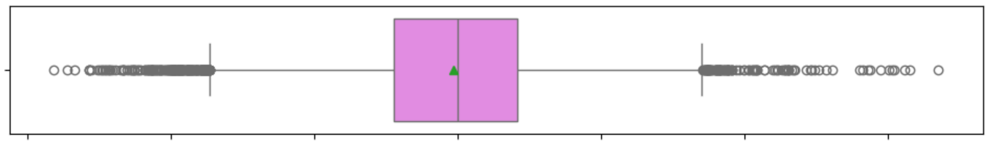


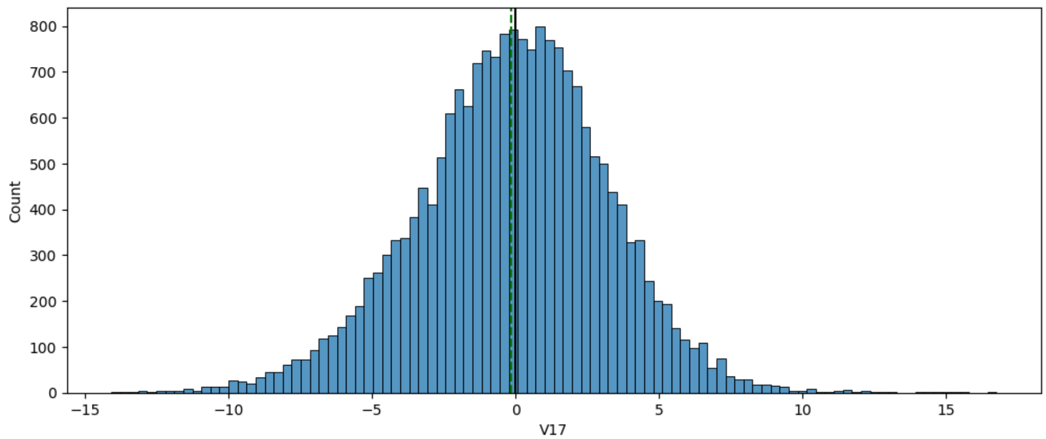

Count
V15
−15
−10
−5
0
5
10

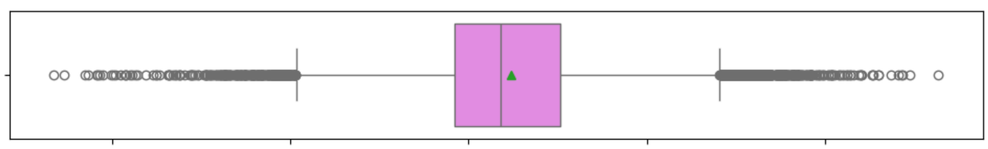


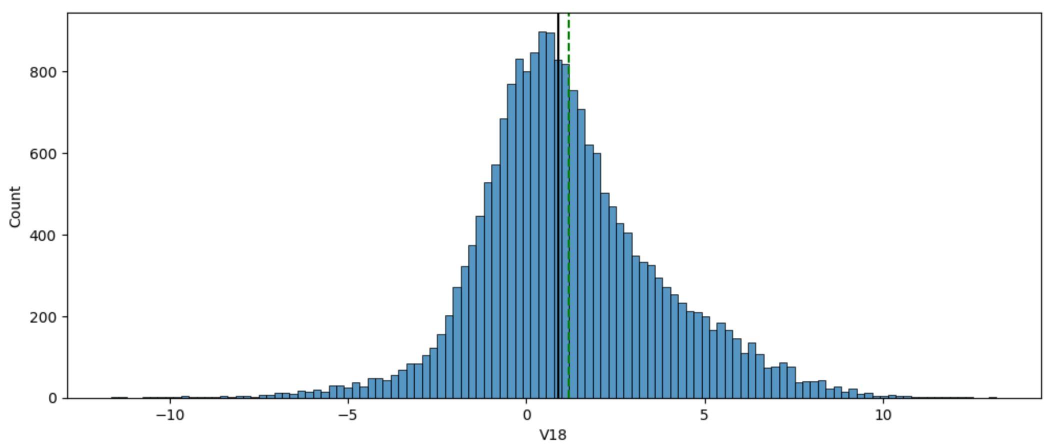


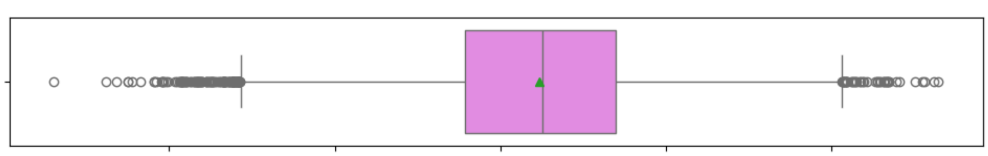

Count
V16

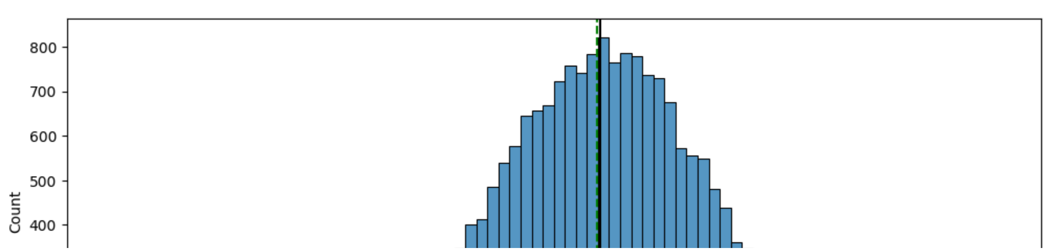


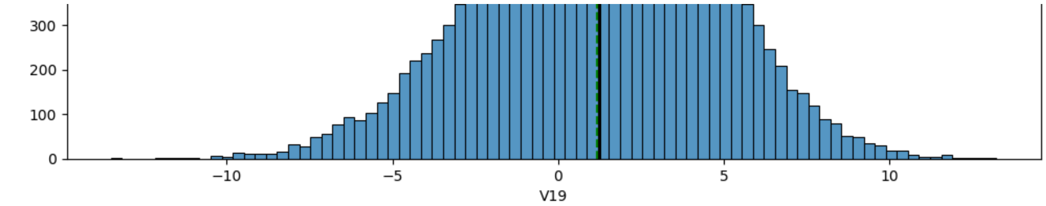

Count
V17

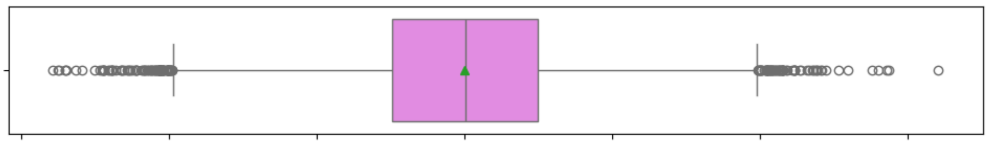


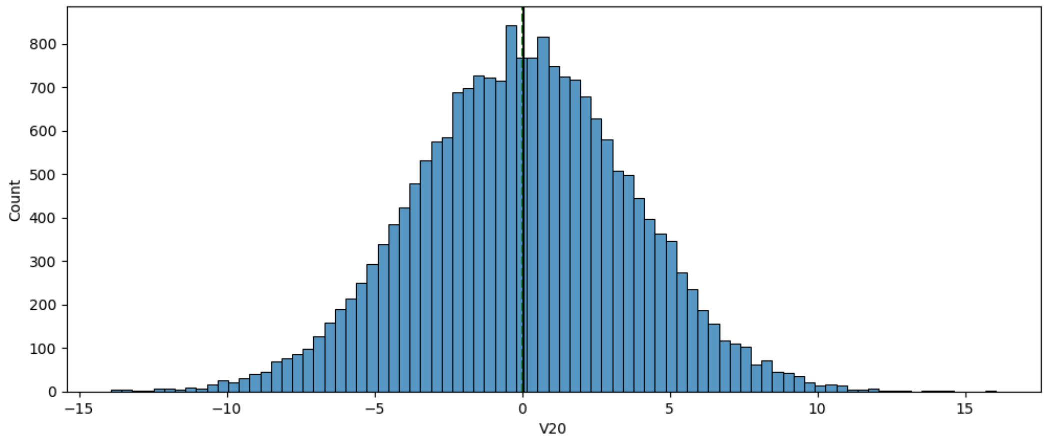

Count
V18

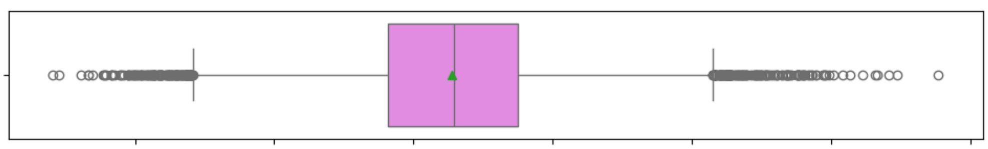


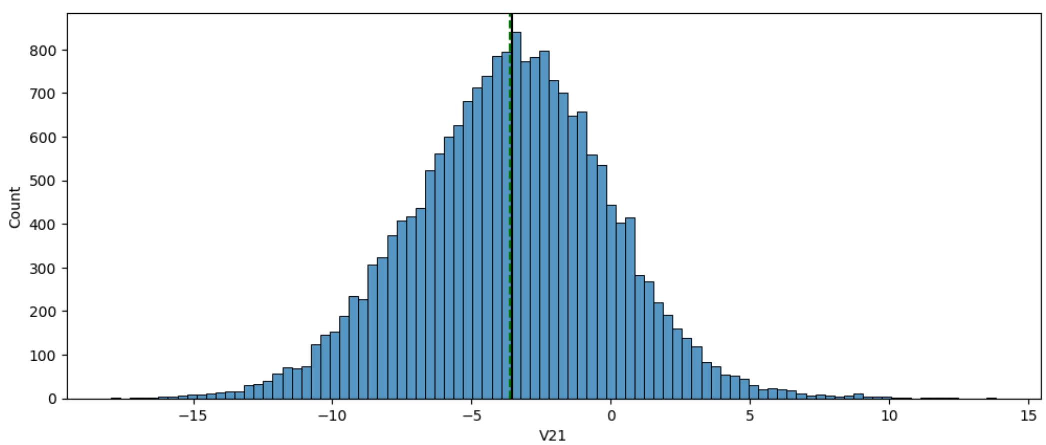

Count

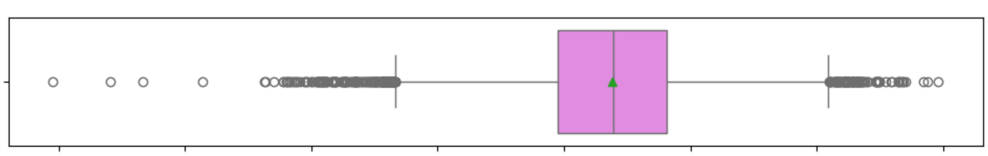

V19

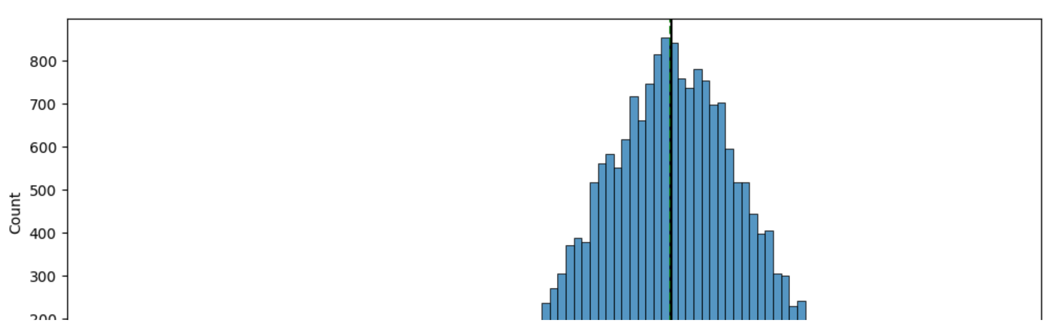


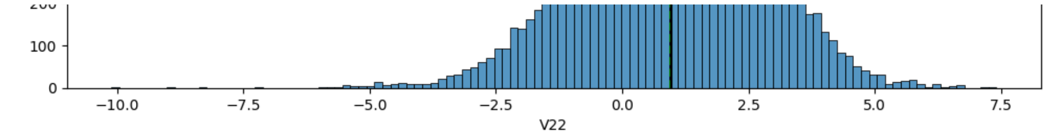

Count
V20

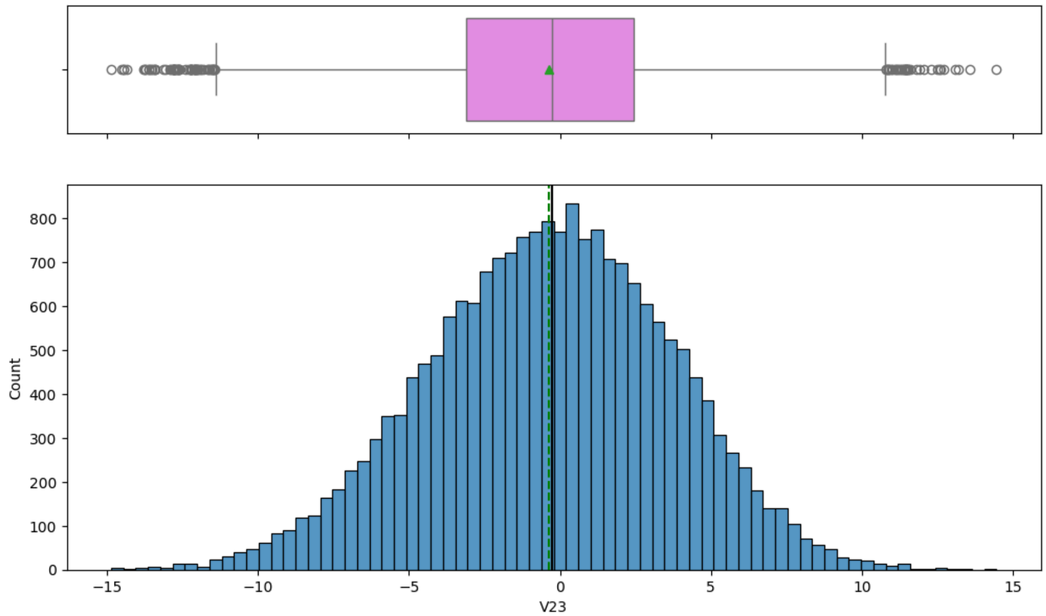


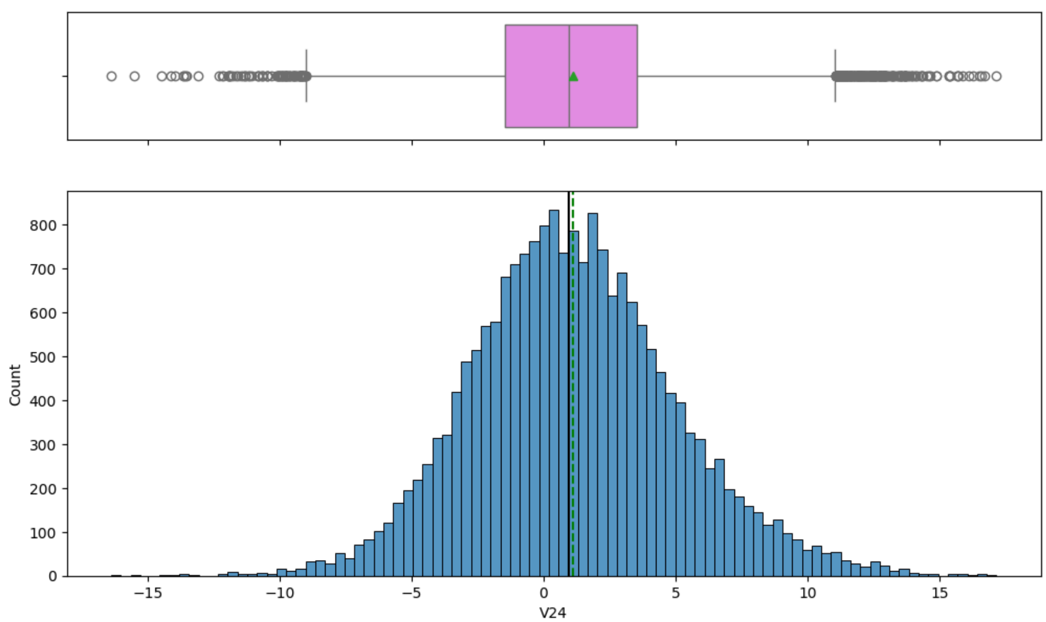

Count
V21

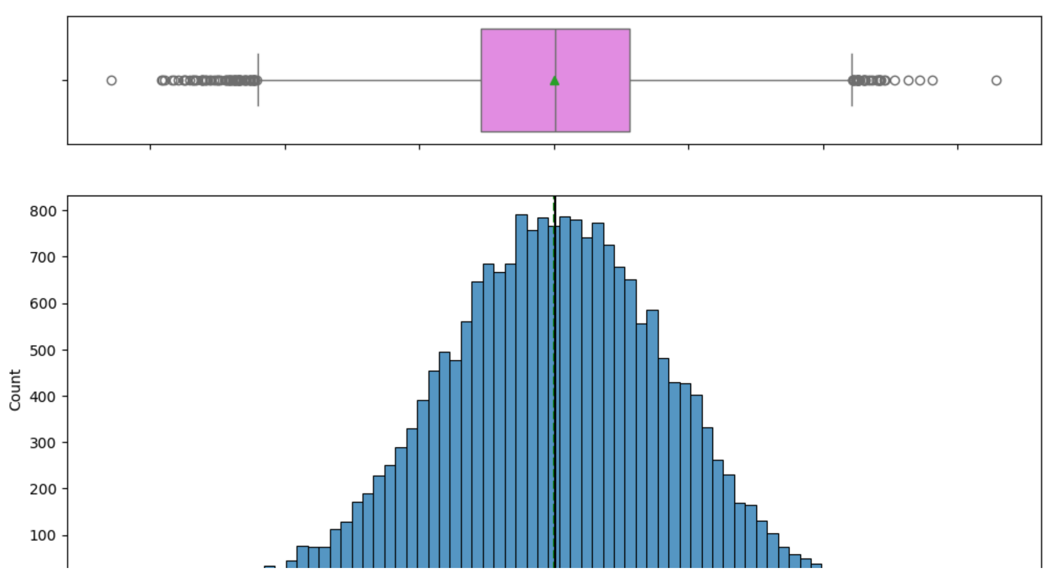


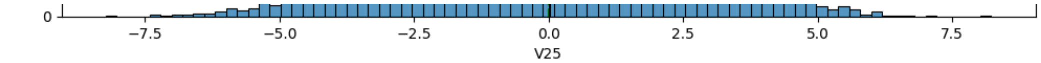

Count

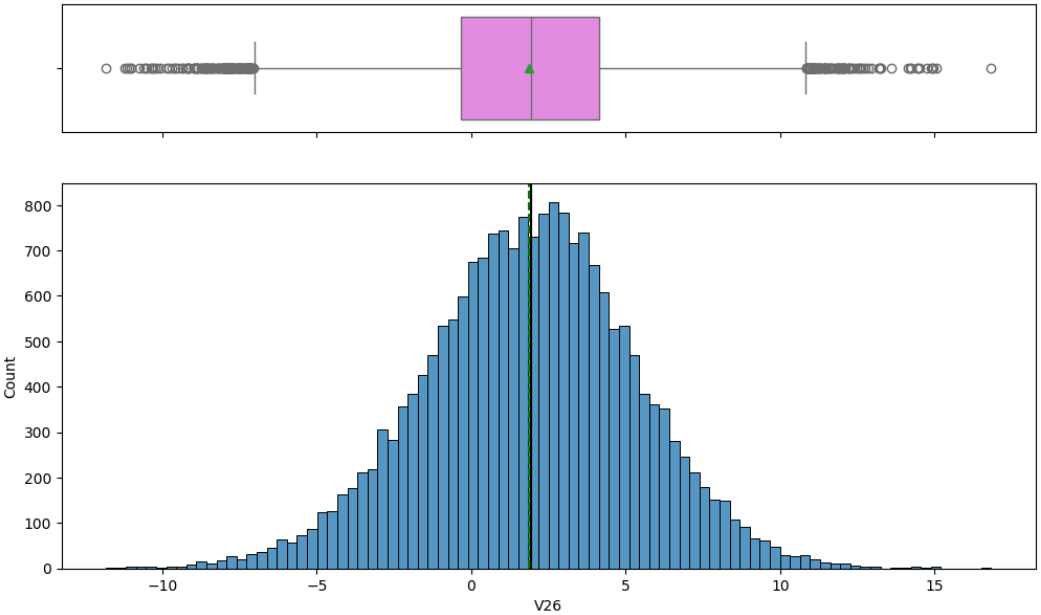

V22

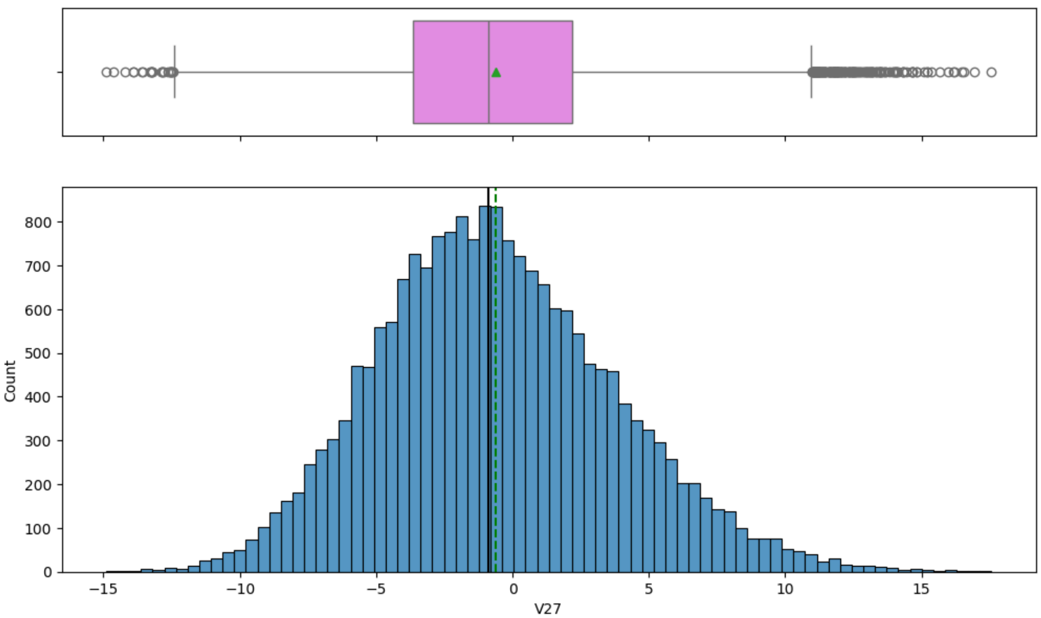

Count
V23

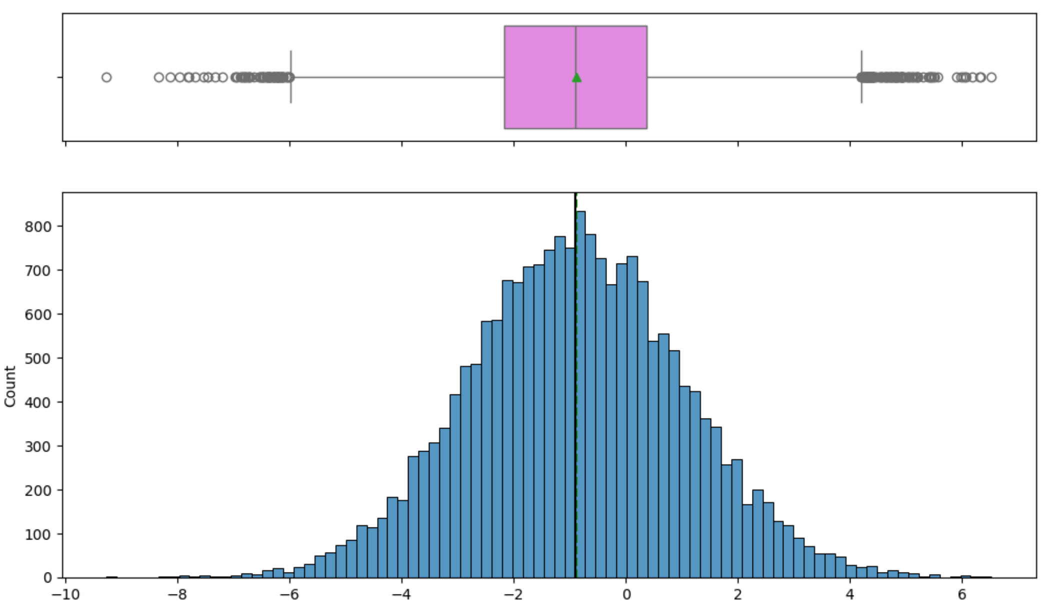

Count
V24

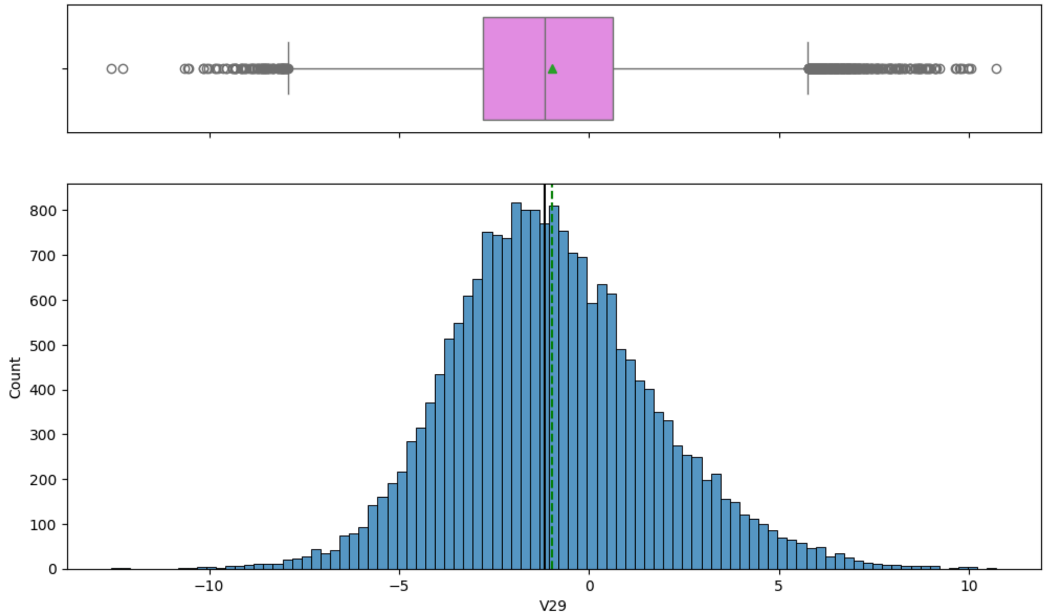

Count

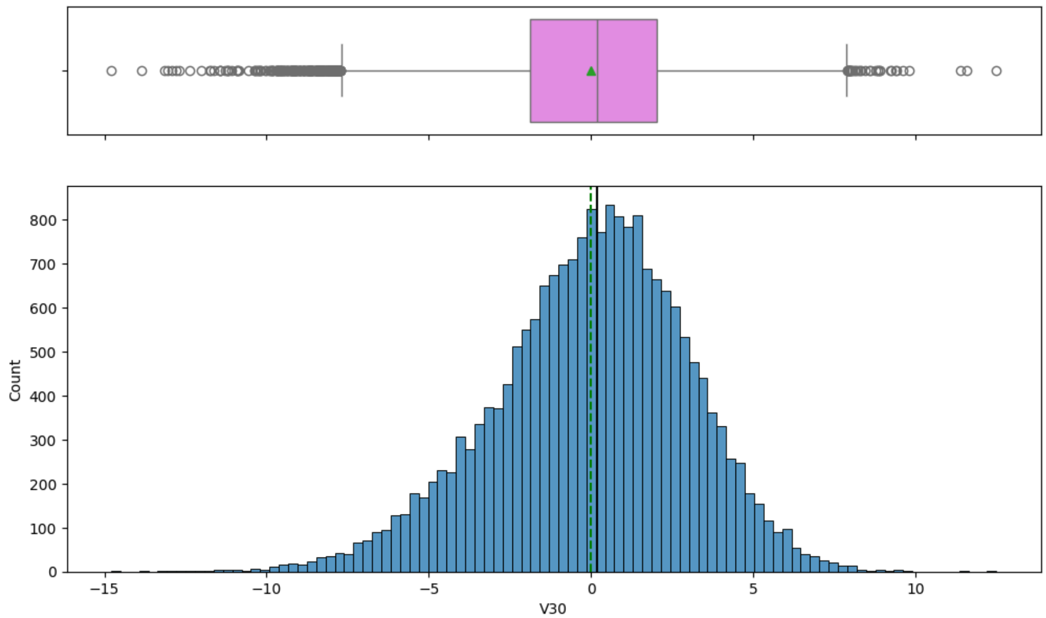

0
−7.5 −5.0 −2.5 0.0 2.5 5.0 7.5
V25

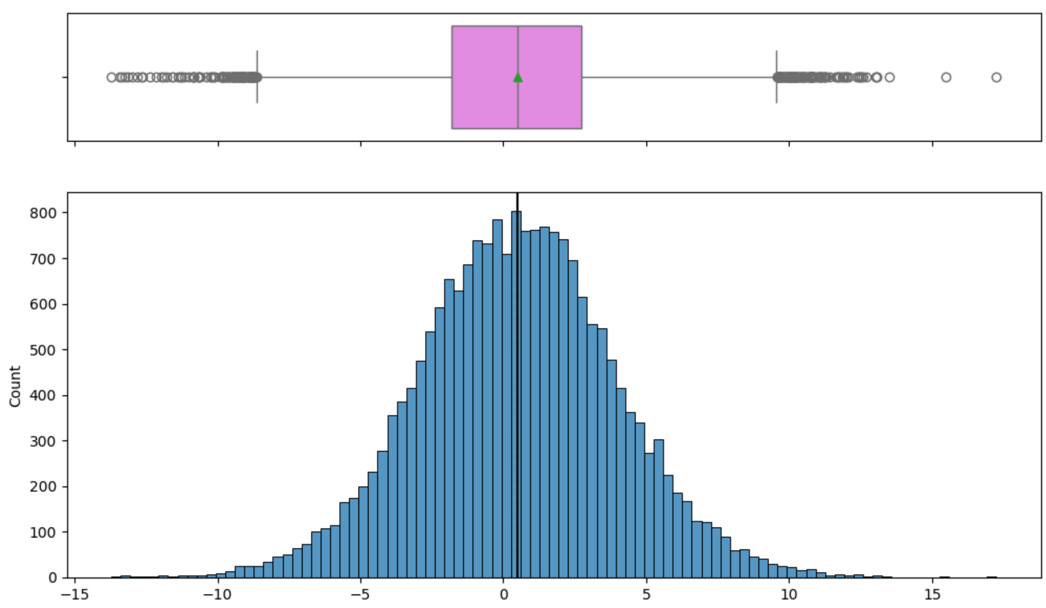

Count
0 100 200 300 400 500 600 700 800
−10 −5 0 5 10 15
V26

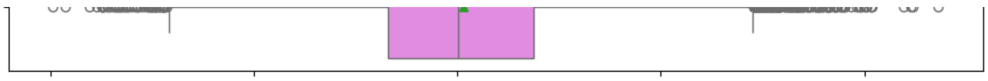

Count
0 100 200 300 400 500 600 700 800
−15 −10 −5 0 5 10 15
V27

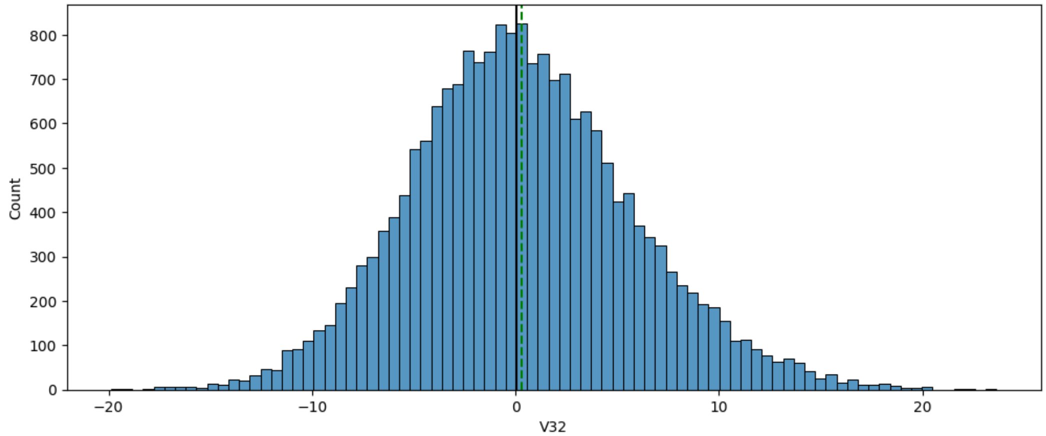

Count
0 100 200 300 400 500 600 700 800
−10 −8 −6 −4 −2 0 2 4 6
V28

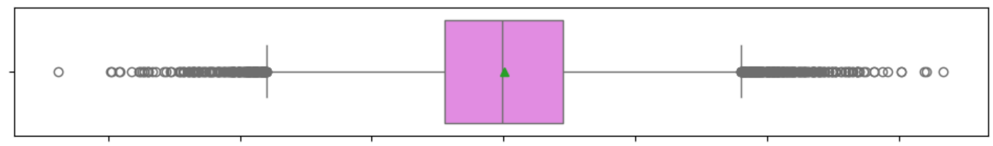

Count
V29

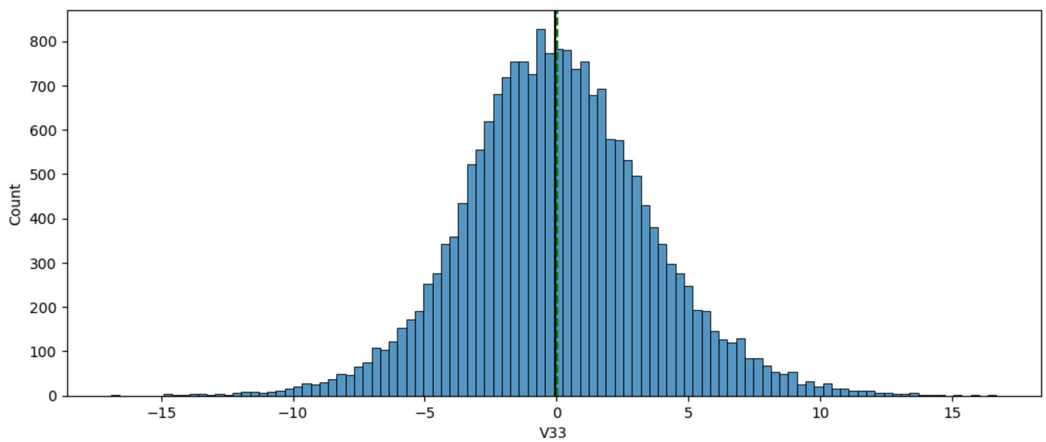

Count
V30

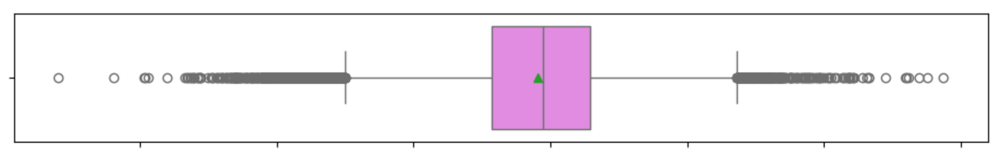

Count
V31

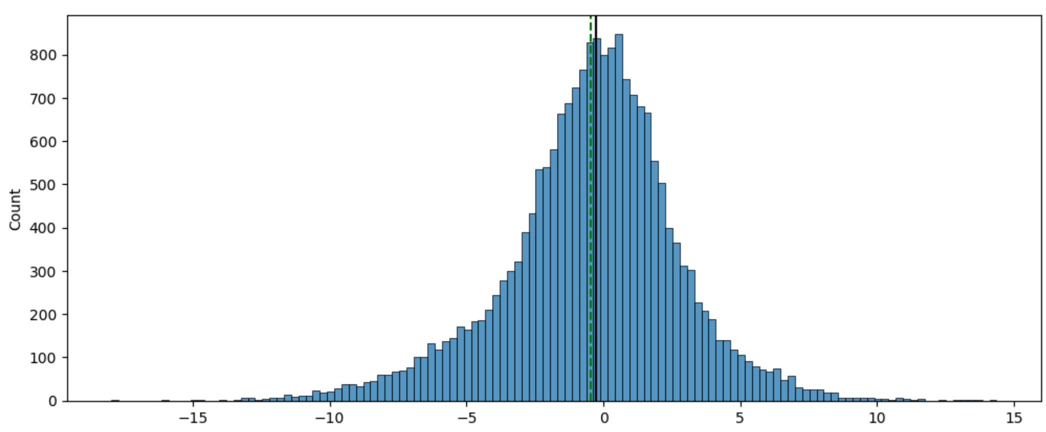


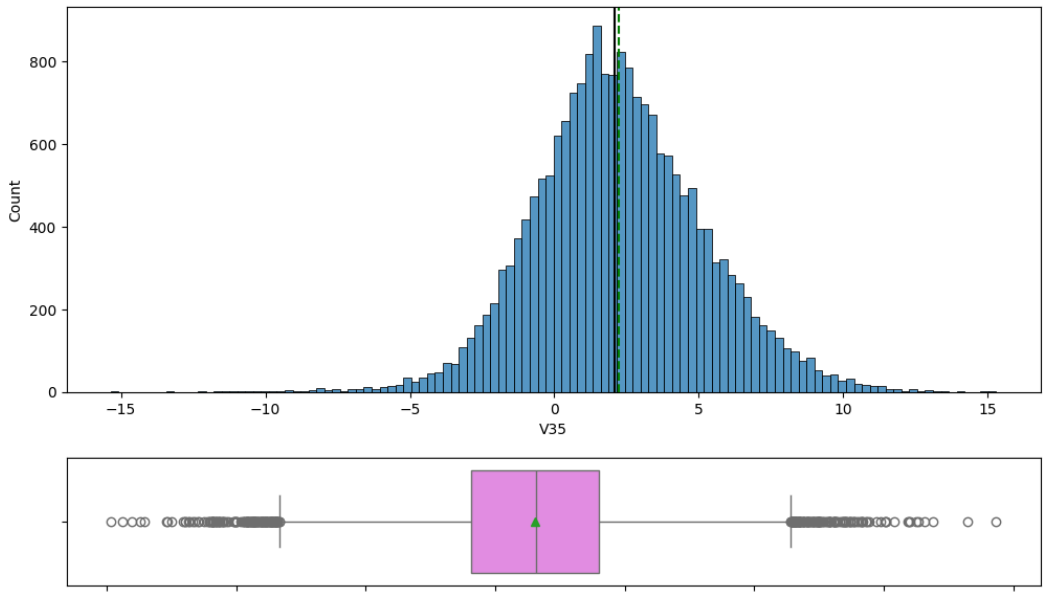

Count
V32

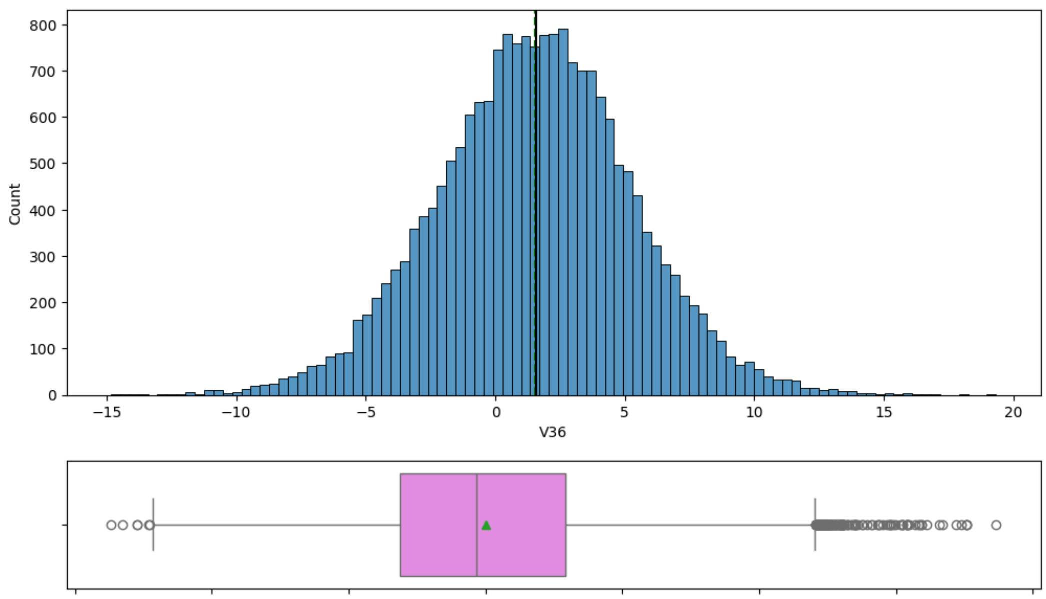


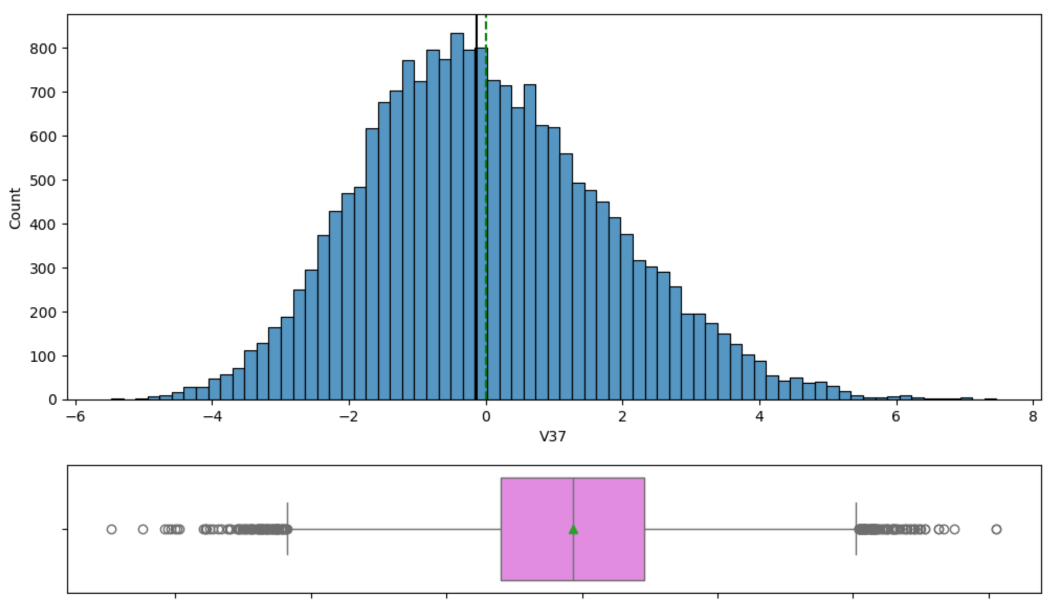

Count
V33



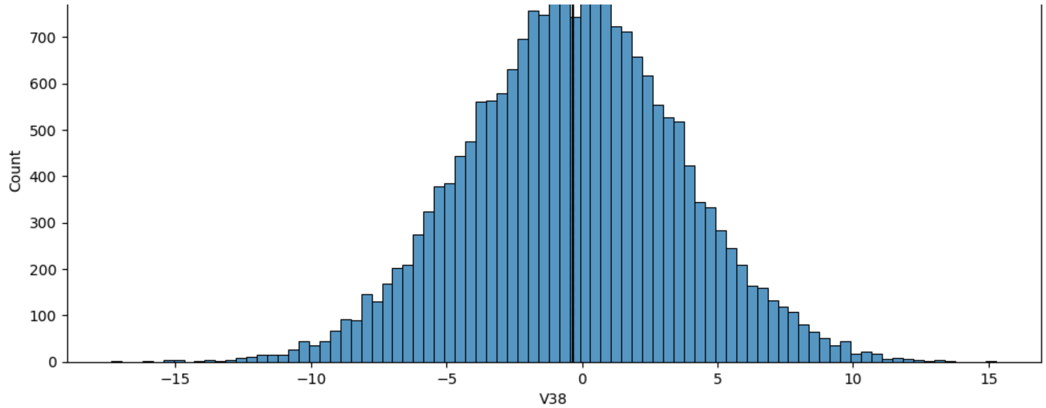

Count
V34

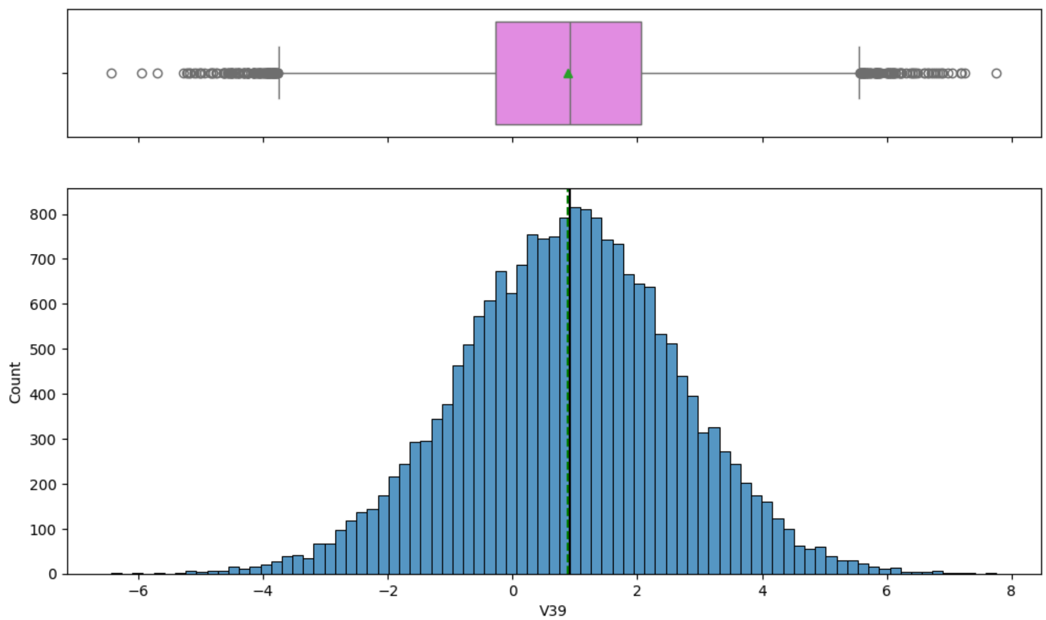

Count
V35
0
200
400
600
800
−15
−10
−5
0
5
10
15

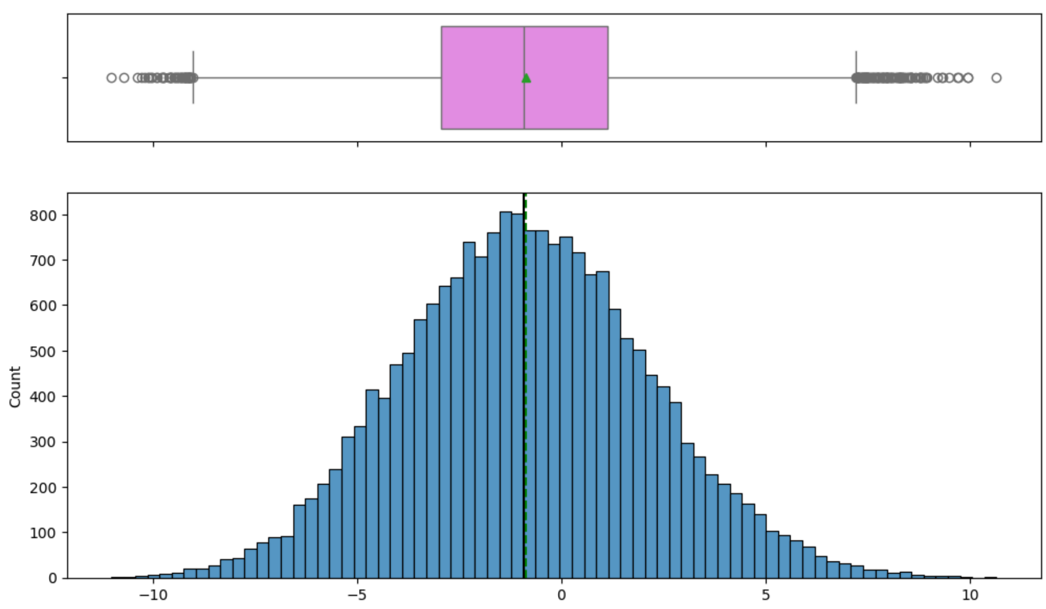

Count
V36
0
100
200
300
400
500
600
700
800
−15
−10
−5
0
5
10
15
20

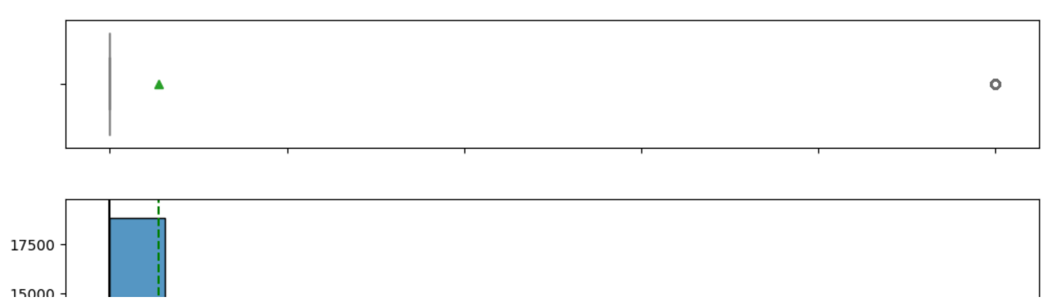

Count
V37
0
100
200
300
400
500
600
700
800
−6
−4
−2
0
2
4
6
8

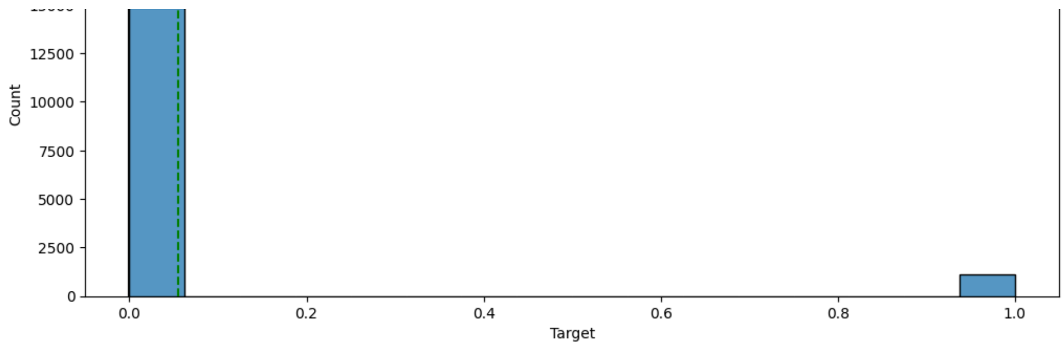

800


Count
V38


Count
V39


Count
V40




12500
10000
7500
5000
2500
0
Count
0.0
0.2
0.4
0.6
0.8
1.0
Target

Check the distribution of the target variable

```
df_train["Target"].value_counts(normalize=True)
```

	Target
0	0.945
1	0.056

```
df_test["Target"].value_counts(normalize=True)
```

	Target
0	0.944
1	0.056

Data Pre-processing

```
# Split train data
X = df_train.drop(["Target"], axis=1)
y = df_train["Target"]
```

```
# Split train dataset into training and validation set
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.25, random_state=42)
```

```
# Checking the number of rows and columns in the X_train data
X_train.shape
```

```
(15000, 40)
```

```
# Checking the number of rows and columns in the X_val data
X_val.shape
```

```
(5000, 40)
```

```
# Split test data
X_test = df_test.drop(["Target"], axis=1)
y_test = df_test["Target"]
```

```
# Checking the number of rows and columns in the X_test data
X_test.shape
```

```
(5000, 40)
```

Missing value imputation

```
# Create imputer
imputer = SimpleImputer(strategy="median")
```

```
# fit_transform the train data
X_train = pd.DataFrame(imputer.fit_transform(X_train), columns=X_train.columns)

# Transform - validation data
X_val = pd.DataFrame(imputer.transform(X_val), columns=X_train.columns)

# Transform - test data
X_test = pd.DataFrame(imputer.transform(X_test), columns=X_train.columns)
```

```
# Checking missing values
print(X_train.isna().sum())
print("-" * 80)
print(X_val.isna().sum())
print("-" * 80)
print(X_test.isna().sum())
```

```
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
```

Model Building

Model evaluation criterion

The nature of predictions made by the classification model will translate as follows:

- True positives (TP) are failures correctly predicted by the model.
- False negatives (FN) are real failures in a generator where there is no detection by model.
- False positives (FP) are failure detections in a generator where there is no failure.

Which metric to optimize?

- We need to choose the metric which will ensure that the maximum number of generator failures are predicted correctly by the model.
- We would want Recall to be maximized as greater the Recall, the higher the chances of minimizing false negatives.
- We want to minimize false negatives because if a model predicts that a machine will have no failure when there will be a failure, it will increase the maintenance cost.

Let's define a function to output different metrics (including recall) on the train and test set and a function to show confusion matrix so that we do not have to use the same code repetitively while evaluating models.

```
# 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
    """

    # 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 to be used for cross-validation and hyperparameter tuning

- We want to reduce false negatives and will try to maximize "Recall".
- To maximize Recall, we can use Recall as a **scorer** in cross-validation and hyperparameter tuning.

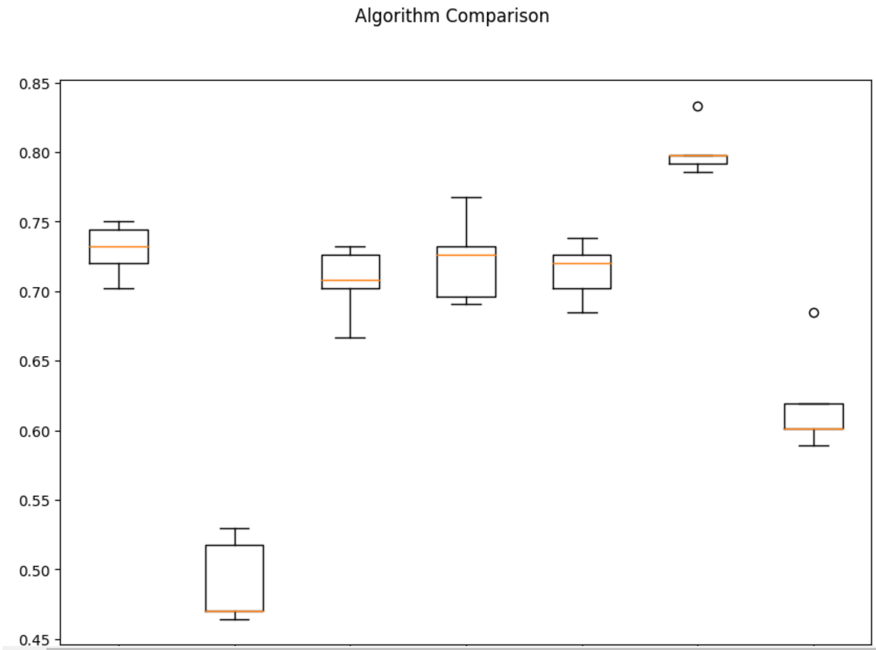
```
# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.recall_score)
```

Model Building with original data

```
models = []  # Empty list to store all the models

# Appending models into the list
models.append(("Decision Tree Classifier", DecisionTreeClassifier(random_state=1)))
models.append(("Logistic Regression", LogisticRegression(random_state=1)))
models.append(("Bagging Classifier", BaggingClassifier(random_state=1)))
models.append(("Random Forest", RandomForestClassifier(random_state=1)))
models.append(("Gradient Boosting", GradientBoostingClassifier(random_state=1)))
models.append(("XGBoost", XGBClassifier(random_state=1)))
models.append(("AdaBoost", AdaBoostClassifier(random_state=1)))

results1 = []  # Empty list to store all model's CV scores
names = []  # Empty list to store name of the models

# loop through all models to get the mean cross validated score
print("\n" "Cross-Validation performance on training dataset:" "\n")

for name, model in models:
    kfold = StratifiedKFold(
        n_splits=5, shuffle=True, random_state=1
    )  # Setting number of splits equal to 5
    cv_result = cross_val_score(
        estimator=model, X=X_train, y=y_train, scoring=scorer, cv=kfold
    )
    results1.append(cv_result)
    names.append(name)
    print("{}: {}".format(name, cv_result.mean()))

print("\n" "Validation Performance:" "\n")

for name, model in models:
    model.fit(X_train, y_train)
    scores = recall_score(y_val, model.predict(X_val))
    print("{}: {}".format(name, scores))
```

```
Cross-Validation performance on training dataset:

Decision Tree Classifier: 0.7297619047619047
Logistic Regression: 0.4904761904761905
Bagging Classifier: 0.7071428571428572
Random Forest: 0.7226190476190476
Gradient Boosting: 0.7142857142857142
XGBoost: 0.8011904761904761
AdaBoost: 0.6190476190476192

Validation Performance:

Decision Tree Classifier: 0.7111111111111111
Logistic Regression: 0.48148148148148145
Bagging Classifier: 0.7222222222222222
Random Forest: 0.6962962962962963
Gradient Boosting: 0.6888888888888889
XGBoost: 0.8
AdaBoost: 0.5777777777777777
```

```
# Boxplots for all models defined above
fig = plt.figure(figsize=(10, 7))

fig.suptitle("Algorithm Comparison")
ax = fig.add_subplot(111)

plt.boxplot(results1)
ax.set_xticklabels(names)

plt.show()
```



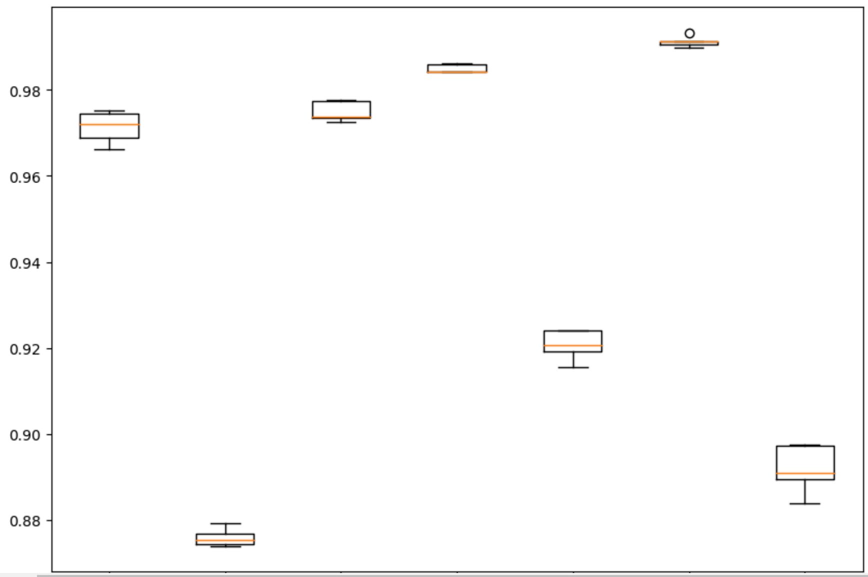
Model Building with Oversampled data

```
# Synthetic Minority Over Sampling Technique
sm = SMOTE(sampling_strategy=1, k_neighbors=5, random_state=1)
X_train_over, y_train_over = sm.fit_resample(X_train, y_train)


models = []  # Empty list to store all the models

# Appending models into the list
models.append(("Decision Tree Classifier", DecisionTreeClassifier(random_state=1)))
models.append(("Logistic Regression", LogisticRegression(random_state=1)))
models.append(("Bagging Classifier", BaggingClassifier(random_state=1)))
models.append(("Random Forest", RandomForestClassifier(random_state=1)))
models.append(("Gradient Boosting", GradientBoostingClassifier(random_state=1)))
models.append(("XGBoost", XGBClassifier(random_state=1)))
models.append(("AdaBoost", AdaBoostClassifier(random_state=1)))

results1 = []  # Empty list to store all model's CV scores
names = []  # Empty list to store name of the models

# loop through all models to get the mean cross validated score
print("\n" "Cross-Validation performance on training dataset:" "\n")

for name, model in models:
    kfold = StratifiedKFold(
        n_splits=5, shuffle=True, random_state=1
    )  # Setting number of splits equal to 5
    cv_result = cross_val_score(
        estimator=model, X=X_train_over, y=y_train_over, scoring=scorer, cv=kfold
    )
    results1.append(cv_result)
    names.append(name)
    print("{}: {}".format(name, cv_result.mean()))

print("\n" "Validation Performance:" "\n")

for name, model in models:
    model.fit(X_train_over, y_train_over)
    scores = recall_score(y_val, model.predict(X_val))
    print("{}: {}".format(name, scores))
```

```
Cross-Validation performance on training dataset:

Decision Tree Classifier: 0.9713983050847457
Logistic Regression: 0.8759180790960451
Bagging Classifier: 0.975
Random Forest: 0.9848870056497174
Gradient Boosting: 0.9206920903954803
XGBoost: 0.9911723163841808
AdaBoost: 0.8918079096045199

Validation Performance:

Decision Tree Classifier: 0.7851851851851852
Logistic Regression: 0.8518518518518519
Bagging Classifier: 0.8148148148148148
Random Forest: 0.8407407407407408
Gradient Boosting: 0.8629629629629629
XGBoost: 0.8592592592592593
AdaBoost: 0.8555555555555555
```

```
# Boxplots for all models defined above
fig = plt.figure(figsize=(10, 7))

fig.suptitle("Algorithm Comparison")
ax = fig.add_subplot(111)

plt.boxplot(results1)
ax.set_xticklabels(names)

plt.show()
```

Algorithm Comparison



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)


models = []  # Empty list to store all the models

# Appending models into the list
models.append(("Decision Tree Classifier", DecisionTreeClassifier(random_state=1)))
models.append(("Logistic Regression", LogisticRegression(random_state=1)))
models.append(("Bagging Classifier", BaggingClassifier(random_state=1)))
models.append(("Random Forest", RandomForestClassifier(random_state=1)))
models.append(("Gradient Boosting", GradientBoostingClassifier(random_state=1)))
models.append(("XGBoost", XGBClassifier(random_state=1)))
models.append(("AdaBoost", AdaBoostClassifier(random_state=1)))

results1 = []  # Empty list to store all model's CV scores
names = []  # Empty list to store name of the models

# loop through all models to get the mean cross validated score
print("\n" "Cross-Validation performance on training dataset:" "\n")

for name, model in models:
    kfold = StratifiedKFold(
        n_splits=5, shuffle=True, random_state=1
    )  # Setting number of splits equal to 5
    cv_result = cross_val_score(
        estimator=model, X=X_train_un, y=y_train_un, scoring=scorer, cv=kfold
    )
    results1.append(cv_result)
    names.append(name)
    print("{}: {}".format(name, cv_result.mean()))

print("\n" "Validation Performance:" "\n")

for name, model in models:
    model.fit(X_train_un, y_train_un)
    scores = recall_score(y_val, model.predict(X_val))
    print("{}: {}".format(name, scores))
```

```
Cross-Validation performance on training dataset:

Decision Tree Classifier: 0.8678571428571427
Logistic Regression: 0.855952380952381
Bagging Classifier: 0.8738095238095237
Random Forest: 0.8988095238095237
Gradient Boosting: 0.8952380952380953
```