ReneWind Project

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 libraries

In [1]:

```
# suppress all warnings
import warnings
warnings.filterwarnings("ignore")

#import libraries needed for data manipulation
import pandas as pd
import numpy as np

pd.set_option("display.float_format", lambda x: "%.3f" % x)

#import libraries needed for data visualization

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# using statsmodels to build our model
import statsmodels.stats.api as sms
import statsmodels.api as sm

# unlimited number of displayed columns and rows
pd.set_option("display.max_columns", None)
pd.set_option("display.max_rows", None)

# split the data into random train and test subsets
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score

# Libraries different ensemble classifiers
from sklearn.ensemble import (
    BaggingClassifier,
    RandomForestClassifier,
    AdaBoostClassifier,
    GradientBoostingClassifier,
    StackingClassifier,
)

from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression

# Libraries to get different metric scores
from sklearn import metrics
from sklearn.metrics import (
    f1_score,
    confusion_matrix,
    accuracy_score,
    precision_score,
    recall_score,
    roc_auc_score,
    plot_confusion_matrix,
)
```

```
# 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 tune different models
from sklearn.model_selection import GridSearchCV
```

In [2]:

```
#import datasets named 'Train.csv' and 'Test.csv'

data = pd.read_csv('Train.csv')
df_test = pd.read_csv('Test.csv')

# read first five rows of the training dataset

data.head()
```

Out[2]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	
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
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
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
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
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

In [3]:

```
# read last five rows of dataset

data.tail()
```

Out[3]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V2
19995	-2.071	-1.088	-0.796	-3.012	-2.288	2.807	0.481	0.105	-0.587	-2.899	8.868	1.717	1.358	-1.777	0.710	4.945	-3.100	-1.199	-1.085	-0.36

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V2
19996	2.890	2.483	5.644	0.937	-1.381	0.412	-1.593	-5.762	2.150	0.272	-2.095	-1.526	0.072	-3.540	-2.762	-10.632	-0.495	1.720	3.872	-1.21
19997	-3.897	-3.942	-0.351	-2.417	1.108	-1.528	-3.520	2.055	-0.234	-0.358	-3.782	2.180	6.112	1.985	-8.330	-1.639	-0.915	5.672	-3.924	2.13
19998	-3.187	-10.052	5.696	-4.370	-5.355	-1.873	-3.947	0.679	-2.389	5.457	1.583	3.571	9.227	2.554	-7.039	-0.994	-9.665	1.155	3.877	3.52
19999	-2.687	1.961	6.137	2.600	2.657	-4.291	-2.344	0.974	-1.027	0.497	-9.589	3.177	1.055	-1.416	-4.669	-5.405	3.720	2.893	2.329	1.45

In [4]:

```
#check shape of training dataset

data.shape
```

Out[4]: (20000, 41)

In [5]:

```
#check shape of test dataset

df_test.shape
```

Out[5]: (5000, 41)

Data Overview

- Observations
- Sanity checks

In [6]:

```
df = data.copy()
```

In [7]:

```
test = df_test.copy()
```

In [8]:

```
df.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
```

In [9]:

```
df.isnull().sum()
```

Out[9]:

```
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
```

In [10]:

```
test.isnull().sum()
```

Out[10]:

```
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
```

In [11]:

```
df.duplicated().sum()
```

Out[11]: 0

In [12]:

```
test.duplicated().sum()
```

Out[12]: 0

In [13]:

```
df.describe().T
```

Out[13]:

	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

	count	mean	std	min	25%	50%	75%	max
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

Observations

- Train set
    - 20000 rows

- Test set
  - 5000 rows
- 41 variables: 40 predictors, 1 target
- First two predictor variables have a few missing values
- No duplicate values
- Not much variation in means across predictor variables
- Some ranges are surpsingly large
- Target variable ranges from 0-1, averaging at 0.056

EDA

Univariate Analysis

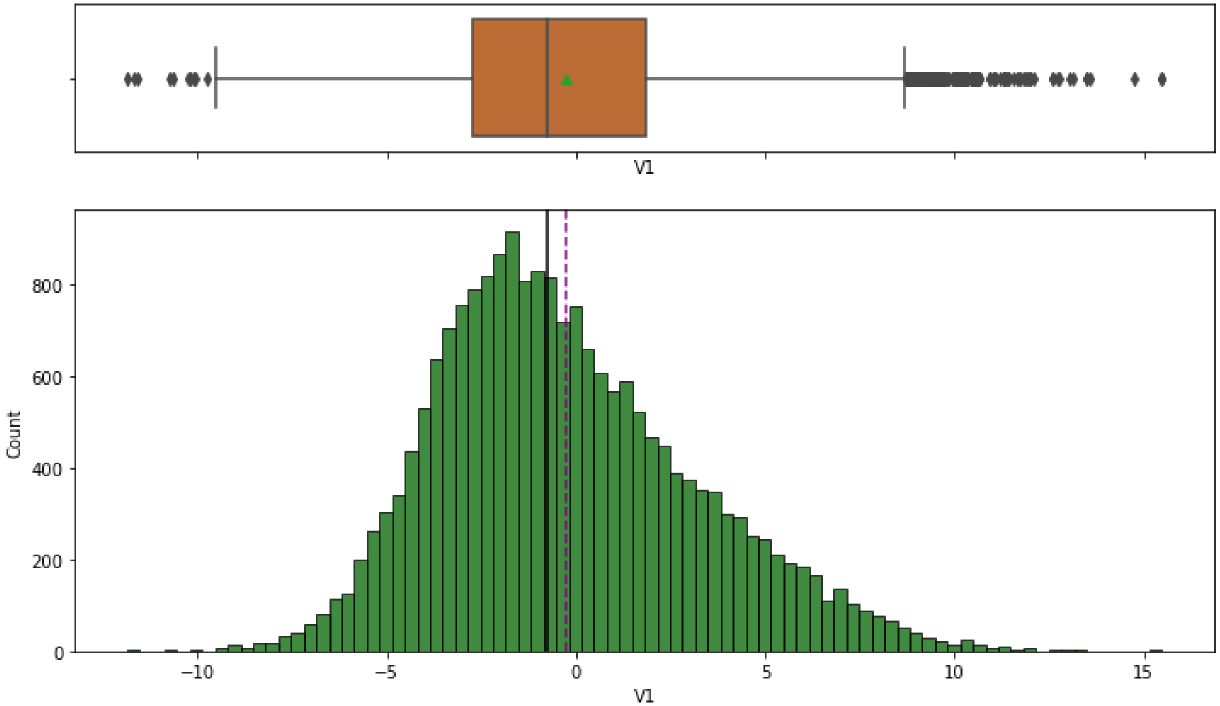
In [14]:

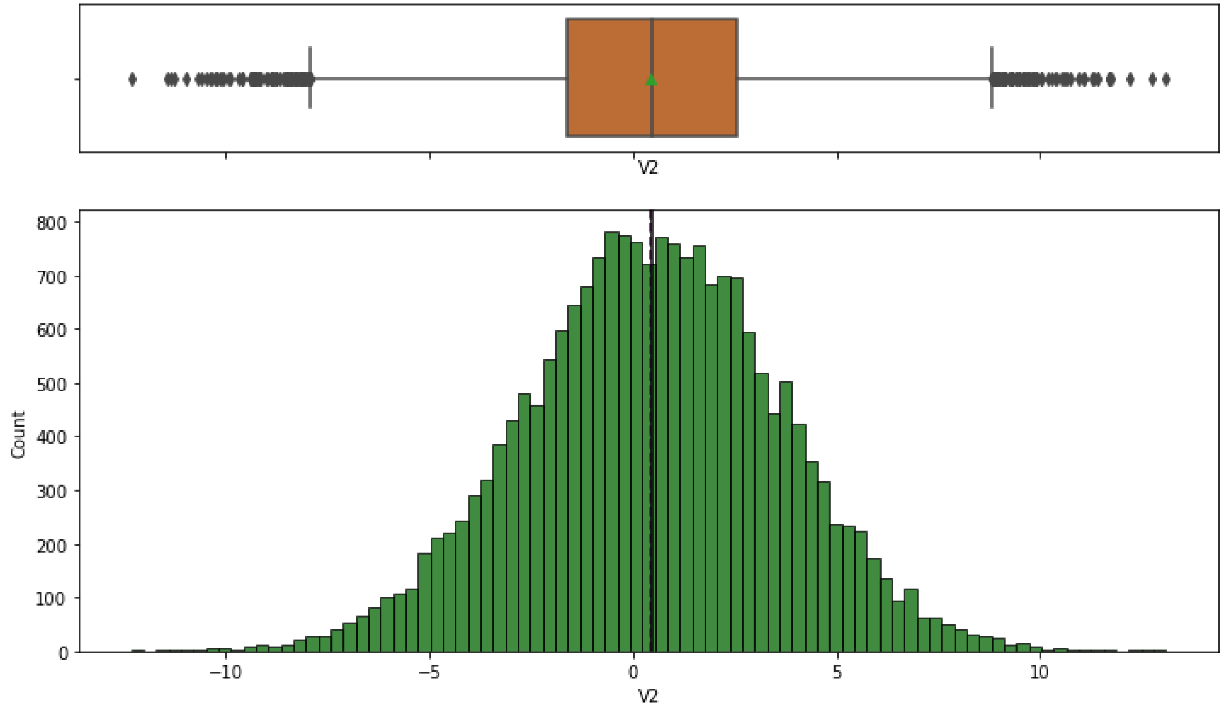
```
# define a function to plot a boxplot and a histogram along the same scale

def histbox(data, feature, figsize=(12, 7), kde=False, bins=None):
    """
    Boxplot and histogram combined
    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (12,7))
    kde: whether to show the density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (box, hist) = plt.subplots(
        nrows=2,                                        # Number of rows of the subplot grid = 2
                                                            # boxplot first then histogram created below
        sharex=True,                                    # x-axis same among all subplots
        gridspec_kw={"height_ratios": (0.25, 0.75)},    # boxplot 1/3 height of histogram
        figsize=figsize,                                # figsize defined above as (12, 7)
    )
    # defining boxplot inside function, so when using it say histbox(df, 'cost'), df: data and cost: feature

    sns.boxplot(
        data=data, x=feature, ax=box, showmeans=True, color="chocolate"
    )  # showmeans makes mean val on boxplot have star, ax =
    sns.histplot(
        data=data, x=feature, kde=kde, ax=hist, bins=bins, color = "darkgreen"
    ) if bins else sns.histplot(
        data=data, x=feature, kde=kde, ax=hist, color = "darkgreen"
    )  # For histogram if there are bins in potential graph

    # add vertical line in histogram for mean and median
    hist.axvline(
        data[feature].mean(), color="purple", linestyle="--"
    )  # Add mean to the histogram
    hist.axvline(
```

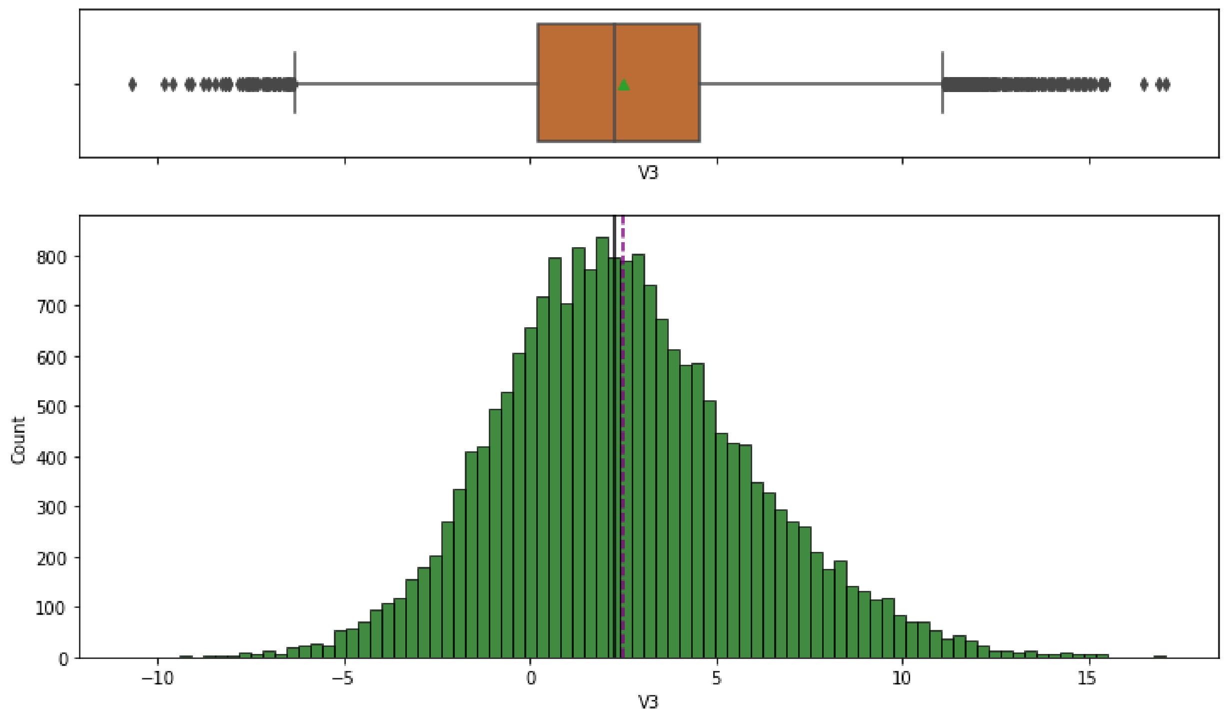
```
        data[feature].median(), color="black", linestyle="-"
    )  # Add median to the histogram
```

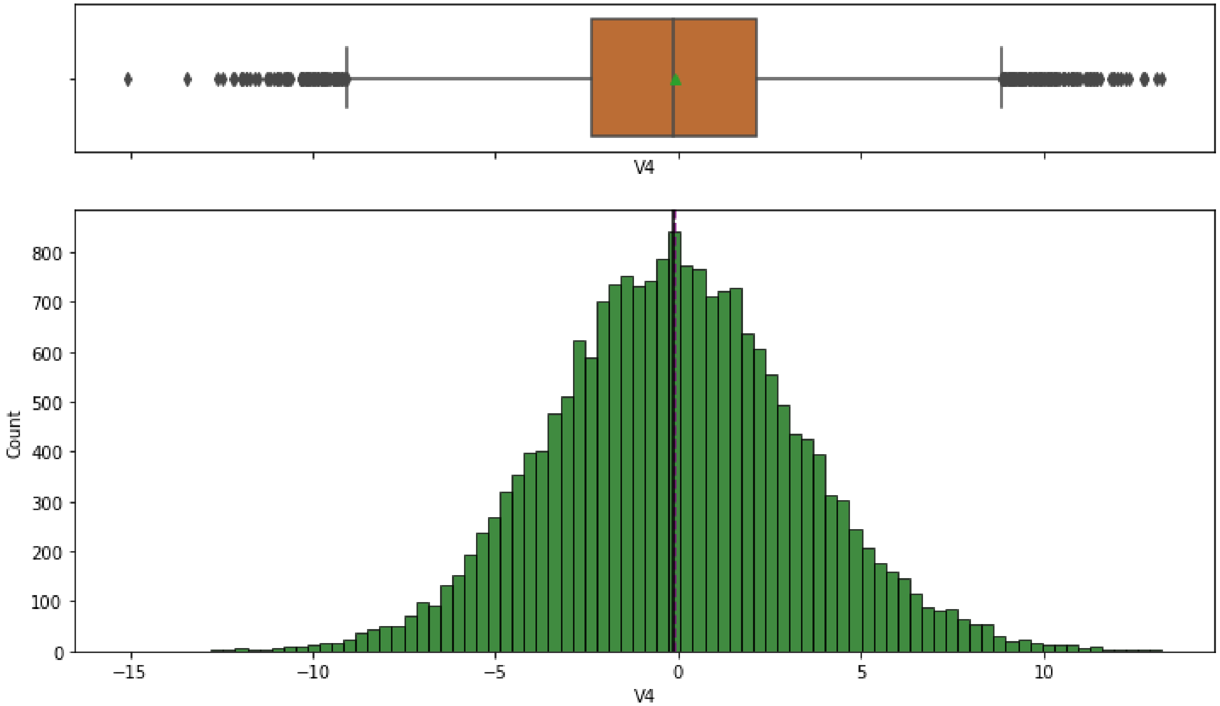
In [15]:

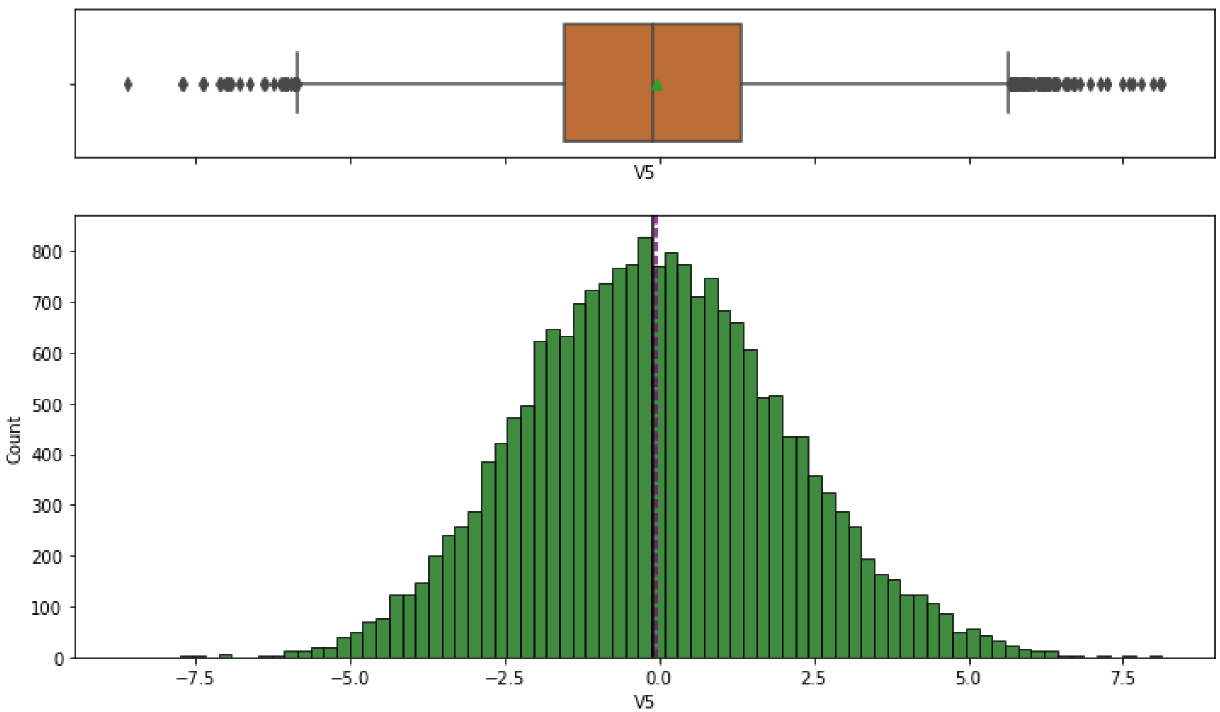
```
# plot histogram/boxplot for all variables

for feature in data.columns:
    histbox(df, feature, figsize=(12, 7), kde=False, bins=None)
```

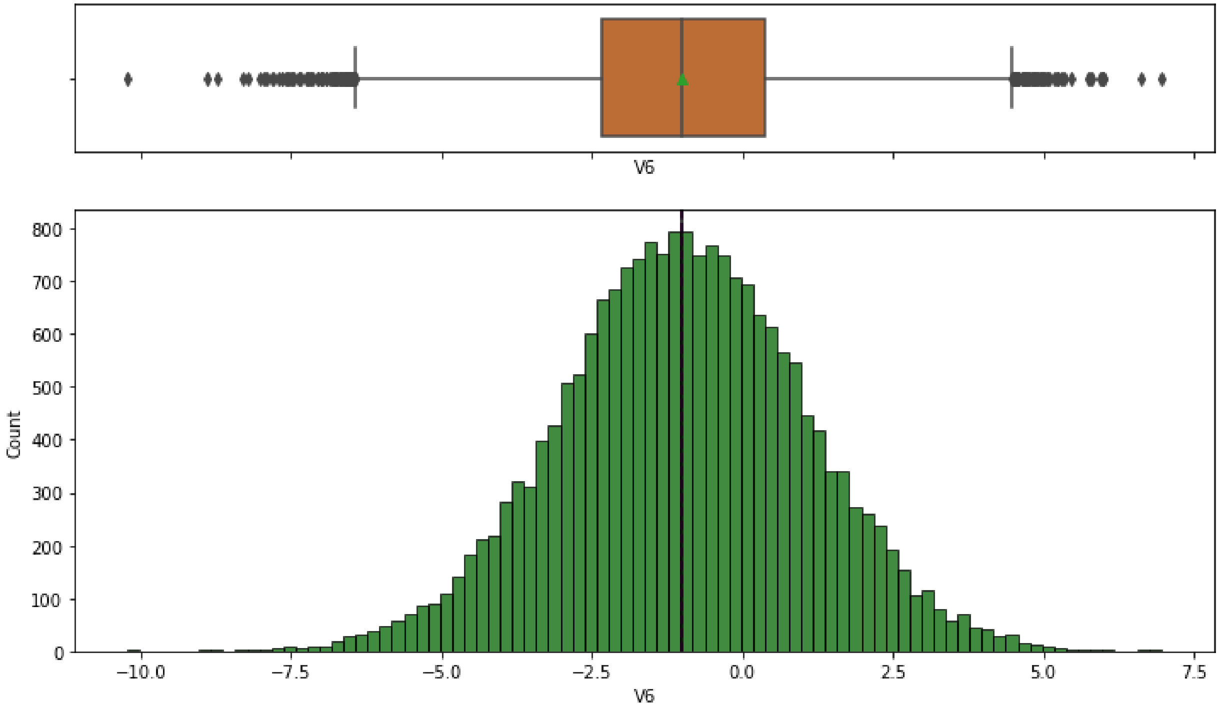


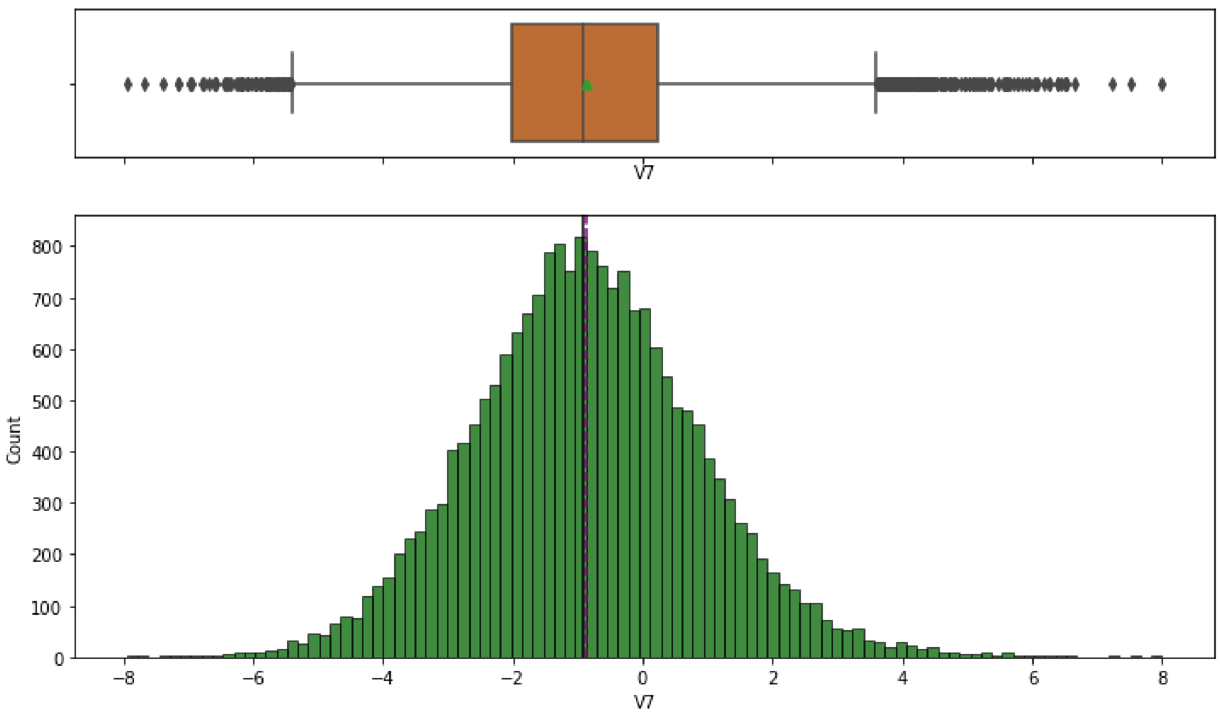

V2
800
700
600
500
400
300
200
100
0
Count
−10
−5
0
5
10
V2

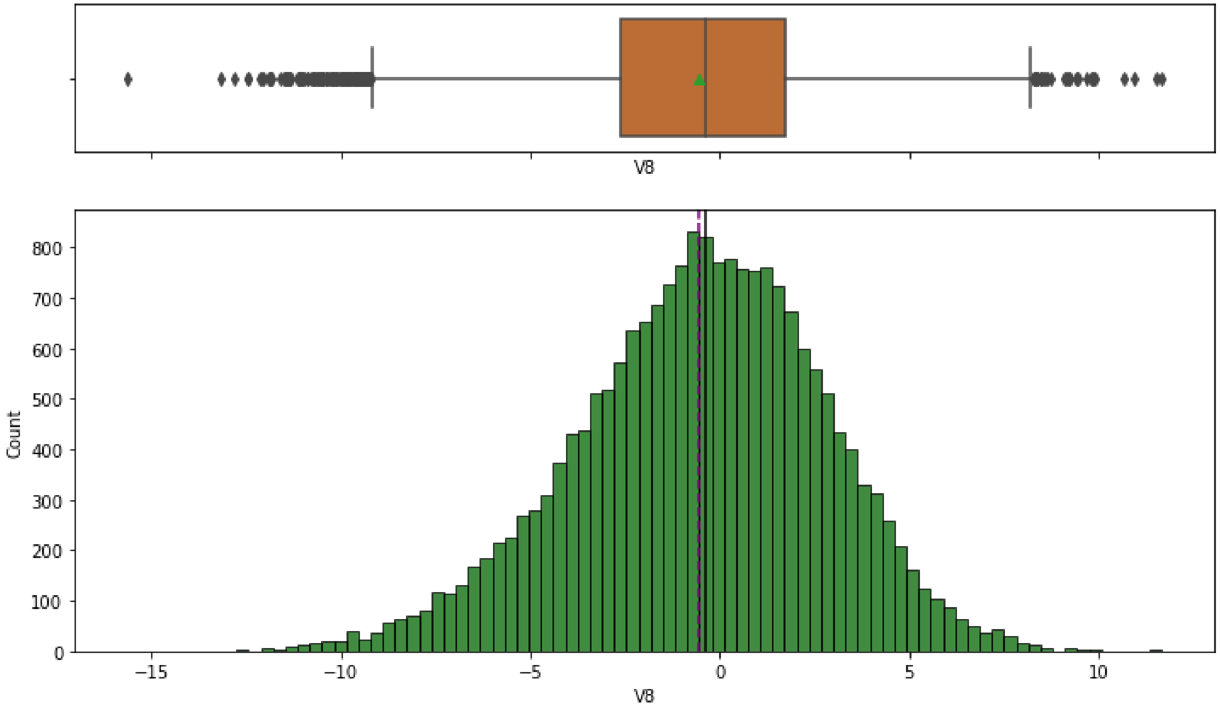

V3
800
700
600
500
400
300
200
100
0
Count
−10
−5
0
5
10
15
V3

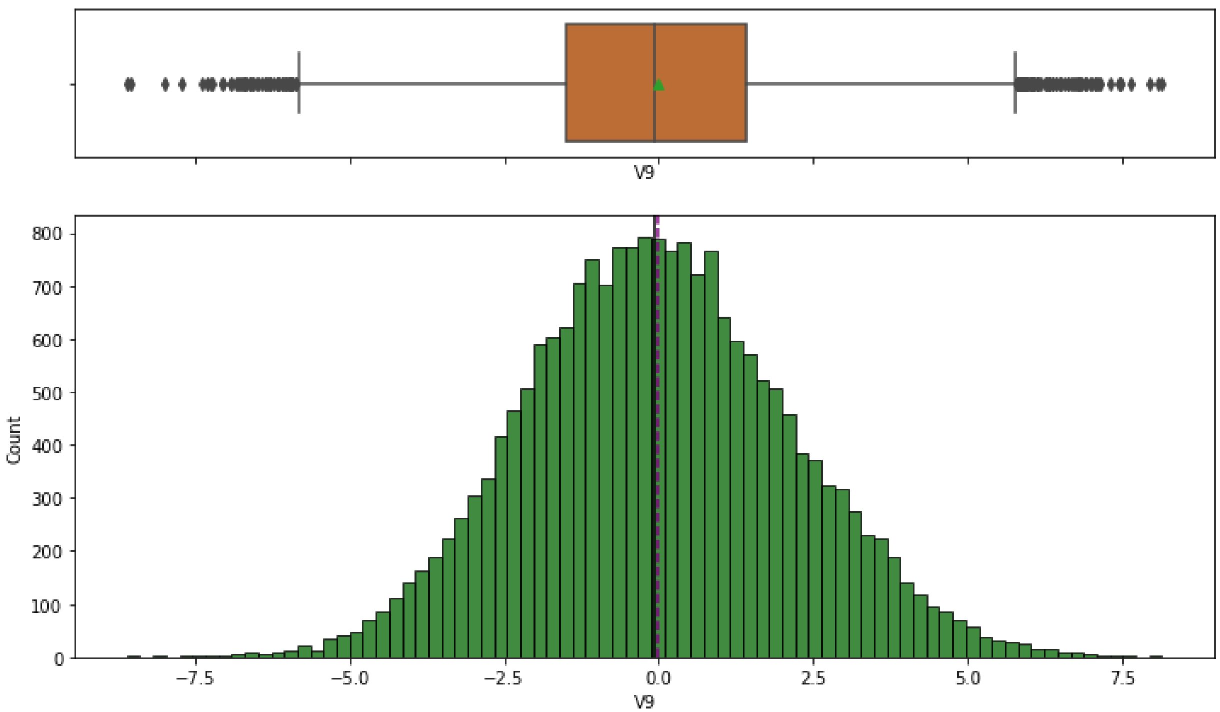

V4
Count
V4

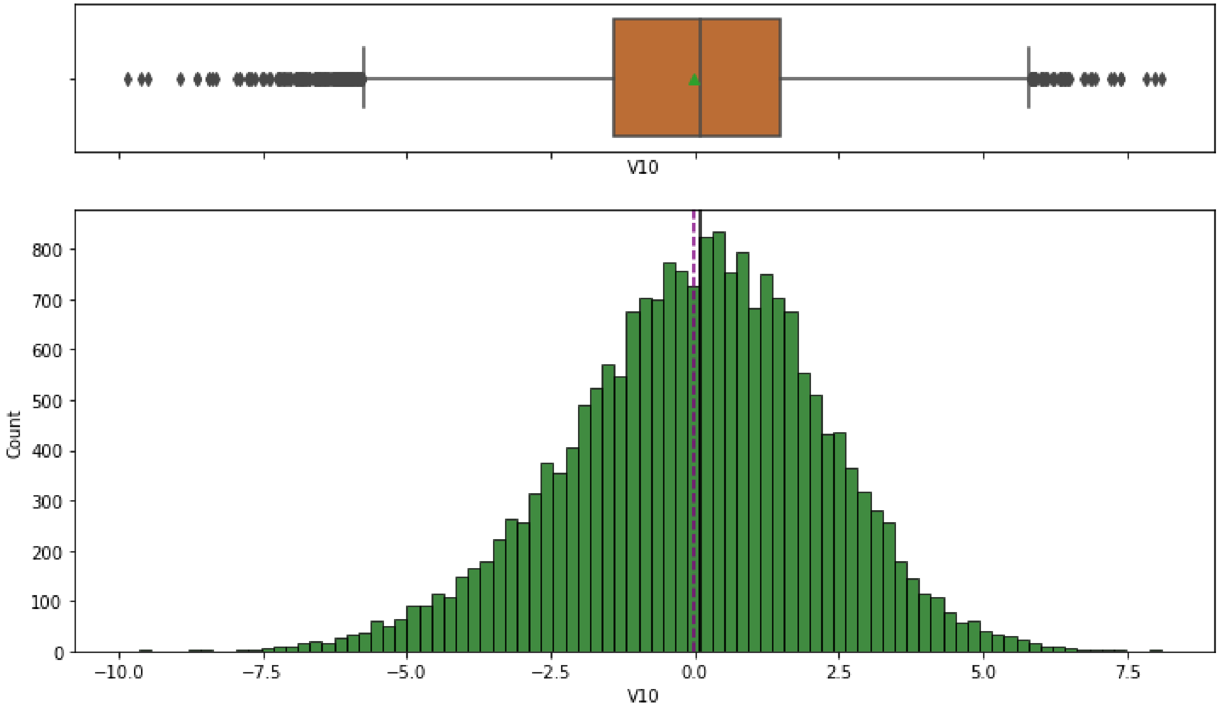

V5
Count
V5

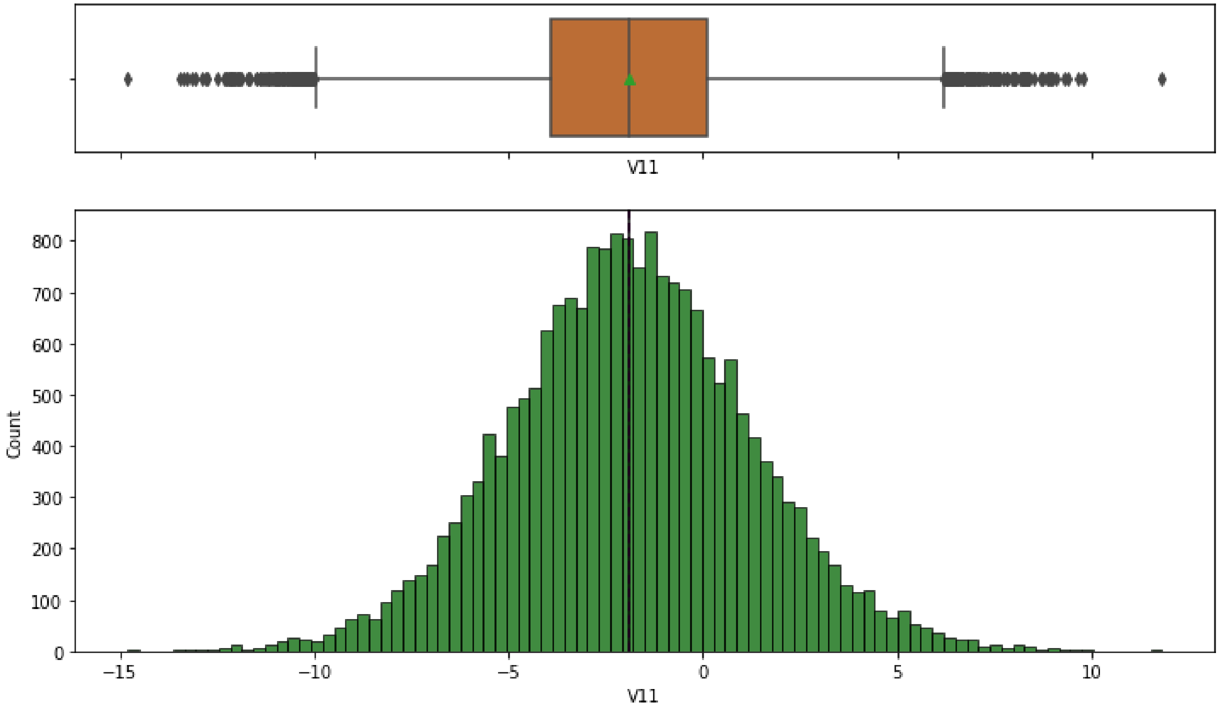

V6
Count
V6

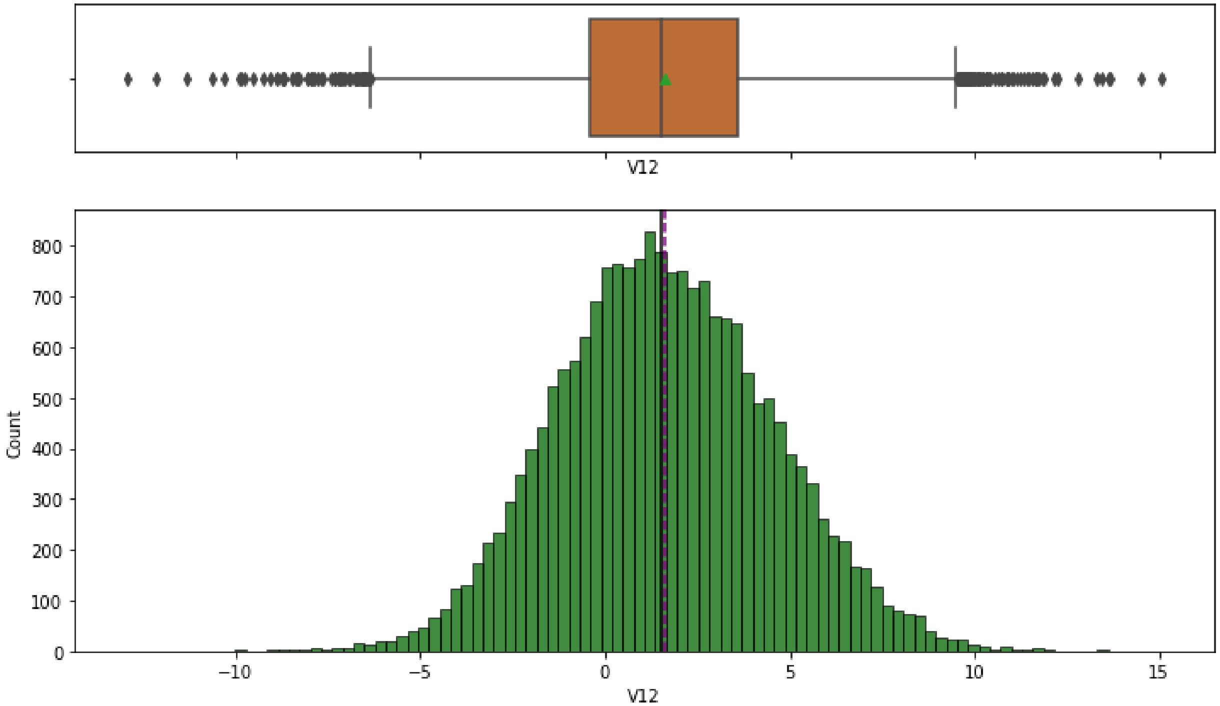

V7
Count
V7

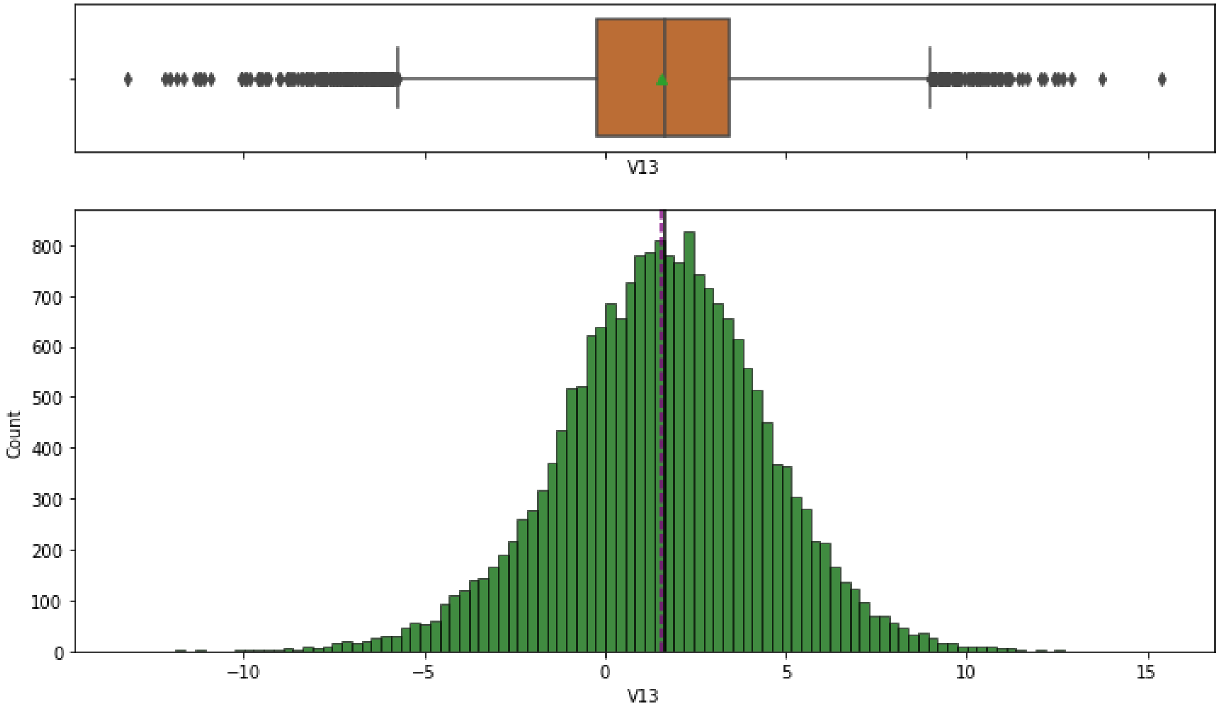

V8
Count
0
100
200
300
400
500
600
700
800
−15
−10
−5
0
5
10
V8

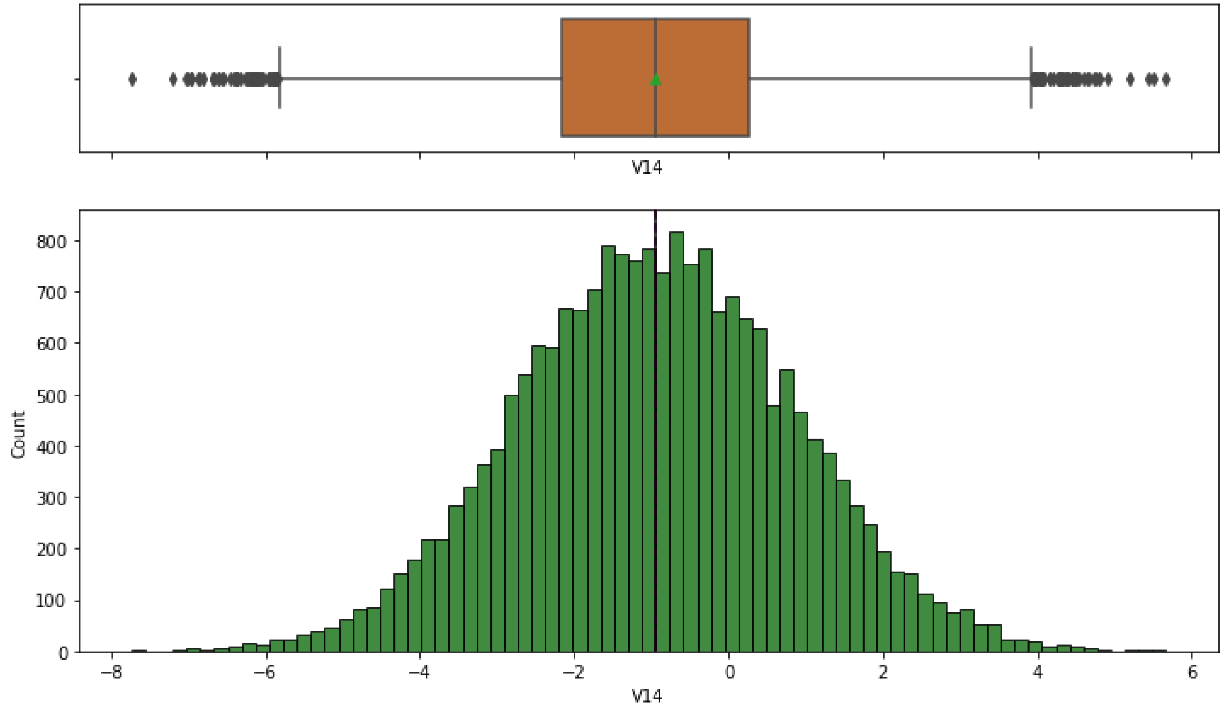

V9
Count
0
100
200
300
400
500
600
700
800
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V9

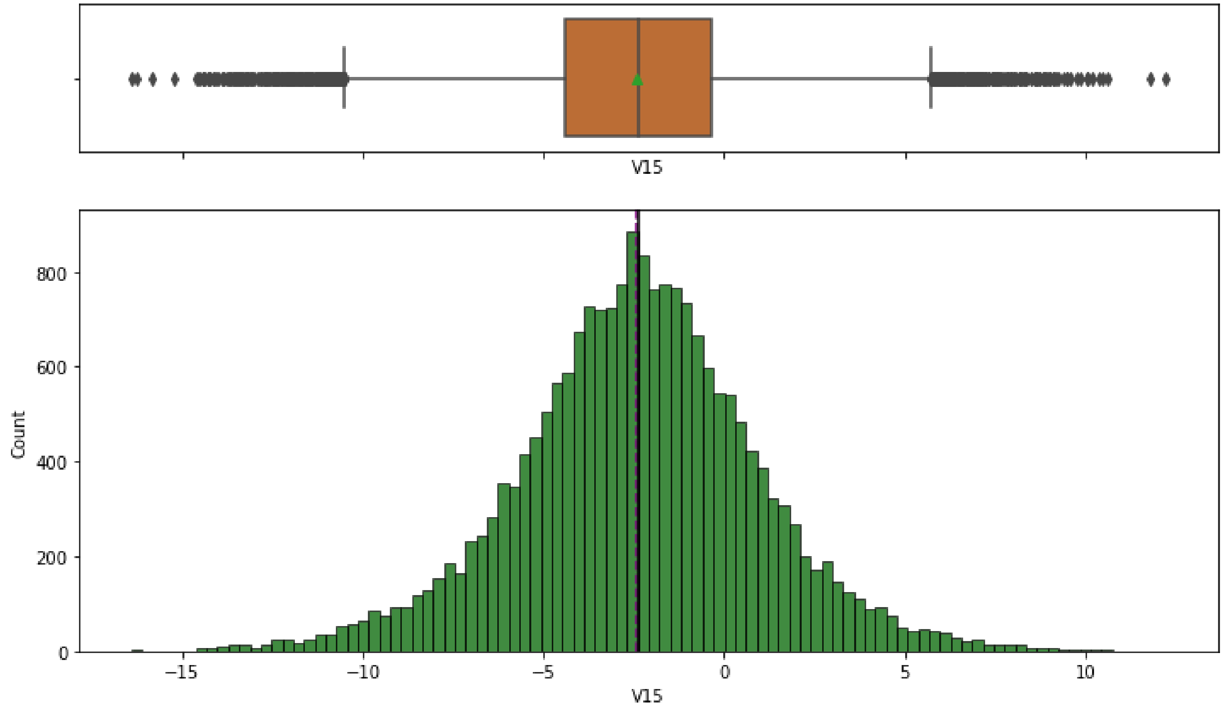

V10
800
700
600
500
400
300
200
100
0
Count
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V10

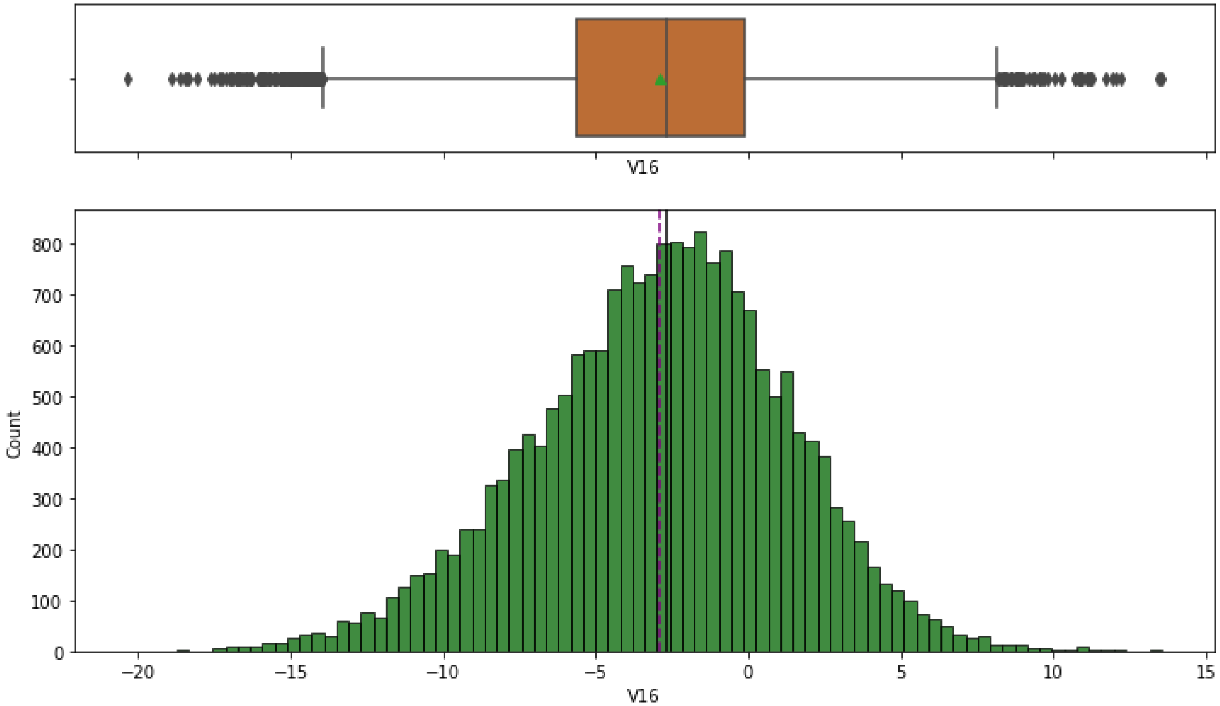

V11
800
700
600
500
400
300
200
100
0
Count
−15
−10
−5
0
5
10
V11

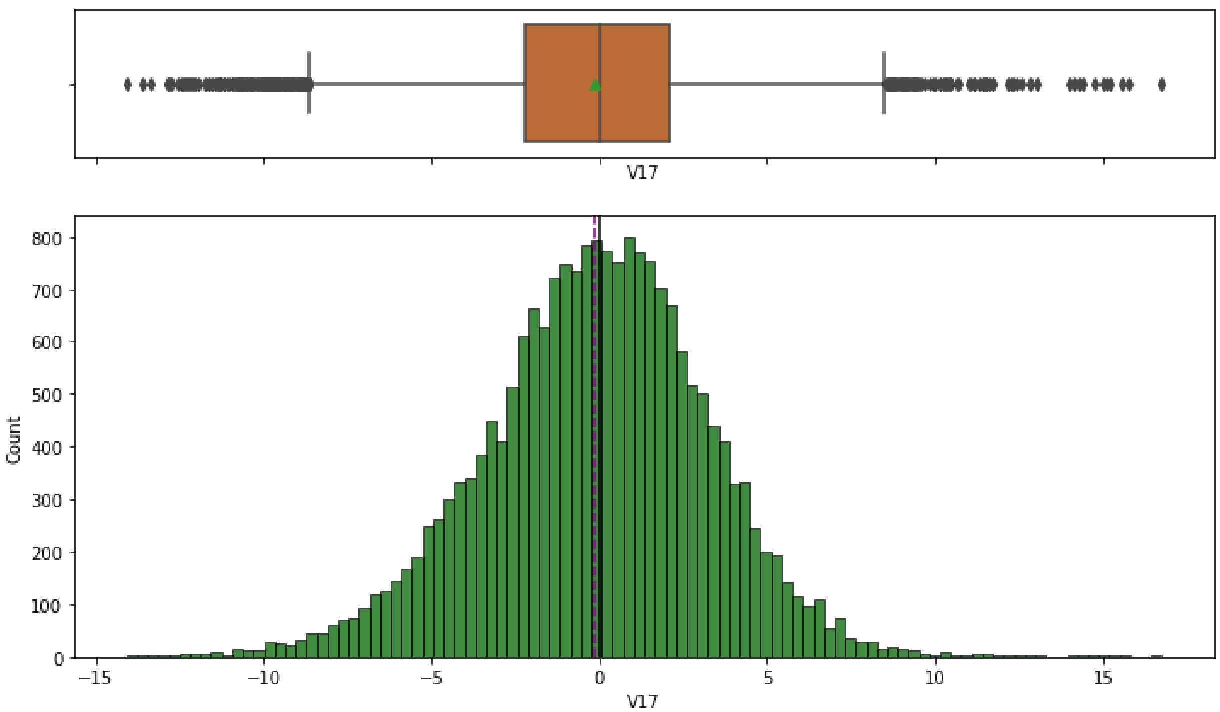

V12
800
700
600
500
400
300
200
100
0
Count
−10
−5
0
5
10
15
V12

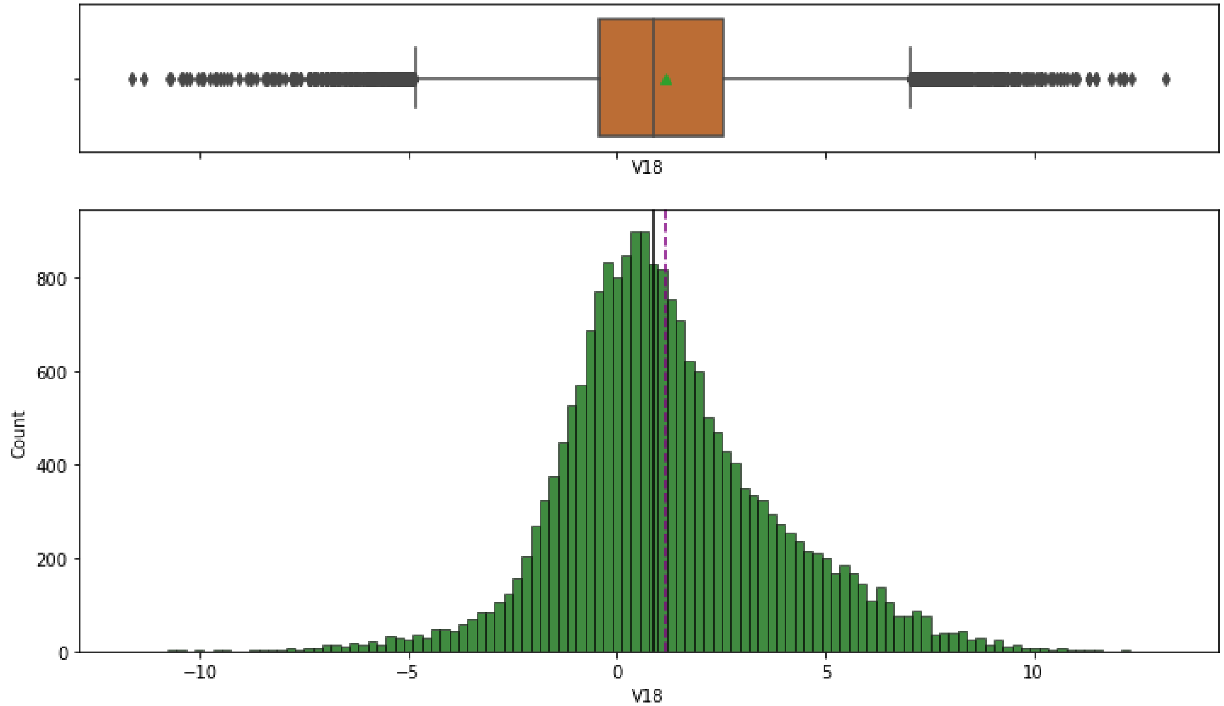

V13
800
700
600
500
400
300
200
100
0
Count
−10
−5
0
5
10
15
V13

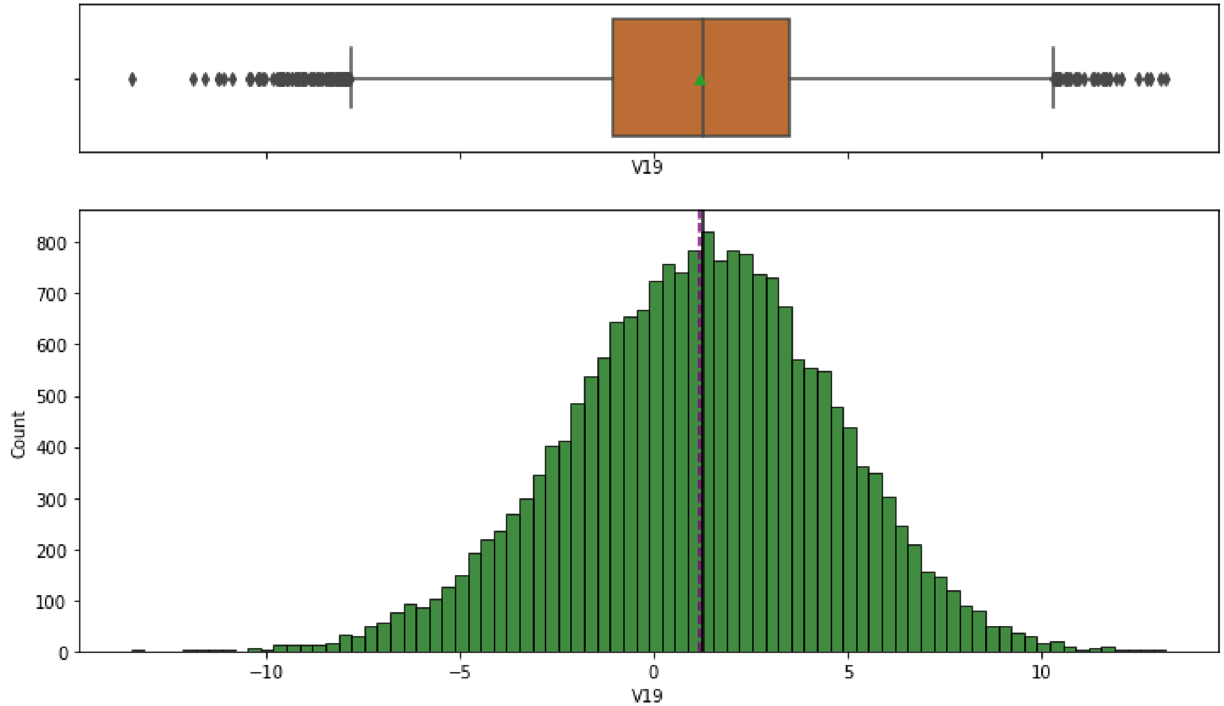

V14
Count
800
700
600
500
400
300
200
100
0
−8
−6
−4
−2
0
2
4
6
V14

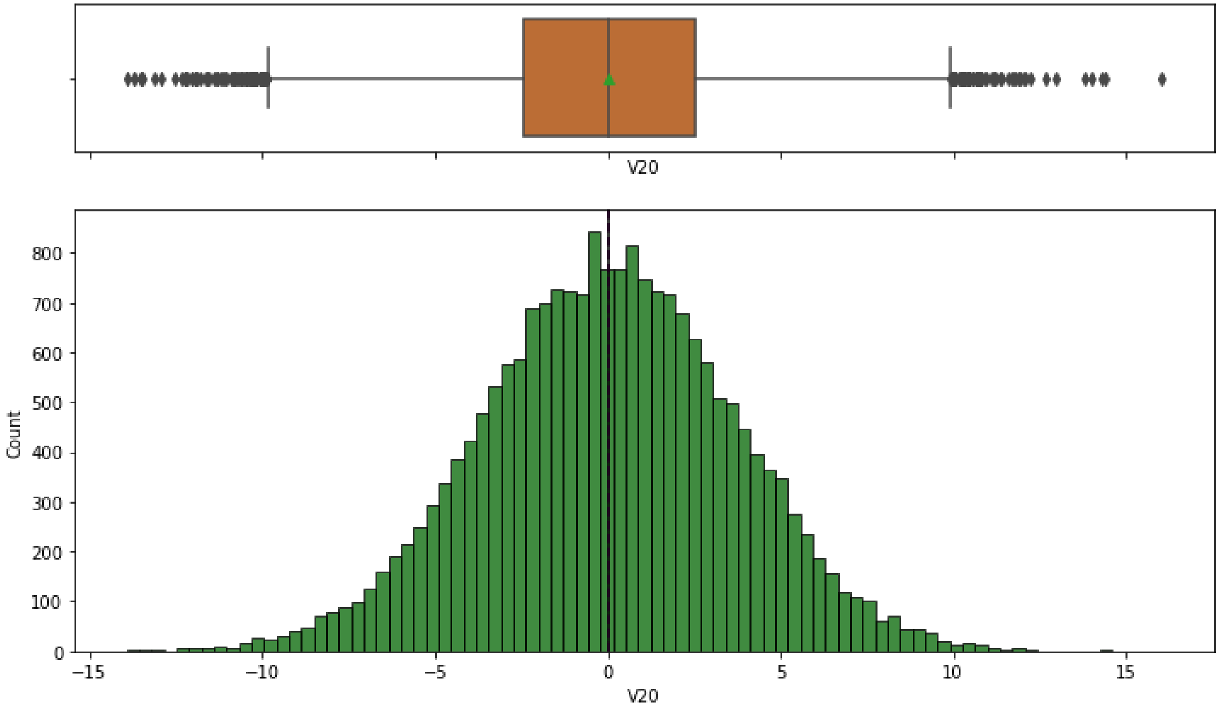

V15
Count
800
600
400
200
0
−15
−10
−5
0
5
10
V15

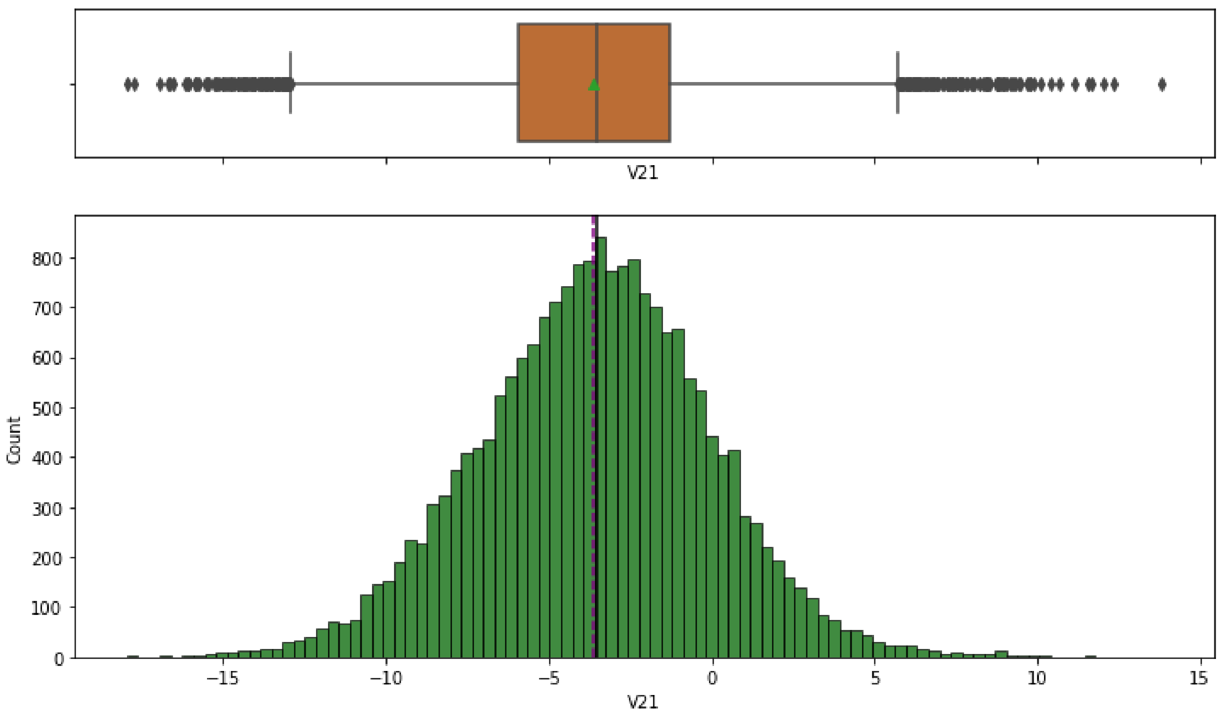

V16
Count
0
100
200
300
400
500
600
700
800
−20
−15
−10
−5
0
5
10
15
V16

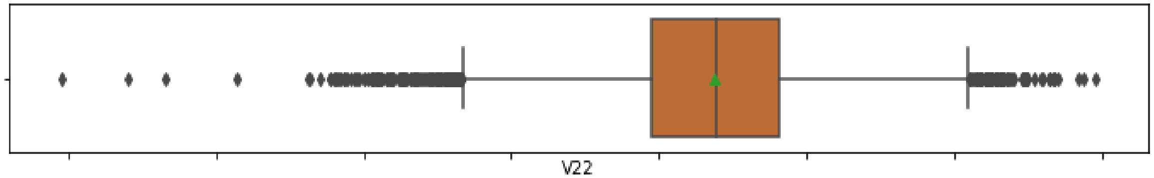

V17
Count
0
100
200
300
400
500
600
700
800
−15
−10
−5
0
5
10
15
V17

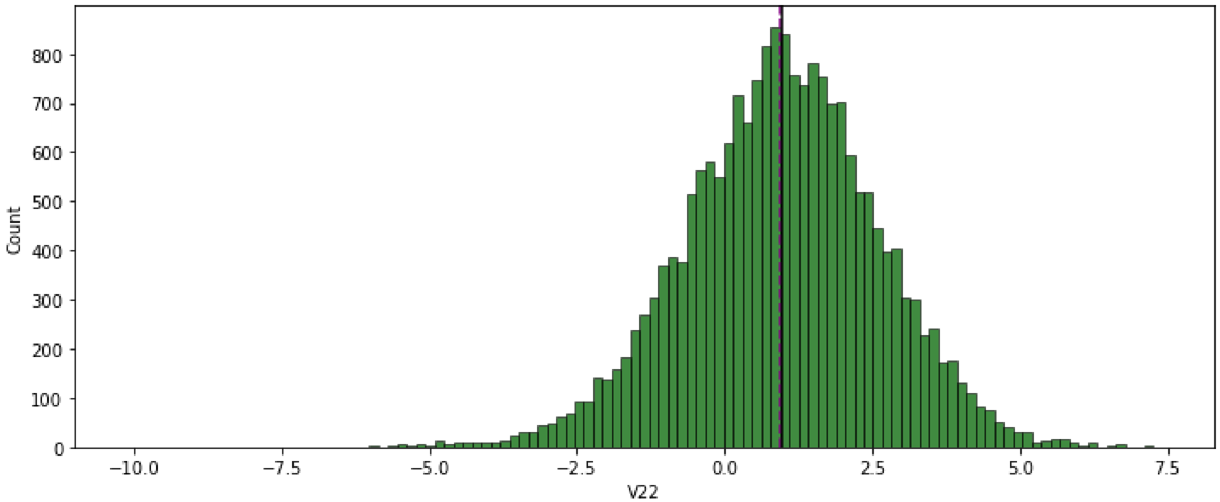

V18
Count
800
600
400
200
0
−10
−5
0
5
10
V18

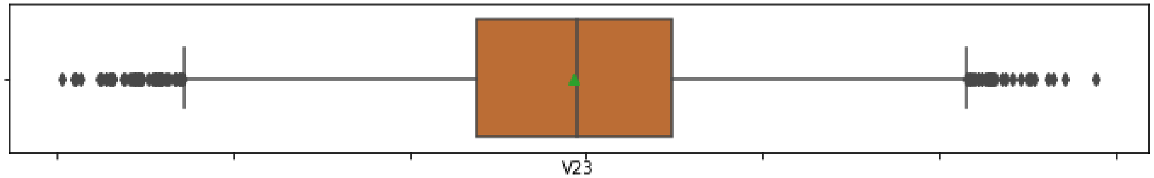

V19
Count
800
700
600
500
400
300
200
100
0
−10
−5
0
5
10
V19

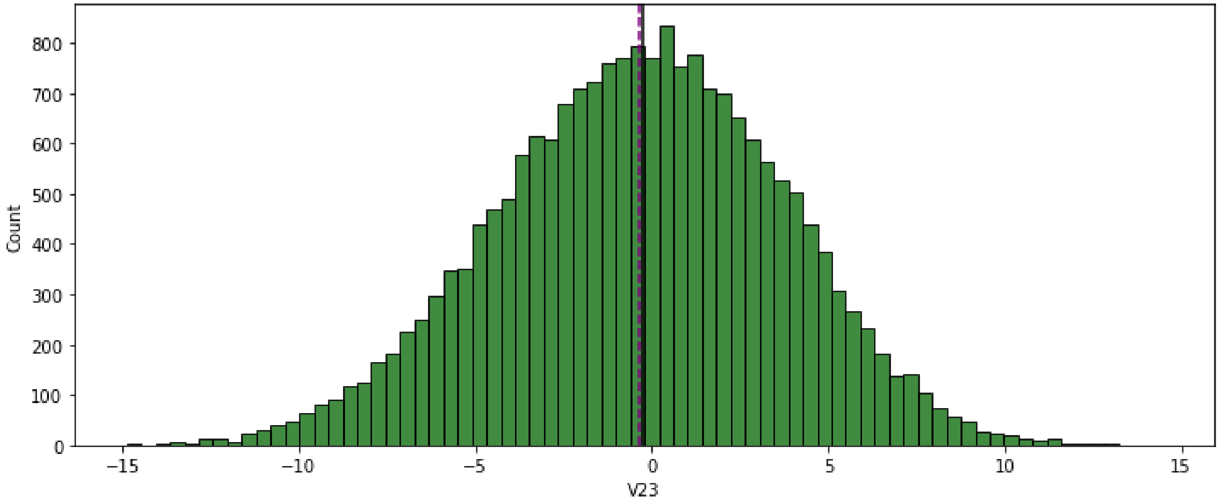

V20
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V20

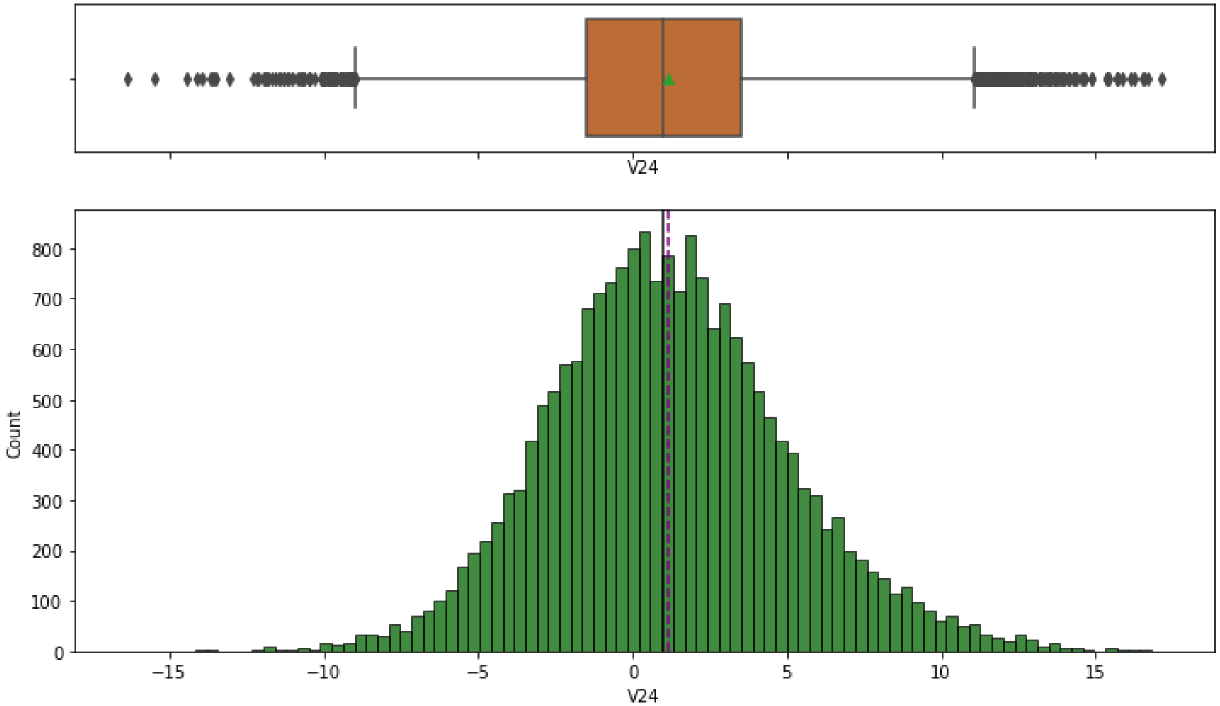

V21
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V21

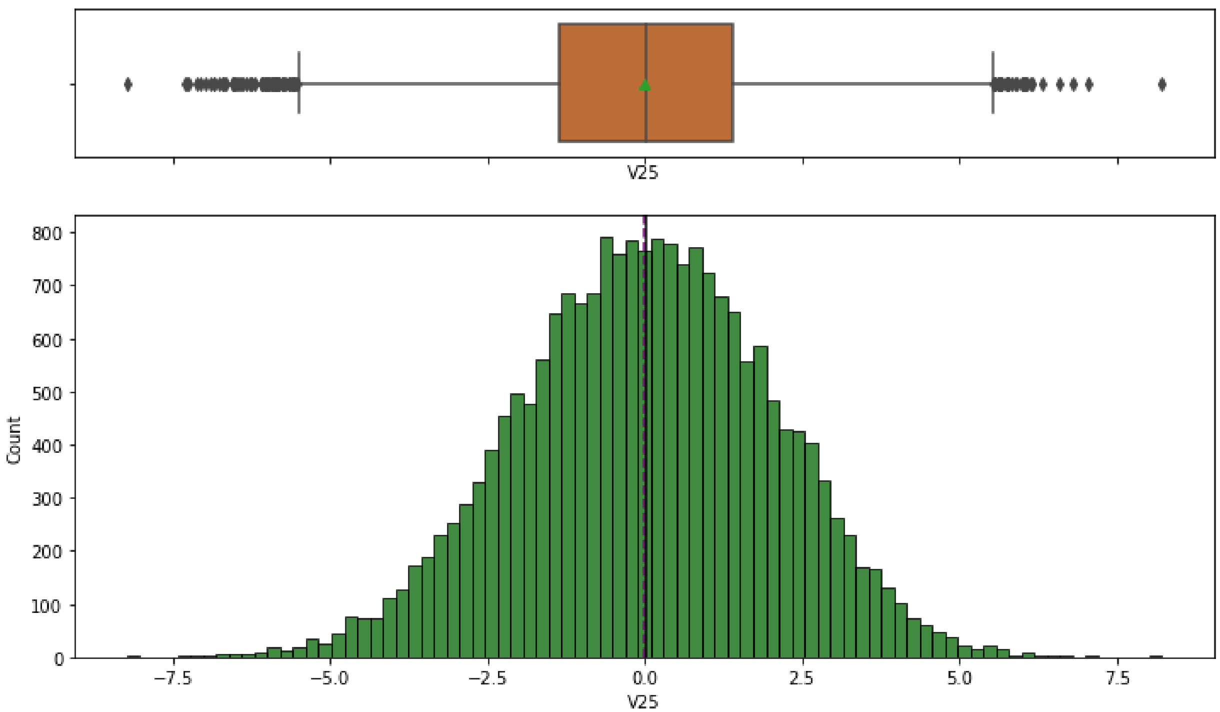

V22

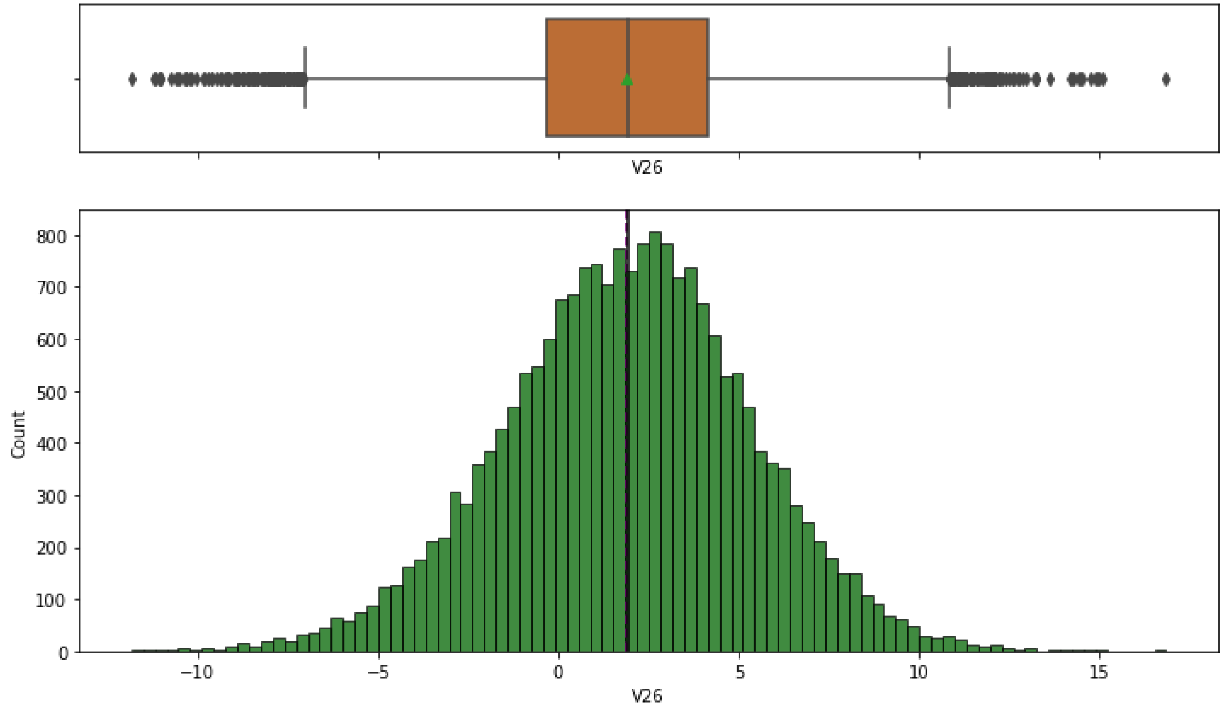

Count
800
700
600
500
400
300
200
100
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V22

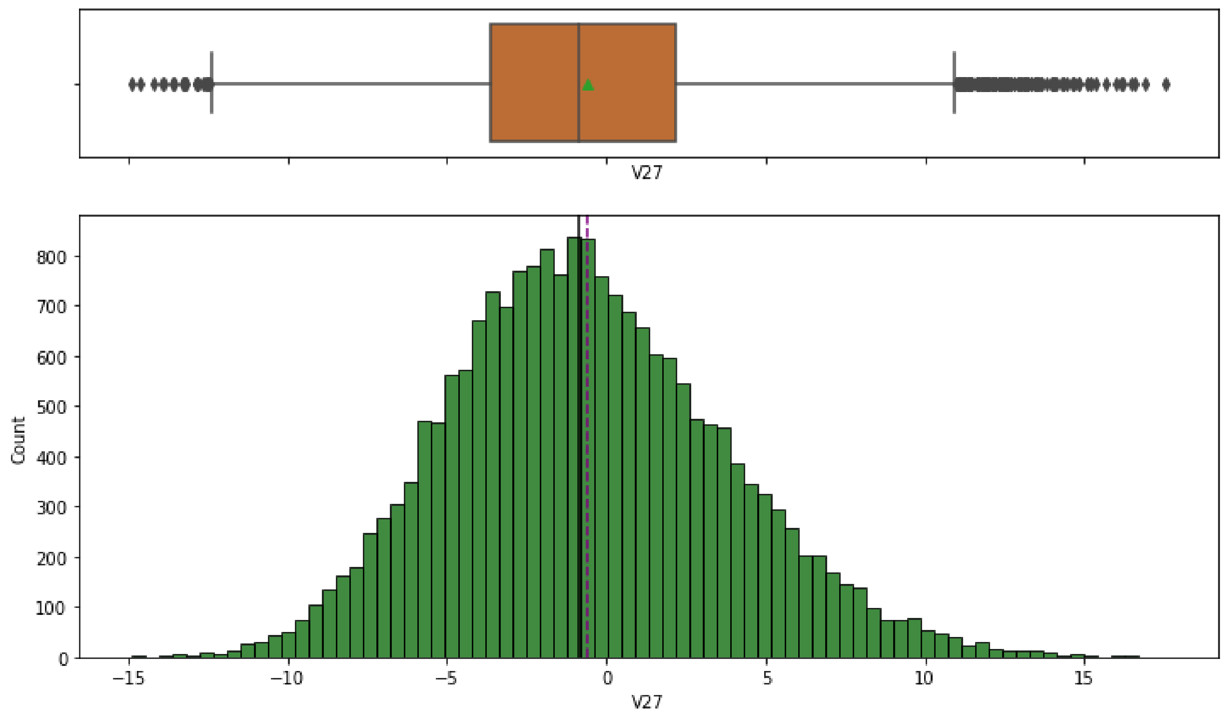

V23

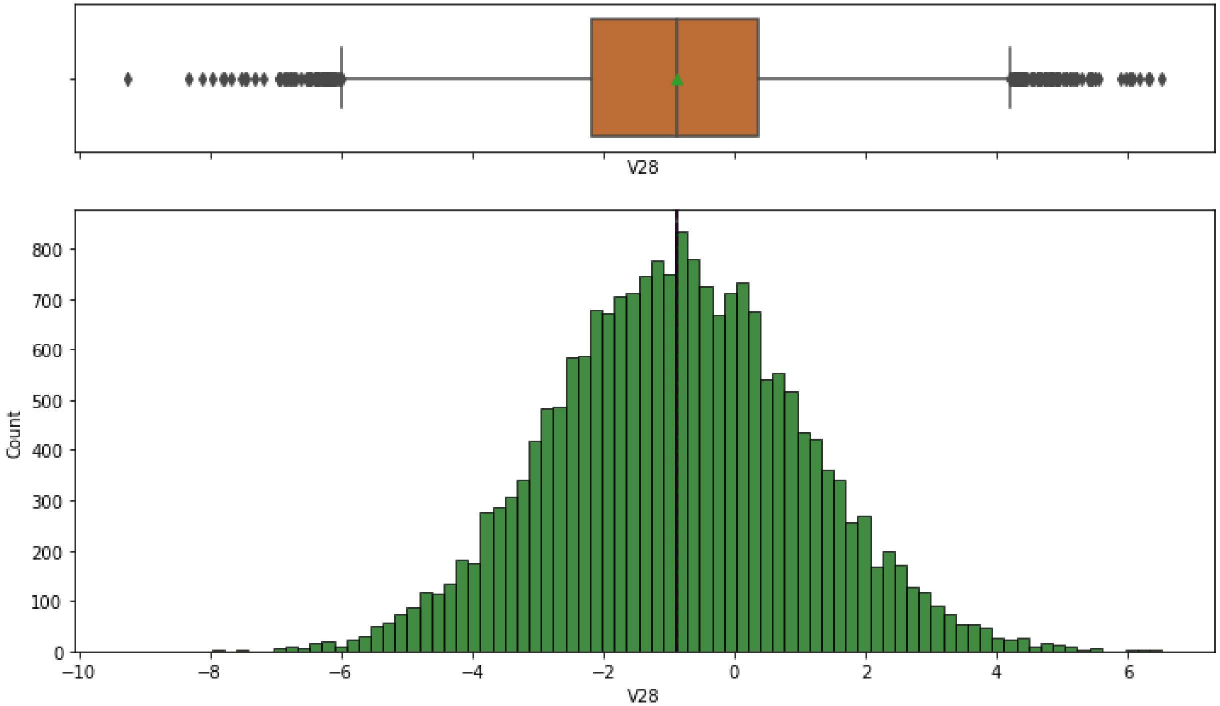

Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V23

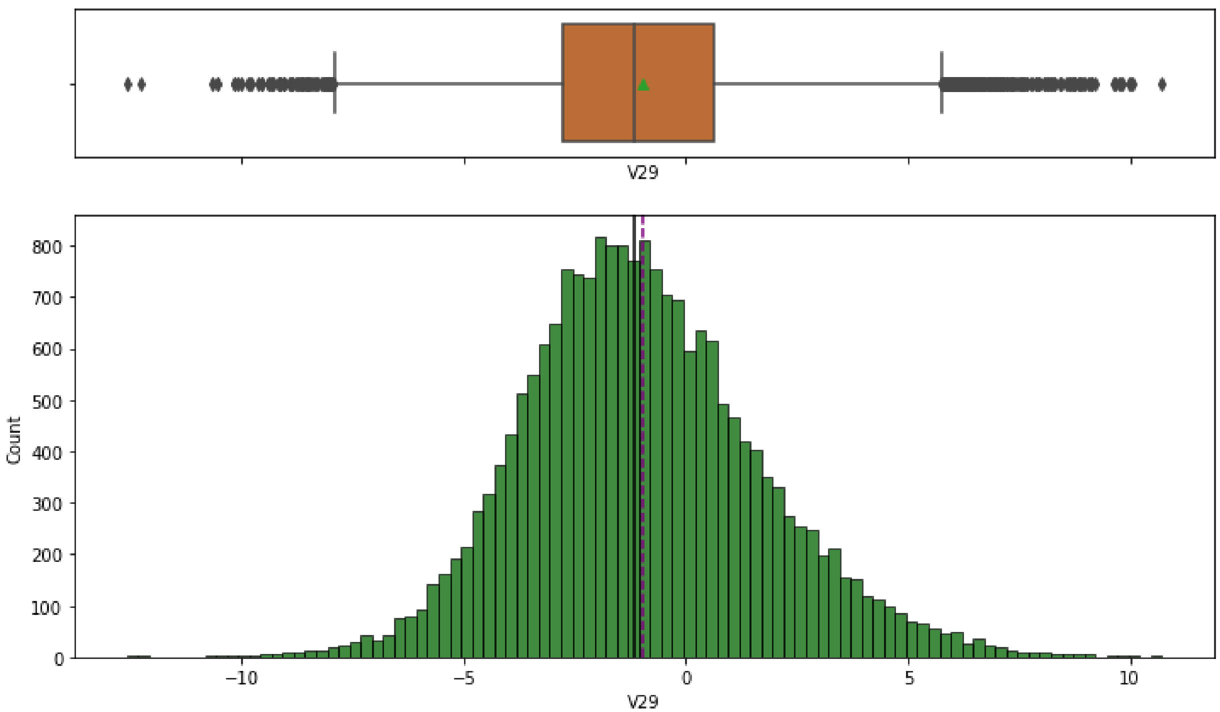

V24
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V24

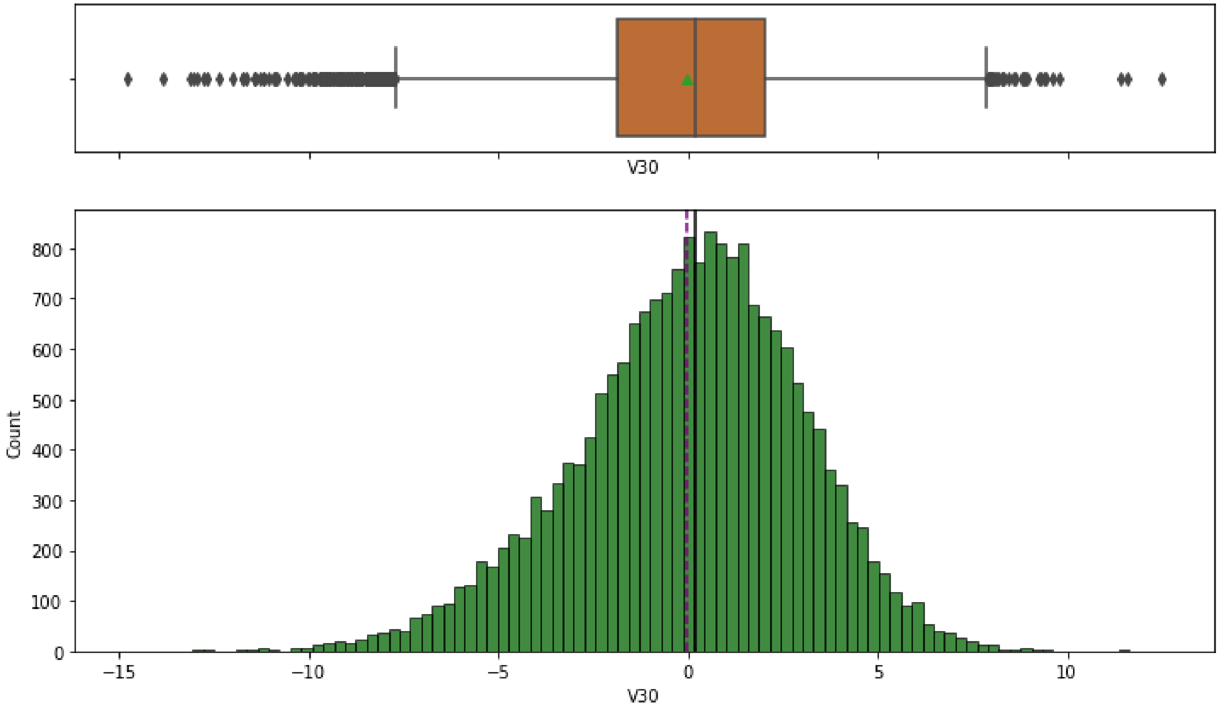

V25
Count
800
700
600
500
400
300
200
100
0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V25

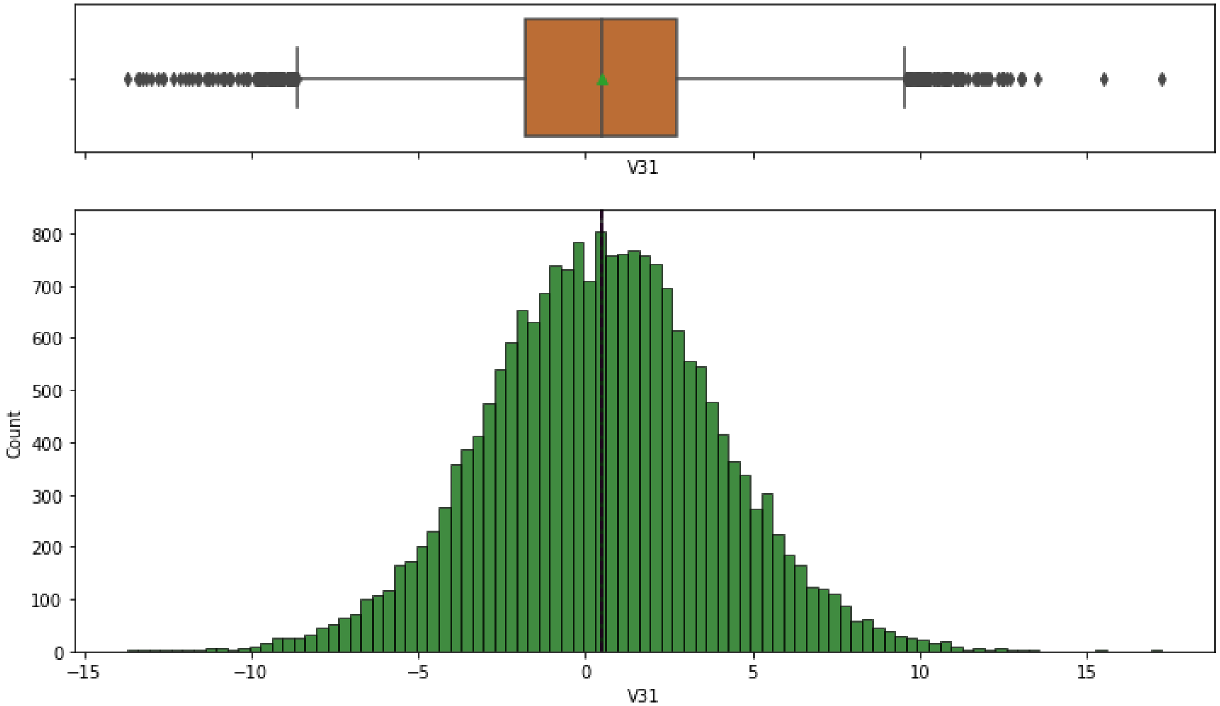

V26
Count
0
100
200
300
400
500
600
700
800
−10
−5
0
5
10
15
V26

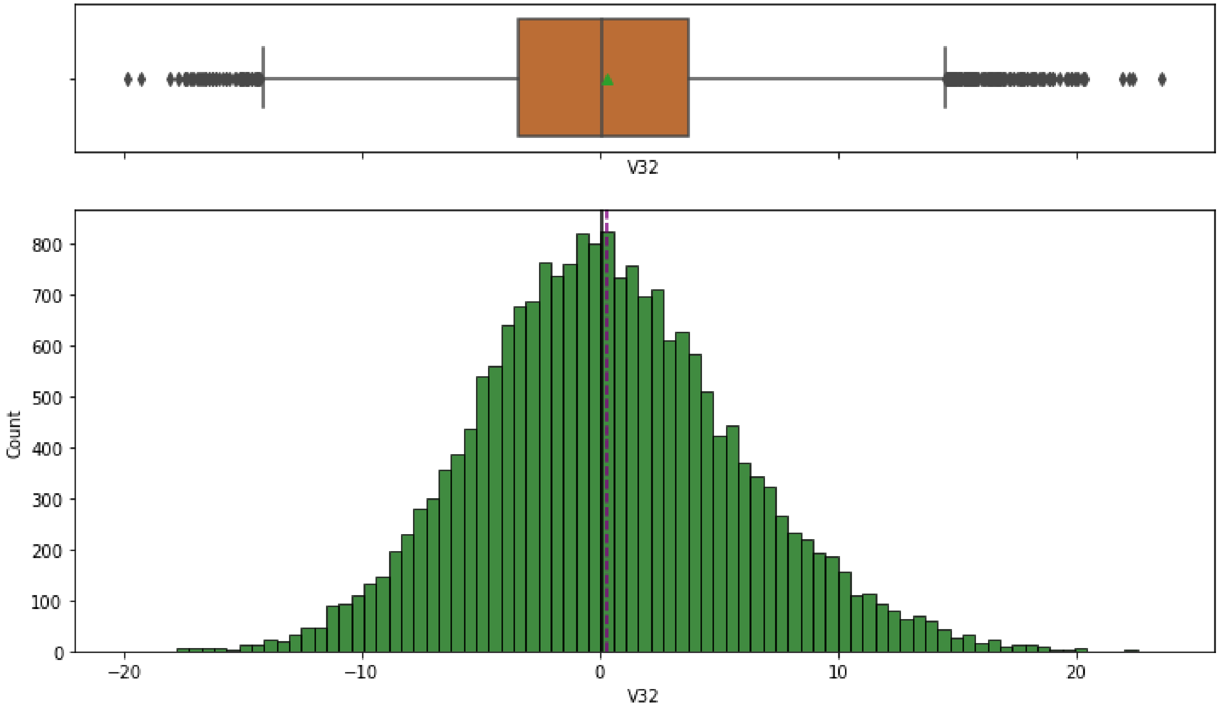

V27
Count
0
100
200
300
400
500
600
700
800
−15
−10
−5
0
5
10
15
V27

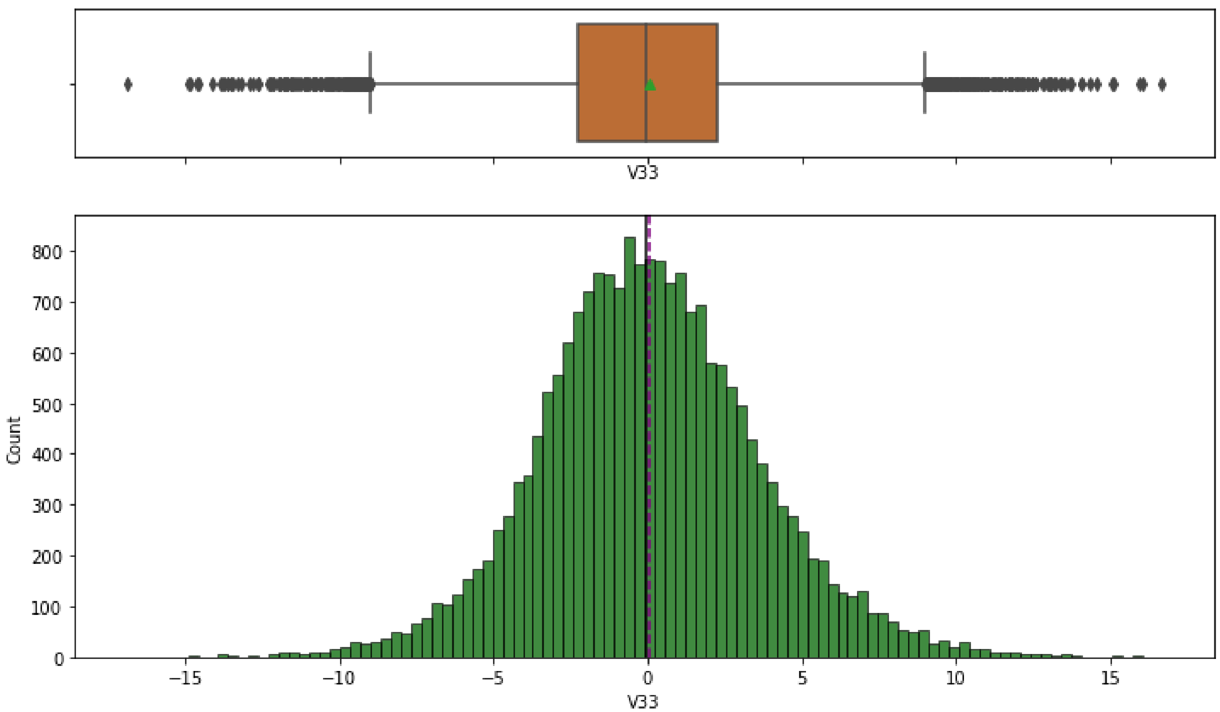

V28
Count
0
100
200
300
400
500
600
700
800
−10
−8
−6
−4
−2
0
2
4
6
V28

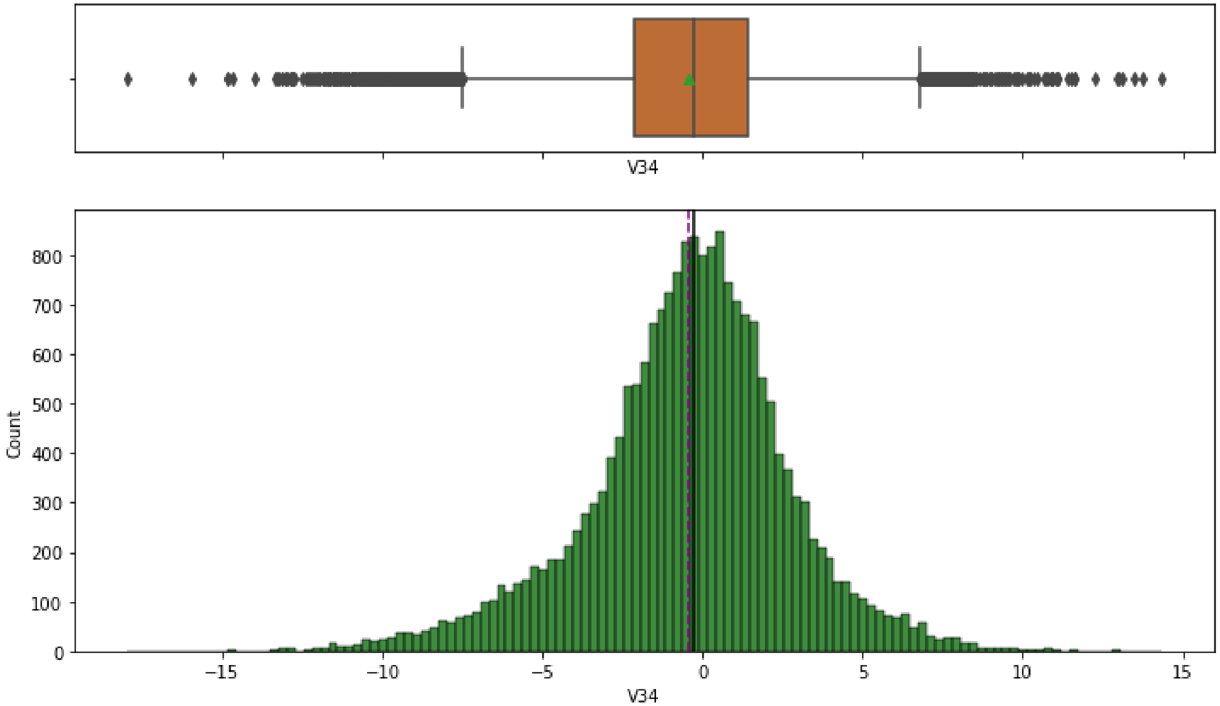

V29
Count
0
100
200
300
400
500
600
700
800
−10
−5
0
5
10
V29

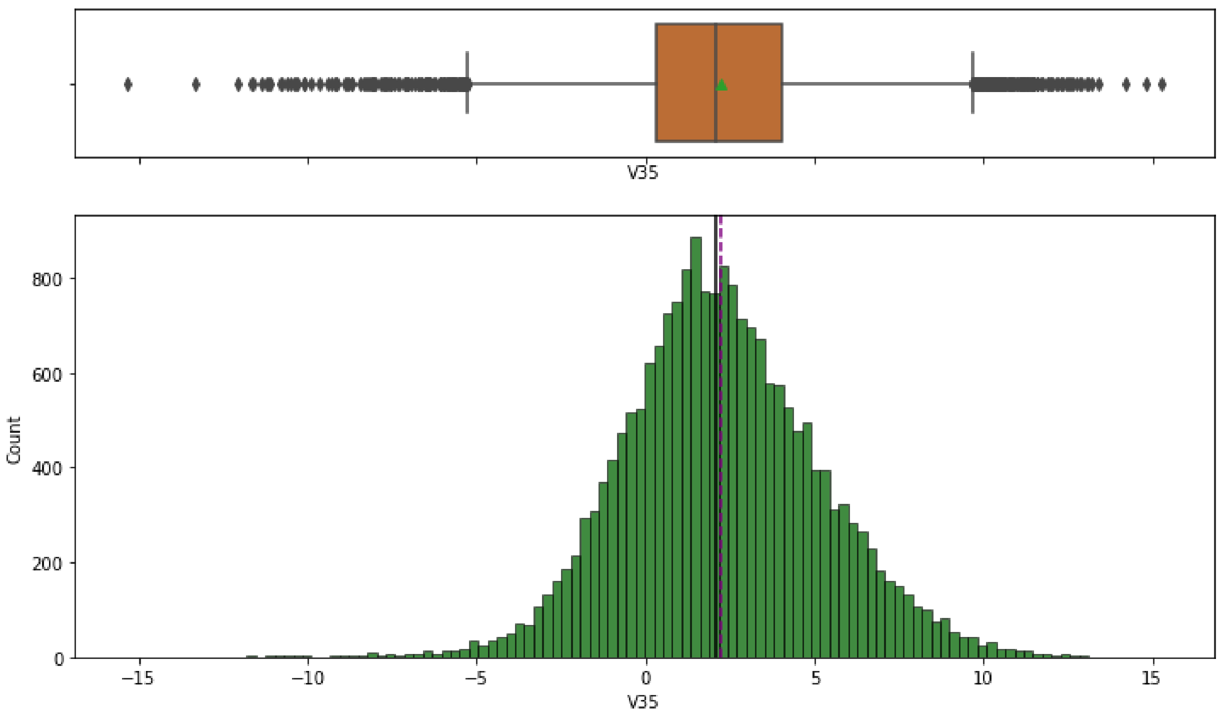

V30
800
700
600
500
400
300
200
100
0
Count
−15
−10
−5
0
5
10
V30

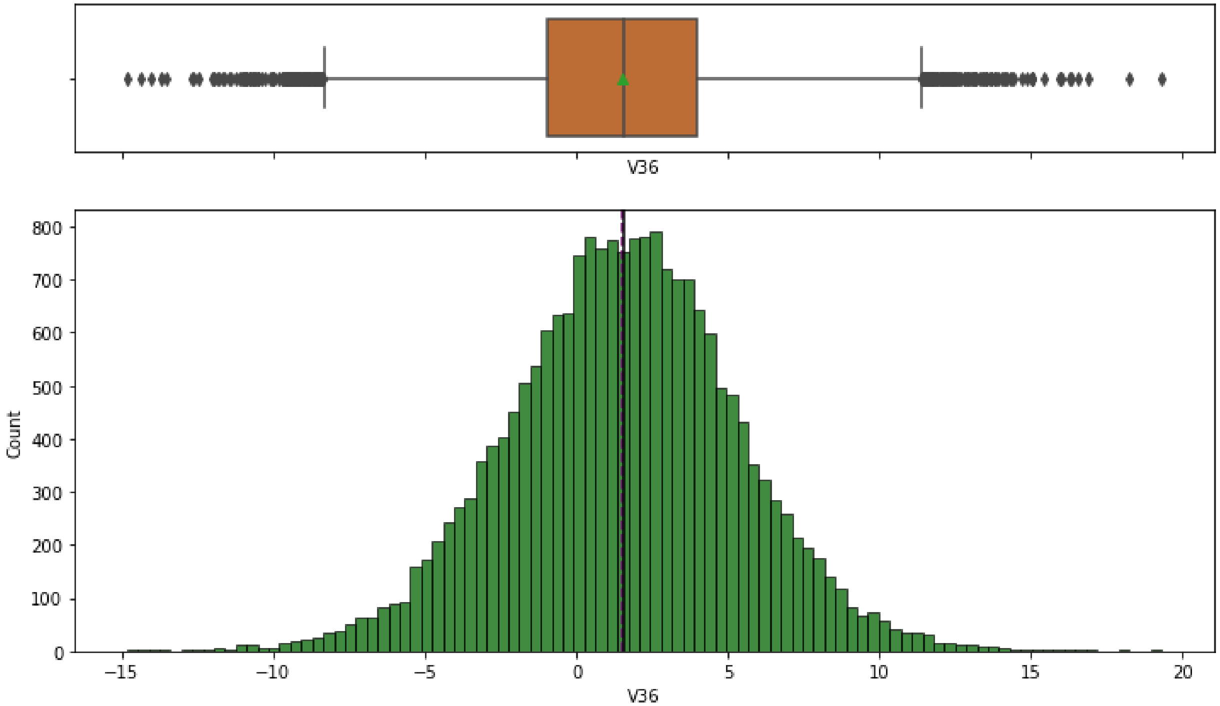

V31
800
700
600
500
400
300
200
100
0
Count
−15
−10
−5
0
5
10
15
V31

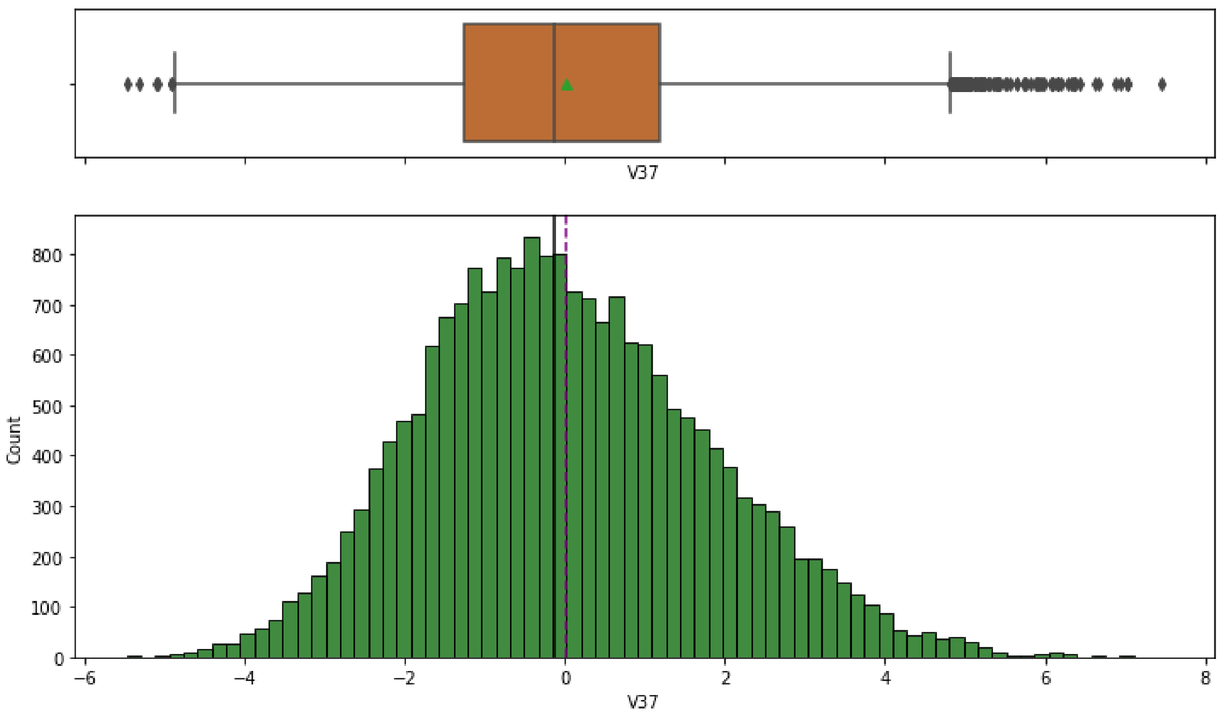

V32
Count
800
700
600
500
400
300
200
100
0
−20
−10
0
10
20
V32

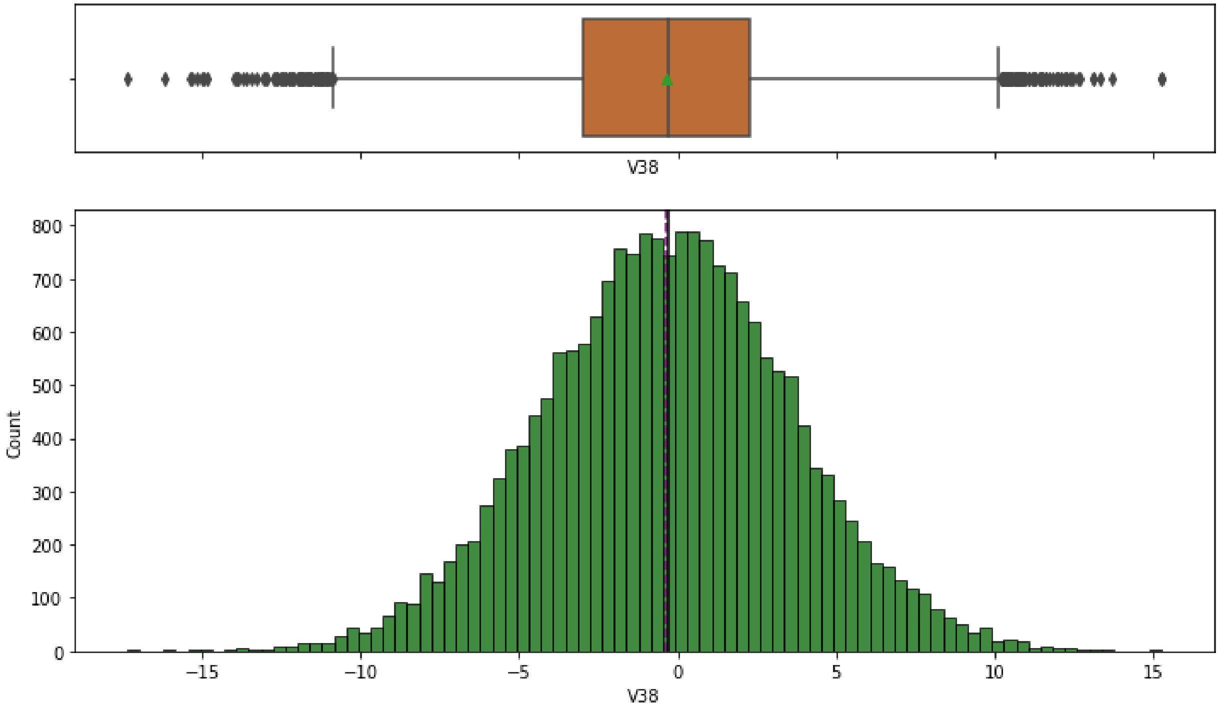

V33
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V33

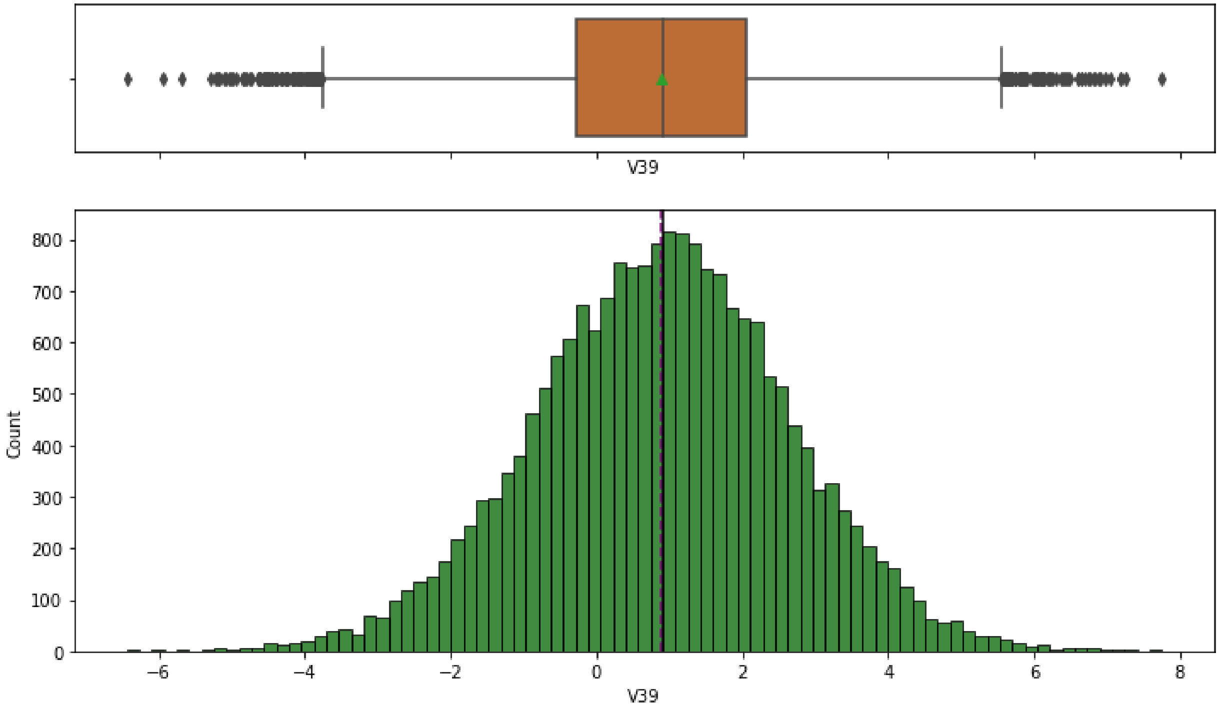

V34
Count
V34

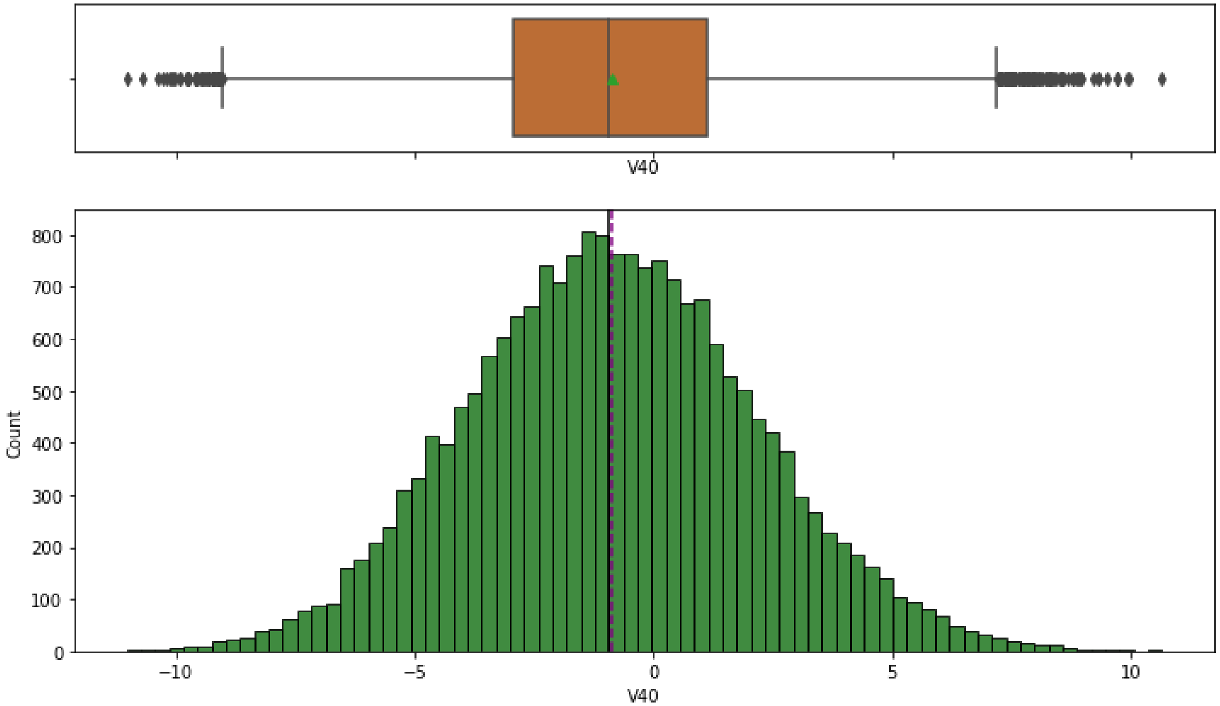

V35
Count
V35

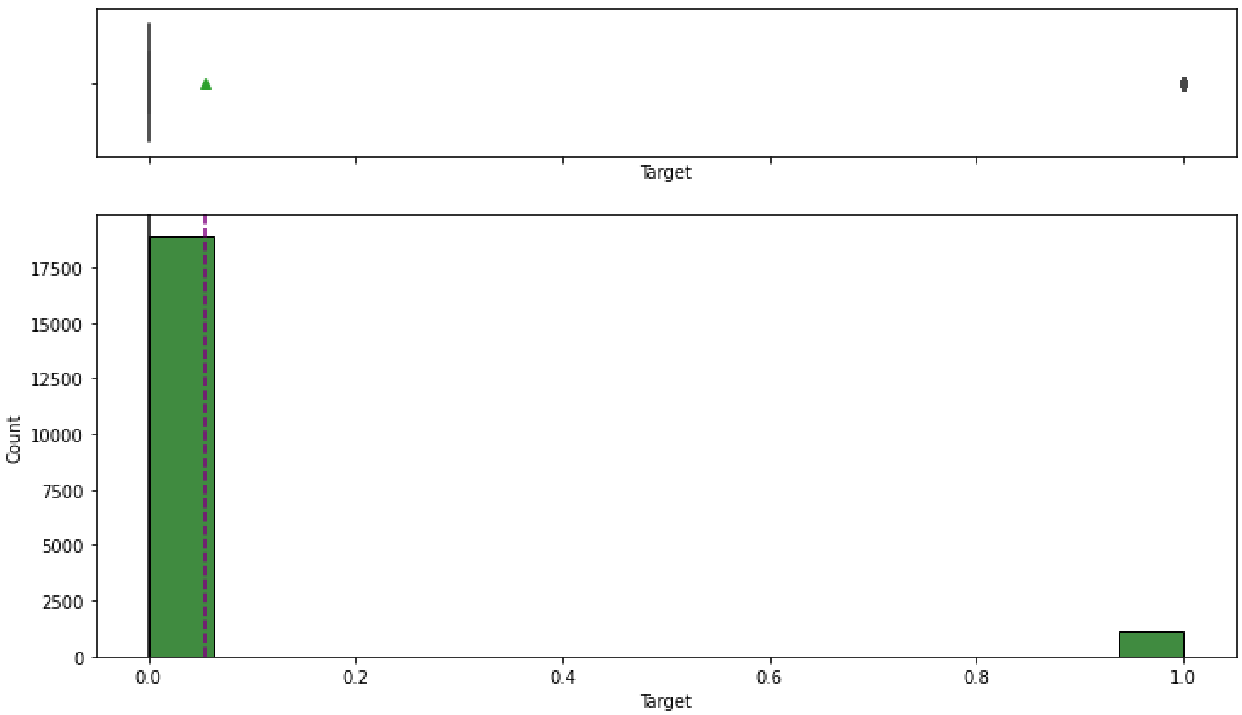

V36
800
700
600
500
400
300
200
100
0
Count
−15
−10
−5
0
5
10
15
20
V36


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


V38
800
700
600
500
400
300
200
100
0
Count
−15
−10
−5
0
5
10
15
V38


V39
800
700
600
500
400
300
200
100
0
Count
−6
−4
−2
0
2
4
6
8
V39


V40
800
700
600
500
400
300
200
100
0
Count
−10
−5
0
5
10
V40


Target
17500
15000
12500
10000
7500
5000
2500
0
Count
0.0
0.2
0.4
0.6
0.8
1.0
Target

Observations

- The 40 predictor variables have very similar distributions, with the ranges on average not exceeding 10.
- Many outliers on both the lower and upper ends of the predictor variable distributions.
- We will now analyze the target variable separately to identify the pattern: above shows multimodal.

In [16]:
```
df["Target"].value_counts()
```

Out[16]:
```
0    18890
1     1110
Name: Target, dtype: int64
```

In [17]:
```
test["Target"].value_counts()
```

Out[17]:
```
0    4718
1     282
Name: Target, dtype: int64
```

Observations

- “1” in the target variables should be considered as “failure” and “0” represents “No failure”.
- Test set has a slightly higher proportion of failures (56.4% vs 55.5%)

Data Pre-processing

In [18]:
```
X = df.drop(["Target"], axis=1)
y = df["Target"]

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

In [19]:
```
# splitting the data in 70:30 ratio for train to test data

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, random_state=1)

print(X_train.shape, X_val.shape, X_test.shape)
```

```
(14000, 40) (6000, 40) (5000, 40)
```

In [20]:
```
imputer = SimpleImputer(strategy="median")
```

In [21]:
```
# Fit and transform the train data
X_train = pd.DataFrame(imputer.fit_transform(X_train), columns=X_train.columns)
```

```
X_val = pd.DataFrame(imputer.fit_transform(X_val), columns=X_val.columns)
X_test = pd.DataFrame(imputer.fit_transform(X_test), columns=X_test.columns)
```

In [22]:

```
# check for missing values

print(X_train.isnull().sum())
print("-" * 30)
print(X_val.isnull().sum())
print("-" * 30)
print(X_test.isnull().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
------------------------------
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
```

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.

In [23]:

```
# 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
```

In [24]:

```
def confusion_matrix_sklearn(model, predictors, target):
    """
    To plot the confusion_matrix with percentages

    model: classifier
    predictors: independent variables
    target: dependent variable
    """
    y_pred = model.predict(predictors)
    cm = confusion_matrix(target, y_pred)
    labels = np.asarray(
        [
            ["{0:0.0f}".format(item) + "\n{0:.2%}".format(item / cm.flatten().sum())]
            for item in cm.flatten()
        ]
    ).reshape(2, 2)

    plt.figure(figsize=(6, 4))
    sns.heatmap(cm, annot=labels, fmt="")
    plt.ylabel("True label")
    plt.xlabel("Predicted label")
```

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.

In [25]:
```
# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.recall_score)
```

Model Building on original data

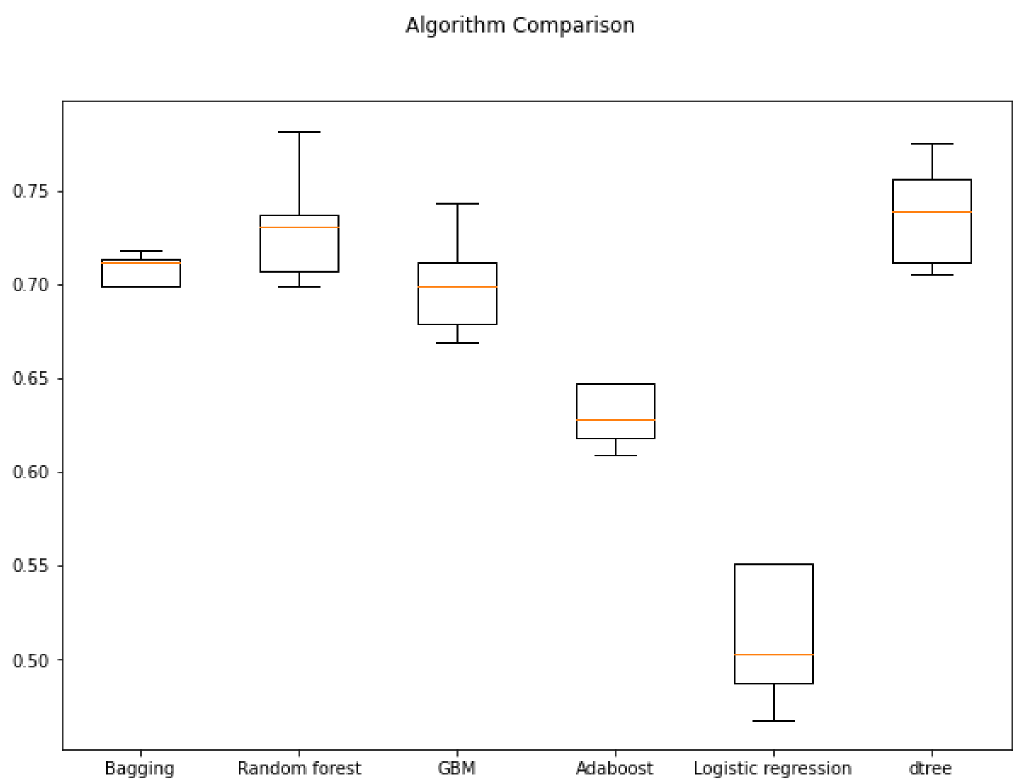
In [26]:
```
models = []  # Empty list to store all the models

# Appending models into the list
models.append(("Bagging", BaggingClassifier(random_state=1)))
models.append(("Random forest", RandomForestClassifier(random_state=1)))
models.append(("GBM", GradientBoostingClassifier(random_state=1)))
models.append(("Adaboost", AdaBoostClassifier(random_state=1)))
models.append(("Logistic regression", LogisticRegression(random_state=1)))
models.append(("dtree", DecisionTreeClassifier(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:" "\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:

Bagging: 0.7080597746202841
Random forest: 0.7311448636289402
GBM: 0.7004246284501063
Adaboost: 0.6299771353911481
Logistic regression: 0.5121754042136208
```

```
dtree: 0.7375142903805324

Validation Performance:

Bagging: 0.7051671732522796
Random forest: 0.7082066869300911
GBM: 0.6838905775075987
Adaboost: 0.5805471124620061
Logistic regression: 0.44680851063829785
dtree: 0.7173252279635258
```

In [27]:

```
# Plotting boxplots for CV scores of 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 on oversampled data

In [28]:
```
# 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)

print("Before OverSampling, counts of label '1': {}".format(sum(y_train == 1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(y_train == 0)))

print("After OverSampling, counts of label '1': {}".format(sum(y_train_over == 1)))
print("After OverSampling, counts 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, counts of label '1': 781
Before OverSampling, counts of label '0': 13219

After OverSampling, counts of label '1': 13219
After OverSampling, counts of label '0': 13219

After OverSampling, the shape of train_X: (26438, 40)
After OverSampling, the shape of train_y: (26438,)
```

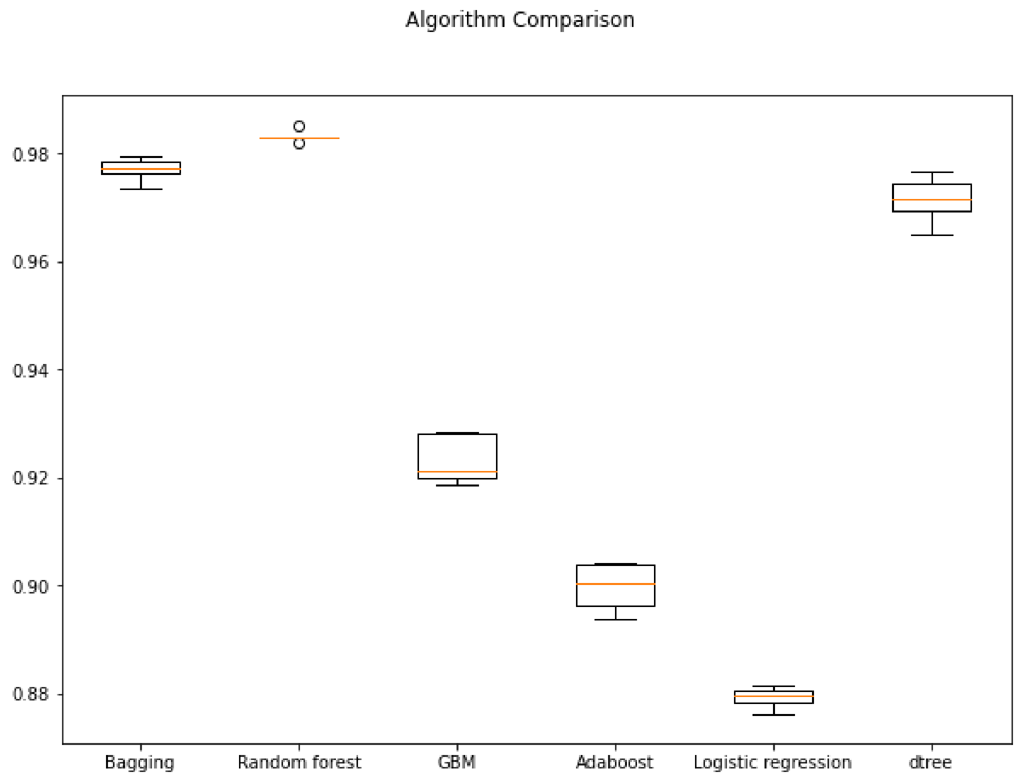
In [29]:
```
models_over = []  # Empty list to store all the models

# Appending models into the list
models_over.append(("Bagging", BaggingClassifier(random_state=1)))
models_over.append(("Random forest", RandomForestClassifier(random_state=1)))
models_over.append(("GBM", GradientBoostingClassifier(random_state=1)))
models_over.append(("Adaboost", AdaBoostClassifier(random_state=1)))
models_over.append(("Logistic regression", LogisticRegression(random_state=1)))
models_over.append(("dtree", DecisionTreeClassifier(random_state=1)))

results2 = []  # 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:" "\n")

for name, model in models_over:
    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
    )
    results2.append(cv_result)
    names.append(name)
    print("{}: {}".format(name, cv_result.mean()))

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

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

```
Cross-Validation Performance:

Bagging: 0.9770028213709836
Random forest: 0.9832058879591166
GBM: 0.9232918799580773
Adaboost: 0.8997650288519384
Logistic regression: 0.8792647263373178
dtree: 0.971329541740435

Validation Performance:

Bagging: 0.8115501519756839
Random forest: 0.8389057750759878
GBM: 0.8844984802431611
Adaboost: 0.8541033434650456
Logistic regression: 0.8358662613981763
dtree: 0.78419452887538
```

In [30]:

```
# Plotting boxplots for CV scores of all models defined above (oversampled)
fig = plt.figure(figsize=(10, 7))

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

plt.boxplot(results2)
ax.set_xticklabels(names)

plt.show()
```



Model Building on undersampled data

In [31]:
```
rus = RandomUnderSampler(random_state=1, sampling_strategy=1)
X_train_un, y_train_un = rus.fit_resample(X_train, y_train)


print("Before UnderSampling, counts of label '1': {}".format(sum(y_train == 1)))
print("Before UnderSampling, counts of label '0': {} \n".format(sum(y_train == 0)))

print("After UnderSampling, counts of label '1': {}".format(sum(y_train_un == 1)))
print("After UnderSampling, counts of label '0': {} \n".format(sum(y_train_un == 0)))

print("After UnderSampling, the shape of train_X: {}".format(X_train_un.shape))
print("After UnderSampling, the shape of train_y: {} \n".format(y_train_un.shape))
```

```
Before UnderSampling, counts of label '1': 781
Before UnderSampling, counts of label '0': 13219

After UnderSampling, counts of label '1': 781
After UnderSampling, counts of label '0': 781

After UnderSampling, the shape of train_X: (1562, 40)
```

```
After UnderSampling, the shape of train_y: (1562,)
```

In [32]:

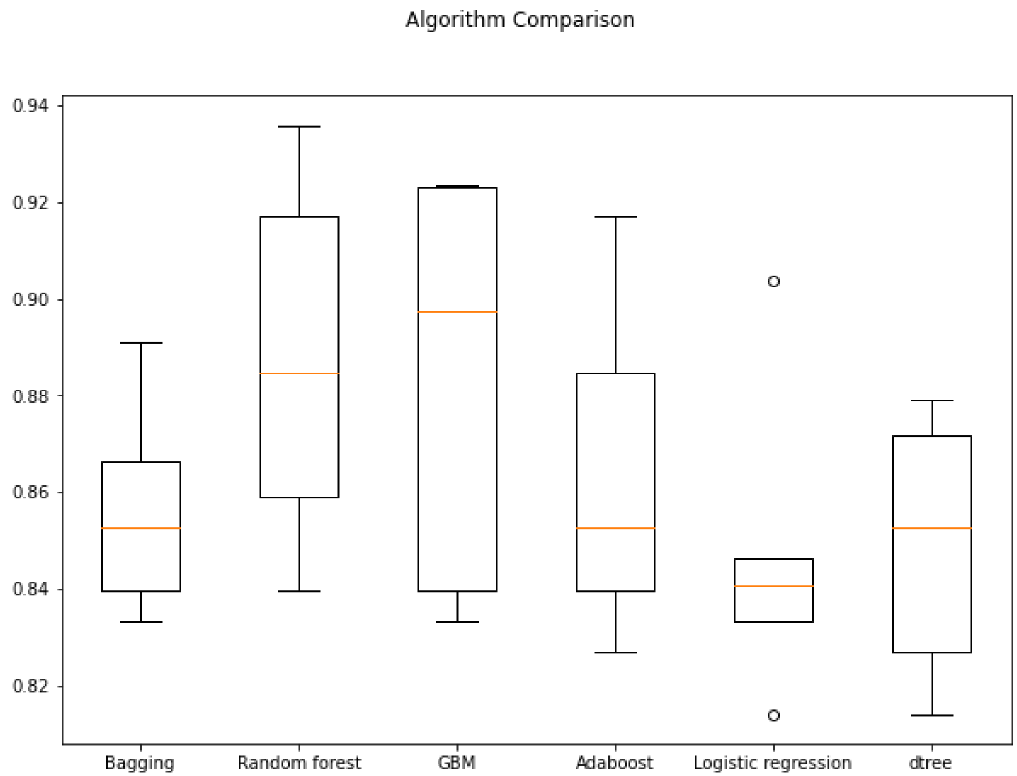
```python
models_un = []  # Empty list to store all the models

# Appending models into the list
models_un.append(("Bagging", BaggingClassifier(random_state=1)))
models_un.append(("Random forest", RandomForestClassifier(random_state=1)))
models_un.append(("GBM", GradientBoostingClassifier(random_state=1)))
models_un.append(("Adaboost", AdaBoostClassifier(random_state=1)))
models_un.append(("Logistic regression", LogisticRegression(random_state=1)))
models_un.append(("dtree", DecisionTreeClassifier(random_state=1)))

results3 = []  # 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:" "\n")

for name, model in models_un:
    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
    )
    results3.append(cv_result)
    names.append(name)
    print("{}: {}".format(name, cv_result.mean()))

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

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

```
Cross-Validation Performance:

Bagging: 0.8565817409766454
Random forest: 0.8872856442920136
GBM: 0.8834313245141271
Adaboost: 0.8642087212150906
Logistic regression: 0.8476400457292177
dtree: 0.8488731014208721

Validation Performance:

Bagging: 0.8480243161094225
Random forest: 0.8814589665653495
GBM: 0.8905775075987842
Adaboost: 0.8541033434650456
```

```
Logistic regression: 0.8358662613981763
dtree: 0.8389057750759878
```

In [116…

```
# Plotting boxplots for CV scores of all models defined above (undersampled)
fig = plt.figure(figsize=(10, 7))

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

plt.boxplot(results3)
ax.set_xticklabels(names)

plt.show()
```



After looking at performance of all the models, let's decide which models can further improve with hyperparameter tuning.

Hyperparameter Tuning

Tuning Bagging: Original

In [57]:

```
# defining model
```

```
Model = BaggingClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    'max_samples': [0.8,0.9,1],
    'max_features': [0.7,0.8,0.9],
    'n_estimators' : [30,50,70], }

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train,y_train)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 50, 'max_samples': 0.9, 'max_features': 0.8} with CV score=0.7323452555936634:
```

Tuning Bagging: Oversampled

In [58]:

```
# defining model
Model = BaggingClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    'max_samples': [0.8,0.9,1],
    'max_features': [0.7,0.8,0.9],
    'n_estimators' : [30,50,70], }

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over,y_train_over)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 70, 'max_samples': 0.8, 'max_features': 0.8} with CV score=0.983583873824214:
```

Tuning Bagging: Undersampled

In [59]:

```
# defining model
Model = BaggingClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    'max_samples': [0.8,0.9,1],
    'max_features': [0.7,0.8,0.9],
    'n_estimators' : [30,50,70], }

#Calling RandomizedSearchCV
```

```
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_un,y_train_un)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 70, 'max_samples': 0.9, 'max_features': 0.7} with CV score=0.8885758615057977:
```

Tuning Decision Tree: Original

In [41]:
```
# defining model
Model = DecisionTreeClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {'max_depth': np.arange(2,6),
              'min_samples_leaf': [1, 4, 7],
              'max_leaf_nodes' : [10,15],
              'min_impurity_decrease': [0.0001,0.001] }

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=10, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train,y_train)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'min_samples_leaf': 7, 'min_impurity_decrease': 0.0001, 'max_leaf_nodes': 15, 'max_depth': 5} with CV score=0.49
56230605912134:
```

Tuning Decision Tree: Oversampled

In [47]:
```
# defining model
Model = DecisionTreeClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {'max_depth': np.arange(2,6),
              'min_samples_leaf': [1, 4, 7],
              'max_leaf_nodes' : [10,15],
              'min_impurity_decrease': [0.0001,0.001] }

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=10, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over,y_train_over)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'min_samples_leaf': 7, 'min_impurity_decrease': 0.001, 'max_leaf_nodes': 15, 'max_depth': 3} with CV score=0.874
```

```
0479375486185:
```

Tuning Decision Tree: Undersampled

In [43]:

```
# defining model
Model = DecisionTreeClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {'max_depth': np.arange(2,20),
              'min_samples_leaf': [1, 2, 5, 7],
              'max_leaf_nodes' : [5, 10,15],
              'min_impurity_decrease': [0.0001,0.001] }

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=10, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_un,y_train_un)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'min_samples_leaf': 7, 'min_impurity_decrease': 0.001, 'max_leaf_nodes': 15, 'max_depth': 6} with CV score=0.809
186673199412:
```

Tuning Random Forest: Original

In [44]:

```
# defining model
Model = RandomForestClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    "n_estimators": [200,250,300],
    "min_samples_leaf": np.arange(1, 4),
    "max_features": [np.arange(0.3, 0.6, 0.1),'sqrt'],
    "max_samples": np.arange(0.4, 0.7, 0.1)}


#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train, y_train)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 200, 'min_samples_leaf': 2, 'max_samples': 0.6, 'max_features': 'sqrt'} with CV score=0.69394087
86542545:
```

Tuning Random Forest: Oversampled

In [45]:

```
# defining model
Model = RandomForestClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    "n_estimators": [200,250,300],
    "min_samples_leaf": np.arange(1, 4),
    "max_features": [np.arange(0.3, 0.6, 0.1),'sqrt'],
    "max_samples": np.arange(0.4, 0.7, 0.1)}


#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over, y_train_over)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 300, 'min_samples_leaf': 1, 'max_samples': 0.6, 'max_features': 'sqrt'} with CV score=0.98237364
36211773:
```

Tuning Random Forest: Undersampled

In [46]:

```
# defining model
Model = RandomForestClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    "n_estimators": [200,250,300],
    "min_samples_leaf": np.arange(1, 4),
    "max_features": [np.arange(0.3, 0.6, 0.1),'sqrt'],
    "max_samples": np.arange(0.4, 0.7, 0.1)}


#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_un, y_train_un)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 200, 'min_samples_leaf': 3, 'max_samples': 0.6, 'max_features': 'sqrt'} with CV score=0.89242201
53519518:
```

Tuning Gradient Boosting: Original

In [48]:

```
# defining model
Model = GradientBoostingClassifier(random_state=1)
```

```
#Parameter grid to pass in RandomSearchCV
param_grid={
    "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]}

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, scoring=scorer, n_iter=50, n_jobs = -1, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train, y_train)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'subsample': 0.7, 'n_estimators': 125, 'max_features': 0.7, 'learning_rate': 0.2} with CV score=0.75930099624367
14:
```

Tuning Gradient Boosting: Oversampled

In [49]:

```
# defining model
Model = GradientBoostingClassifier(random_state=1)

#Parameter grid to pass in RandomSearchCV
param_grid={
    "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]}

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, scoring=scorer, n_iter=50, n_jobs = -1, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over, y_train_over)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'subsample': 0.7, 'n_estimators': 125, 'max_features': 0.5, 'learning_rate': 1} with CV score=0.969362280862930
7:
```

Tuning Gradient Boosting: Undersampled

In [50]:

```
# defining model
Model = GradientBoostingClassifier(random_state=1)

#Parameter grid to pass in RandomSearchCV
param_grid={
    "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]}

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, scoring=scorer, n_iter=50, n_jobs = -1, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_un, y_train_un)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'subsample': 0.5, 'n_estimators': 100, 'max_features': 0.5, 'learning_rate': 0.2} with CV score=0.89754205454842
41:
```

Tuning Adaboost: Original

In [51]:

```
# defining model
Model = AdaBoostClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    "n_estimators": [100, 150, 200],
    "learning_rate": [0.2, 0.05],
    "base_estimator": [DecisionTreeClassifier(max_depth=1, random_state=1), DecisionTreeClassifier(max_depth=2, random_state=1), Deci
    ]
}

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train,y_train)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 200, 'learning_rate': 0.2, 'base_estimator': DecisionTreeClassifier(max_depth=3, random_state=
1)} with CV score=0.7708149599869344:
```

Tuning Adaboost: Oversampled

In [52]:

```
# defining model
Model = AdaBoostClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    "n_estimators": [100, 150, 200],
    "learning_rate": [0.2, 0.05],
    "base_estimator": [DecisionTreeClassifier(max_depth=1, random_state=1), DecisionTreeClassifier(max_depth=2, random_state=1), Deci
    ]
}

#Calling RandomizedSearchCV
```

```
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over,y_train_over)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 200, 'learning_rate': 0.2, 'base_estimator': DecisionTreeClassifier(max_depth=3, random_state=
1)} with CV score=0.9740525167670946:
```

Tuning Adaboost: Undersampled

In [53]:

```
# defining model
Model = AdaBoostClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
    "n_estimators": [100, 150, 200],
    "learning_rate": [0.2, 0.05],
    "base_estimator": [DecisionTreeClassifier(max_depth=1, random_state=1), DecisionTreeClassifier(max_depth=2, random_state=1), Deci
    ]
}

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, n_iter=50, n_jobs = -1, scoring=scorer, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_un,y_train_un)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'n_estimators': 100, 'learning_rate': 0.2, 'base_estimator': DecisionTreeClassifier(max_depth=3, random_state=
1)} with CV score=0.8885676955740649:
```

Tuning Logistic Regression: Original

In [54]:

```
# defining model
Model = LogisticRegression(random_state=1)

#Parameter grid to pass in RandomSearchCV
param_grid = {'C': np.arange(0.1,1.1,0.1)}

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, scoring=scorer, n_iter=50, n_jobs = -1, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train, y_train)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'C': 0.2} with CV score=0.5044667646578475:
```

Tuning Logistic Regression: Oversampled

In [55]:
```
# defining model
Model = LogisticRegression(random_state=1)

#Parameter grid to pass in RandomSearchCV
param_grid = {'C': np.arange(0.1,1.1,0.1)}

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, scoring=scorer, n_iter=50, n_jobs = -1, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over, y_train_over)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'C': 0.1} with CV score=0.87941624122865:
```

Tuning Logistic Regression: Undersampled

In [56]:
```
# defining model
Model = LogisticRegression(random_state=1)

#Parameter grid to pass in RandomSearchCV
param_grid = {'C': np.arange(0.1,1.1,0.1)}

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=param_grid, scoring=scorer, n_iter=50, n_jobs = -1, cv=5, ran

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_un, y_train_un)

print("Best parameters are {} with CV score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score_))
```

```
Best parameters are {'C': 0.1} with CV score=0.8488567695574065:
```

Model Performance comparison

Choose best score of each classifier for a total of 6 chosen models.

Bagging: ***Oversampled data CV score=0.983583873824214:***

- 0.7323452555936634 with original data
- 0.8885758615057977 with undersampled data

Decision Tree: ***Oversampled data CV score=0.8740479375486185:***

- 0.4956230605912134 with original data

- 0.809186673199412 with undersampled data

Random Forest: ***Oversampled data CV score=0.9823736436211773***

- 0.6939408786542545 with original data
- 0.8924220153519518 with undersampled data

Gradient Boosting: ***Oversampled data CV score=0.9693622808629307:***

- 0.7593009962436714 with original data
- 0.8975420545484241 with undersampled data

Adaboost: ***Oversampled data CV score=0.9740525167670946:***

- 0.7708149599869344 with original data
- 0.8885676955740649 with undersampled data

Logistic Regression: ***Oversampled data CV score=0.87941624122865:***

- 0.5044667646578475 with original data
- 0.8488567695574065 with undersampled data

Bagging

In [62]:

```
# BAGGING: Create new pipeline with best parameters
bg_tuned = BaggingClassifier(
    n_estimators=70,
    max_samples=0.8,
    max_features=0.8)

bg_tuned.fit(X_train_over, y_train_over)
```

Out[62]:

```
BaggingClassifier(max_features=0.8, max_samples=0.8, n_estimators=70)
```

In [71]:

```
# Check performance on oversampled train set
bg_train_perf = model_performance_classification_sklearn(bg_tuned, X_train_over, y_train_over)
bg_train_perf
```

Out[71]:

	Accuracy	Recall	Precision	F1
0	1.000	1.000	1.000	1.000

In [72]:

```
# Check performance on validation set
bg_val_perf = model_performance_classification_sklearn(bg_tuned, X_val, y_val)
bg_val_perf
```

Out[72]:

	Accuracy	Recall	Precision	F1
0	0.987	0.842	0.920	0.879

Decision Tree

In [65]:

```
# DECISION TREE: Create new pipeline with best parameters
dtree_tuned = DecisionTreeClassifier(
    min_samples_leaf=7,
    min_impurity_decrease=0.001,
    max_leaf_nodes=15,
    max_depth=3)

dtree_tuned.fit(X_train_over, y_train_over)
```

Out[65]:

```
DecisionTreeClassifier(max_depth=3, max_leaf_nodes=15,
                       min_impurity_decrease=0.001, min_samples_leaf=7)
```

In [73]:

```
# Check performance on oversampled train set
dtree_train_perf = model_performance_classification_sklearn(dtree_tuned, X_train_over, y_train_over)
dtree_train_perf
```

Out[73]:

	Accuracy	Recall	Precision	F1
0	0.866	0.871	0.861	0.866

In [74]:

```
# Check performance on validation set
dtree_val_perf = model_performance_classification_sklearn(dtree_tuned, X_val, y_val)
dtree_val_perf
```

Out[74]:

	Accuracy	Recall	Precision	F1
0	0.841	0.818	0.231	0.361

Random Forest

In [60]:

```
# RANDOM FOREST: Create new pipeline with best parameters
rf_tuned = RandomForestClassifier(
    max_features=0.3,
    random_state=1,
    max_samples=0.6,
    n_estimators=300,
    min_samples_leaf=1,)

rf_tuned.fit(X_train_over, y_train_over)
```

Out[60]:
```
RandomForestClassifier(max_features=0.3, max_samples=0.6, n_estimators=300,
                       random_state=1)
```

In [75]:
```
# Check performance on oversampled train set
rf_train_perf = model_performance_classification_sklearn(rf_tuned, X_train_over, y_train_over)
rf_train_perf
```

Out[75]:

	Accuracy	Recall	Precision	F1
0	1.000	1.000	1.000	1.000

In [76]:
```
# Check performance on validation set
rf_val_perf = model_performance_classification_sklearn(rf_tuned, X_val, y_val)
rf_val_perf
```

Out[76]:

	Accuracy	Recall	Precision	F1
0	0.989	0.851	0.936	0.892

Gradient Boosting

In [67]:
```
# GRADIENT BOOSTING: Create new pipeline with best parameters
gbm_tuned = GradientBoostingClassifier(
    subsample=0.7,
    n_estimators=125,
    max_features=0.5,
    learning_rate=1)

gbm_tuned.fit(X_train_over, y_train_over)
```

Out[67]:
```
GradientBoostingClassifier(learning_rate=1, max_features=0.5, n_estimators=125,
                           subsample=0.7)
```

In [77]:
```
# Check performance on oversampled train set
gbm_train_perf = model_performance_classification_sklearn(gbm_tuned, X_train_over, y_train_over)
gbm_train_perf
```

Out[77]:

	Accuracy	Recall	Precision	F1
0	0.993	0.992	0.994	0.993

In [78]:
```
# Check performance on validation set
gbm_val_perf = model_performance_classification_sklearn(gbm_tuned, X_val, y_val)
gbm_val_perf
```

Out[78]:

	Accuracy	Recall	Precision	F1

	Accuracy	Recall	Precision	F1
0	0.963	0.860	0.615	0.717

Adaboost

In [68]:
```
# ADABOOST: Create new pipeline with best parameters
ada_tuned = AdaBoostClassifier(
    n_estimators= 200,
    learning_rate= 0.2,
    base_estimator= DecisionTreeClassifier(max_depth=3,random_state=1)
)

ada_tuned.fit(X_train_over, y_train_over)
```

Out[68]:
```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=3,
                                                         random_state=1),
                   learning_rate=0.2, n_estimators=200)
```

In [79]:
```
# Check performance on oversampled train set
ada_train_perf = model_performance_classification_sklearn(ada_tuned, X_train_over, y_train_over)
ada_train_perf
```

Out[79]:

	Accuracy	Recall	Precision	F1
0	0.993	0.990	0.996	0.993

In [80]:
```
# Check performance on validation set
ada_val_perf = model_performance_classification_sklearn(ada_tuned, X_val, y_val)
ada_val_perf
```

Out[80]:

	Accuracy	Recall	Precision	F1
0	0.984	0.872	0.837	0.854

Logistic Regression

In [69]:
```
# LOGISTIC REGRESSION: Create new pipeline with best parameters

log_tuned = LogisticRegression(C = 0.1)

log_tuned.fit(X_train_over, y_train_over)
```

Out[69]:
```
LogisticRegression(C=0.1)
```

In [81]:
```
# Check performance on oversampled train set
```

```python
log_train_perf = model_performance_classification_sklearn(log_tuned, X_train_over, y_train_over)
log_train_perf
```

Out[81]:

	Accuracy	Recall	Precision	F1
0	0.880	0.880	0.880	0.880

In [82]:

```python
# Check performance on validation set
log_val_perf = model_performance_classification_sklearn(log_tuned, X_val, y_val)
log_val_perf
```

Out[82]:

	Accuracy	Recall	Precision	F1
0	0.879	0.836	0.290	0.431

In [87]:

```python
# training performance comparison

models_train_comp_df = pd.concat(
    [
        bg_train_perf.T,
        dtree_train_perf.T,
        rf_train_perf.T,
        gbm_train_perf.T,
        ada_train_perf.T,
        log_train_perf.T
    ],
    axis=1,
)

models_train_comp_df.columns = [
    "Bagging tuned with oversampled data",
    "Decision Tree tuned with oversampled data",
    "Random forest tuned with oversampled data",
    "Gradient Boosting tuned with oversampled data",
    "AdaBoost classifier tuned with oversampled data",
    "Logistic Regression tuned with oversampled data",
]
print("Training performance comparison:")
models_train_comp_df
```

```
Training performance comparison:
```

Out[87]:

	Bagging tuned with oversampled data	Decision Tree tuned with oversampled data	Random forest tuned with oversampled data	Gradient Boosting tuned with oversampled data	AdaBoost classifier tuned with oversampled data	Logistic Regression tuned with oversampled data
Accuracy	1.000	0.866	1.000	0.993	0.993	0.880
Recall	1.000	0.871	1.000	0.992	0.990	0.880
Precision	1.000	0.861	1.000	0.994	0.996	0.880

	Bagging tuned with oversampled data	Decision Tree tuned with oversampled data	Random forest tuned with oversampled data	Gradient Boosting tuned with oversampled data	AdaBoost classifier tuned with oversampled data	Logistic Regression tuned with oversampled data
F1	1.000	0.866	1.000	0.993	0.993	0.880

In [91]:

```
# validation performance comparison

models_val_comp_df = pd.concat(
    [
        bg_val_perf.T,
        dtree_val_perf.T,
        rf_val_perf.T,
        gbm_val_perf.T,
        ada_val_perf.T,
        log_val_perf.T
    ],
    axis=1,
)

models_val_comp_df.columns = [
    "Bagging tuned with oversampled data",
    "Decision Tree tuned with oversampled data",
    "Random forest tuned with oversampled data",
    "Gradient Boosting tuned with oversampled data",
    "AdaBoost classifier tuned with oversampled data",
    "Logistic Regression tuned with oversampled data",
]
print("Validation performance comparison:")
models_val_comp_df
```

Validation performance comparison:

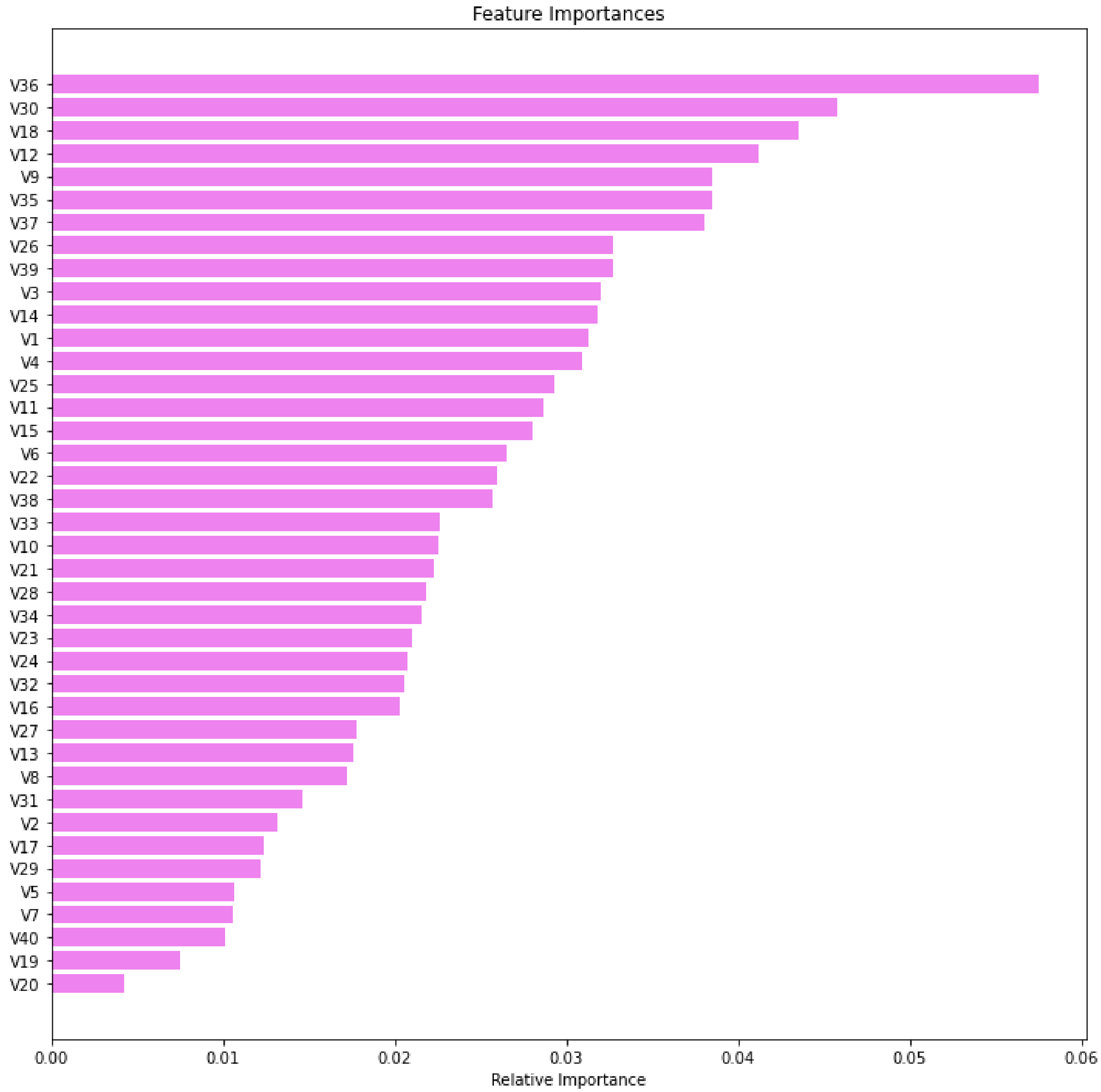
Out[91]:

	Bagging tuned with oversampled data	Decision Tree tuned with oversampled data	Random forest tuned with oversampled data	Gradient Boosting tuned with oversampled data	AdaBoost classifier tuned with oversampled data	Logistic Regression tuned with oversampled data
Accuracy	0.987	0.841	0.989	0.963	0.984	0.879
Recall	0.842	0.818	0.851	0.860	0.872	0.836
Precision	0.920	0.231	0.936	0.615	0.837	0.290
F1	0.879	0.361	0.892	0.717	0.854	0.431

In [107…

```
### Important features of the final model

feature_names = X_train.columns
importances = ada_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
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

In [109…

```
# Create pipeline for the best model

Model = Pipeline([
    ('scaler',StandardScaler()),
    ('clf', AdaBoostClassifier())
])
```

In [110…

```
X1 = df.drop(columns="Target")
Y1 = df["Target"]

# Built an existing test set above, don't need to divide data here

X_test1 = test.drop(columns="Target")
y_test1 = test["Target"]
```

In [111…

```
# impute missing values in X1

X1 = imputer.fit_transform(X1)
```

In [112…

```
sm = SMOTE(sampling_strategy=1, k_neighbors=5, random_state=1)
X_over1, y_over1 = sm.fit_resample(X1, Y1)
```

In [113…

```
Model.fit(X_train,y_train)
```

Out[113…

```
Pipeline(steps=[('scaler', StandardScaler()), ('clf', AdaBoostClassifier())])
```

In [114…

```
# pipeline object's accuracy on the train set

Model.score(X_train, y_train)
```

Out[114…

```
0.9762857142857143
```

In [115…

```
# pipeline object's accuracy on the test set

Model.score(X_test, y_test)
```

Out[115…

```
0.9724
```

Business Insights and Conclusions

Best model: Adaboost Classifier with oversampled data, highest recall score.

- Recall: 87.2%
- Accuracy: 98.4%
- Precision: 83.7%
- F1: 85.4%

The most important features noted in our chosen model are V36, V30, V18, V12, V9, and V35.

With high recall, we will be able to minimize false negatives, i.e. predicting a machine will have no failure when there will be a failure. This will control maintenance costs.

Considering the predictor variables, ReneWind may point out important sensors to implement generator failure warning signs.