# 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

## Please read the instructions carefully before starting the project.

This is a commented Jupyter IPython Notebook file in which all the instructions and tasks to be performed are mentioned.

- Blanks '**___**' are provided in the notebook that needs to be filled with an appropriate code to get the correct result. With every '**___**' blank, there is a comment that briefly describes what needs to be filled in the blank space.
- Identify the task to be performed correctly, and only then proceed to write the required code.
- Fill the code wherever asked by the commented lines like "# write your code here" or "# complete the code". Running incomplete code may throw error.
- Please run the codes in a sequential manner from the beginning to avoid any unnecessary errors.
- Add the results/observations (wherever mentioned) derived from the analysis in the presentation and submit the same.

# Importing necessary libraries

In [82]:
```
# 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,
    plot_confusion_matrix,
)
from sklearn import metrics

from sklearn.model_selection import train_test_split, StratifiedKFold, cr

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

# 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

In [83]:
```
df = pd.read_csv('train.csv.csv') ##  Complete the code to read the data
df_test = pd.read_csv('test.csv.csv') ##  Complete the code to read the d
```

# Data Overview

The initial steps to get an overview of any dataset is to:

- observe the first few rows of the dataset, to check whether the dataset has been loaded properly or not
- get information about the number of rows and columns in the dataset
- find out the data types of the columns to ensure that data is stored in the preferred format and the value of each property is as expected.
- check the statistical summary of the dataset to get an overview of the numerical columns of the data

## Checking the shape of the dataset

In [84]:
```
# Checking the number of rows and columns in the training data
df.shape ##  Complete the code to view dimensions of the train data
```

Out[84]:
```
(20000, 41)
```

In [85]:
```
# Checking the number of rows and columns in the test data
df_test.shape ##  Complete the code to view dimensions of the test data
```

Out[85]:
```
(5000, 41)
```

In [86]:
```
# let's create a copy of the training data
data = df.copy()
```

In [87]:
```
# let's create a copy of the training data
data_test = df_test.copy()
```

## Displaying the first few rows of the dataset

In [88]:
```
# let's view the first 5 rows of the data
data.head() ##  Complete the code to view top 5 rows of the data
```

Out[88]:

| | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | V10 | V11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | -4.465 | -4.679 | 3.102 | 0.506 | -0.221 | -2.033 | -2.911 | 0.051 | -1.522 | 3.762 | -5.715 |
| **1** | 3.366 | 3.653 | 0.910 | -1.368 | 0.332 | 2.359 | 0.733 | -4.332 | 0.566 | -0.101 | 1.914 |
| **2** | -3.832 | -5.824 | 0.634 | -2.419 | -1.774 | 1.017 | -2.099 | -3.173 | -2.082 | 5.393 | -0.771 |
| **3** | 1.618 | 1.888 | 7.046 | -1.147 | 0.083 | -1.530 | 0.207 | -2.494 | 0.345 | 2.119 | -3.053 |
| **4** | -0.111 | 3.872 | -3.758 | -2.983 | 3.793 | 0.545 | 0.205 | 4.849 | -1.855 | -6.220 | 1.998 |

In [89]:

```
# let's view the last 5 rows of the data
data_test.tail() ##  Complete the code to view last 5 rows of the data
```

Out[89]:

| | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | V10 | V |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **4995** | -5.120 | 1.635 | 1.251 | 4.036 | 3.291 | -2.932 | -1.329 | 1.754 | -2.985 | 1.249 | -6.8 |
| **4996** | -5.172 | 1.172 | 1.579 | 1.220 | 2.530 | -0.669 | -2.618 | -2.001 | 0.634 | -0.579 | -3.6 |
| **4997** | -1.114 | -0.404 | -1.765 | -5.879 | 3.572 | 3.711 | -2.483 | -0.308 | -0.922 | -2.999 | -0.1 |
| **4998** | -1.703 | 0.615 | 6.221 | -0.104 | 0.956 | -3.279 | -1.634 | -0.104 | 1.388 | -1.066 | -7.9 |
| **4999** | -0.604 | 0.960 | -0.721 | 8.230 | -1.816 | -2.276 | -2.575 | -1.041 | 4.130 | -2.731 | -3.2 |

## Checking the data types of the columns for the dataset

In [90]:

```
# let's check the data types of the columns in the dataset
data.info()
```

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

## Checking for duplicate values

In [91]:
```
# let's check for duplicate values in the data
data.duplicated().sum() ##  Complete the code to check duplicate entries
```

Out[91]: 0

## Checking for missing values

In [92]:
```
# let's check for missing values in the data
data.isna().sum() ##  Complete the code to check missing entries in the t
```

Out[92]:
```
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 [93]:
```
# let's check for missing values in the data
data_test.isna().sum()##  Complete the code to check missing entries in t
```

Out[93]:
```
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
```

## Statistical summary of the dataset

In [94]:
```
# let's view the statistical summary of the numerical columns in the data
data.describe()##  Complete the code to print the statitical summary of t
```

Out[94]:

| | V1 | V2 | V3 | V4 | V5 | V6 | V7 |
|---|---|---|---|---|---|---|---|
| **count** | 19982.000 | 19982.000 | 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 |
| **std** | 3.442 | 3.151 | 3.389 | 3.432 | 2.105 | 2.041 | 1.762 |
| **min** | -11.876 | -12.320 | -10.708 | -15.082 | -8.603 | -10.227 | -7.950 |
| **25%** | -2.737 | -1.641 | 0.207 | -2.348 | -1.536 | -2.347 | -2.031 |
| **50%** | -0.748 | 0.472 | 2.256 | -0.135 | -0.102 | -1.001 | -0.917 |
| **75%** | 1.840 | 2.544 | 4.566 | 2.131 | 1.340 | 0.380 | 0.224 |
| **max** | 15.493 | 13.089 | 17.091 | 13.236 | 8.134 | 6.976 | 8.006 |

# Exploratory Data Analysis

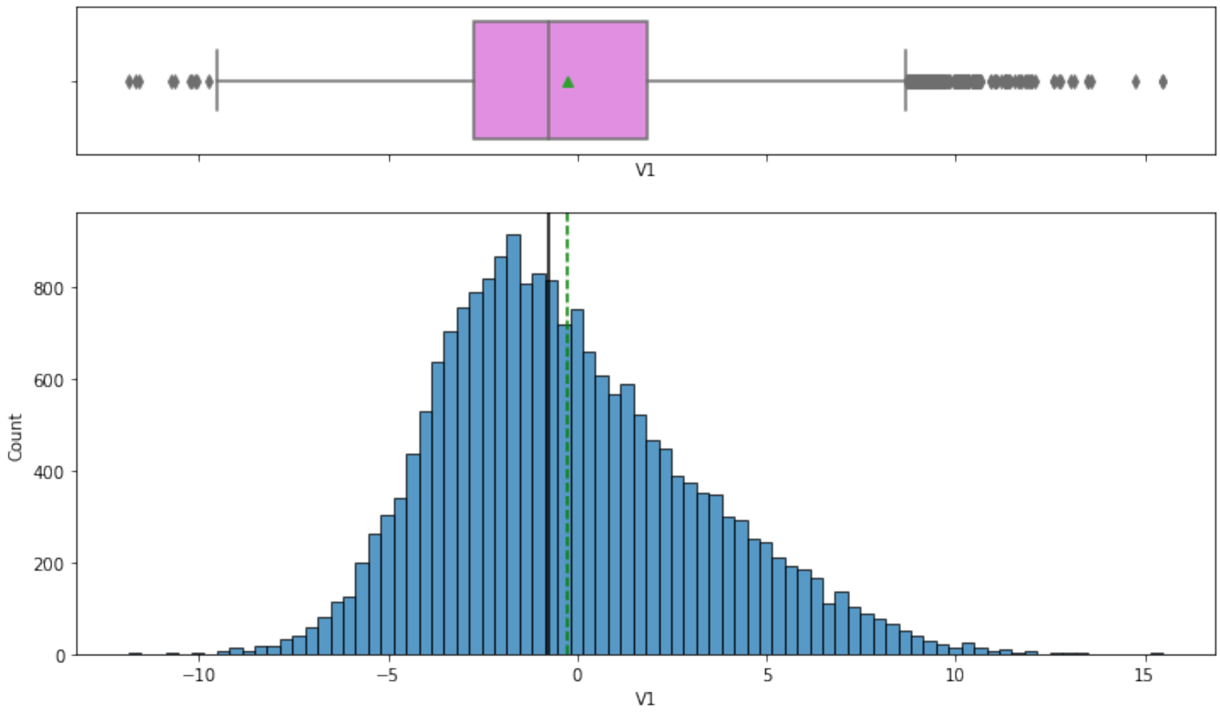
## Univariate analysis

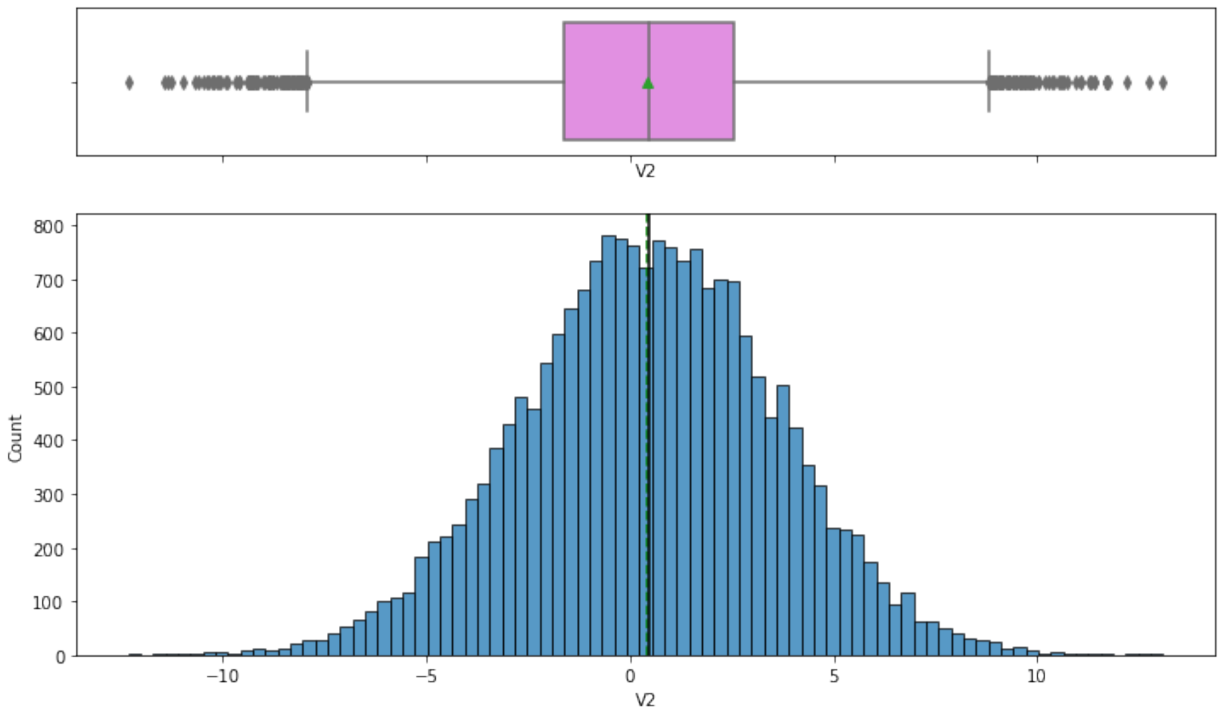
In [95]:

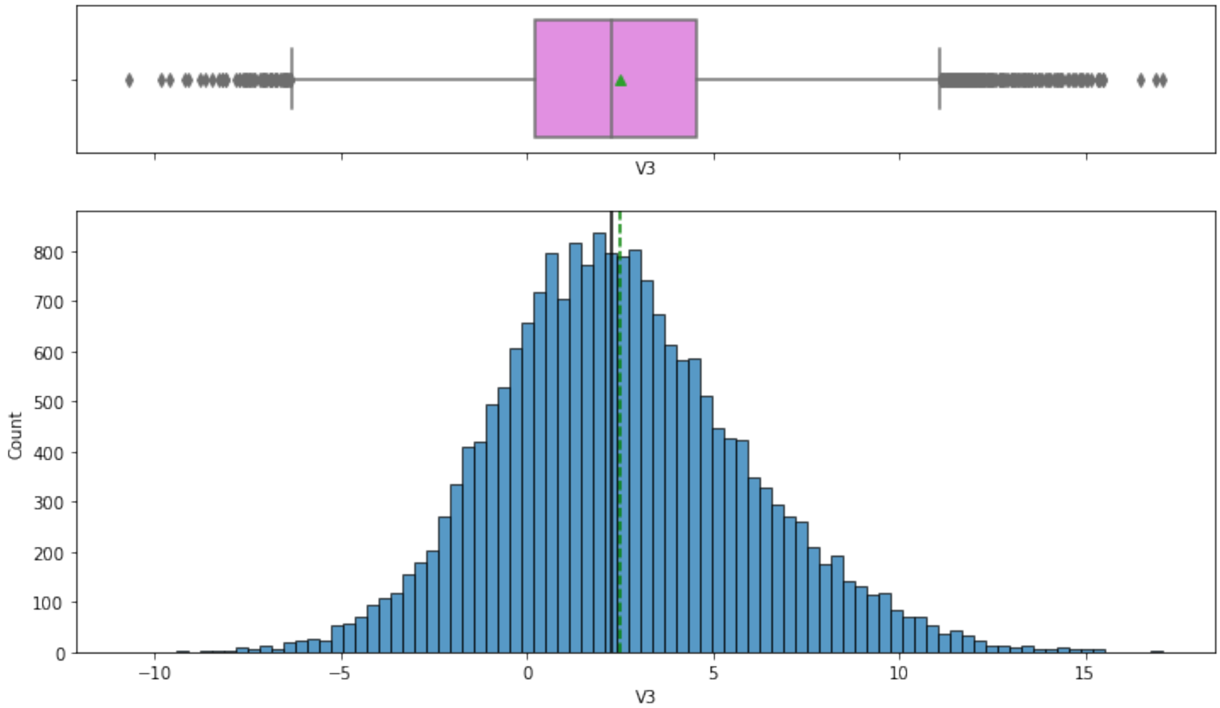
```
# function to plot a boxplot and a histogram along the same scale.


def histogram_boxplot(data, feature, figsize=(12, 7), kde=False, bins=Non
    """
    Boxplot and histogram combined

    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (12,7))
    kde: whether to the show density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows=2,  # Number of rows of the subplot grid= 2
        sharex=True,  # x-axis will be shared among all subplots
        gridspec_kw={"height_ratios": (0.25, 0.75)},
        figsize=figsize,
    )  # creating the 2 subplots
    sns.boxplot(
        data=data, x=feature, ax=ax_box2, showmeans=True, color="violet"
    )  # boxplot will be created and a triangle will indicate the mean va
    sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2, bins=bins, palette="w
    ) if bins else sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2
    )  # For histogram
    ax_hist2.axvline(
        data[feature].mean(), color="green", linestyle="--"
    )  # Add mean to the histogram
    ax_hist2.axvline(
        data[feature].median(), color="black", linestyle="-"
    )  # Add median to the histogram
```

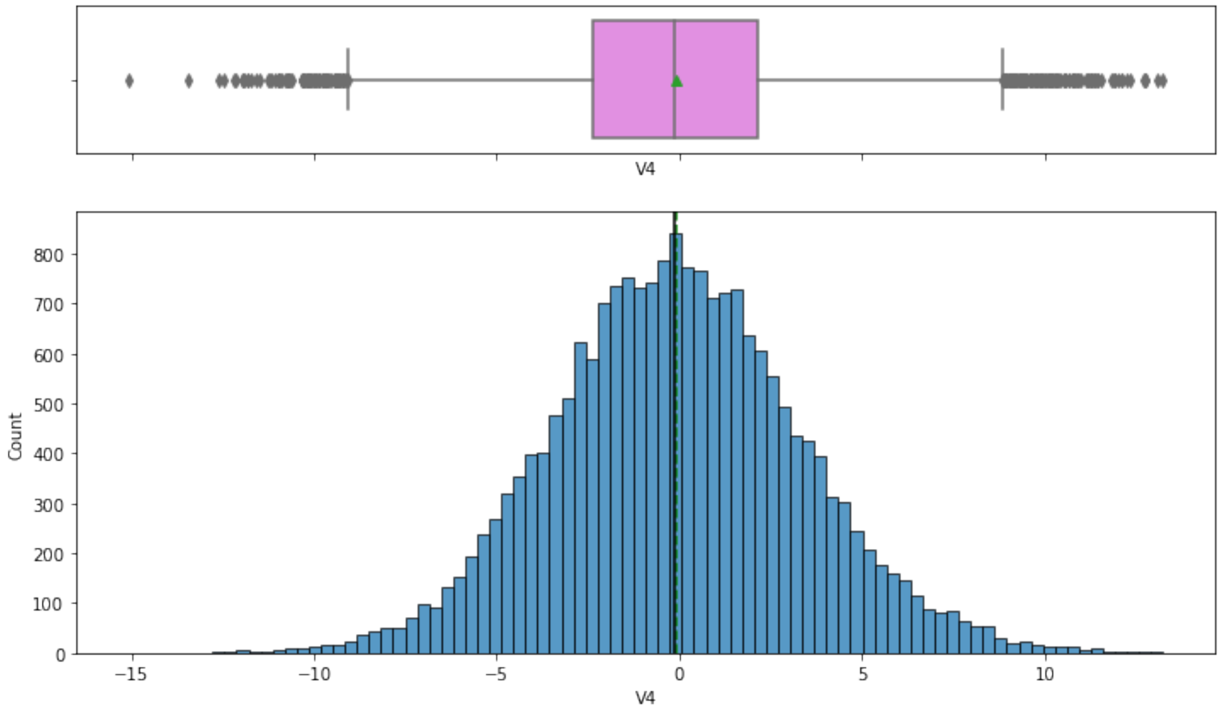
### Plotting histograms and boxplots for all the variables

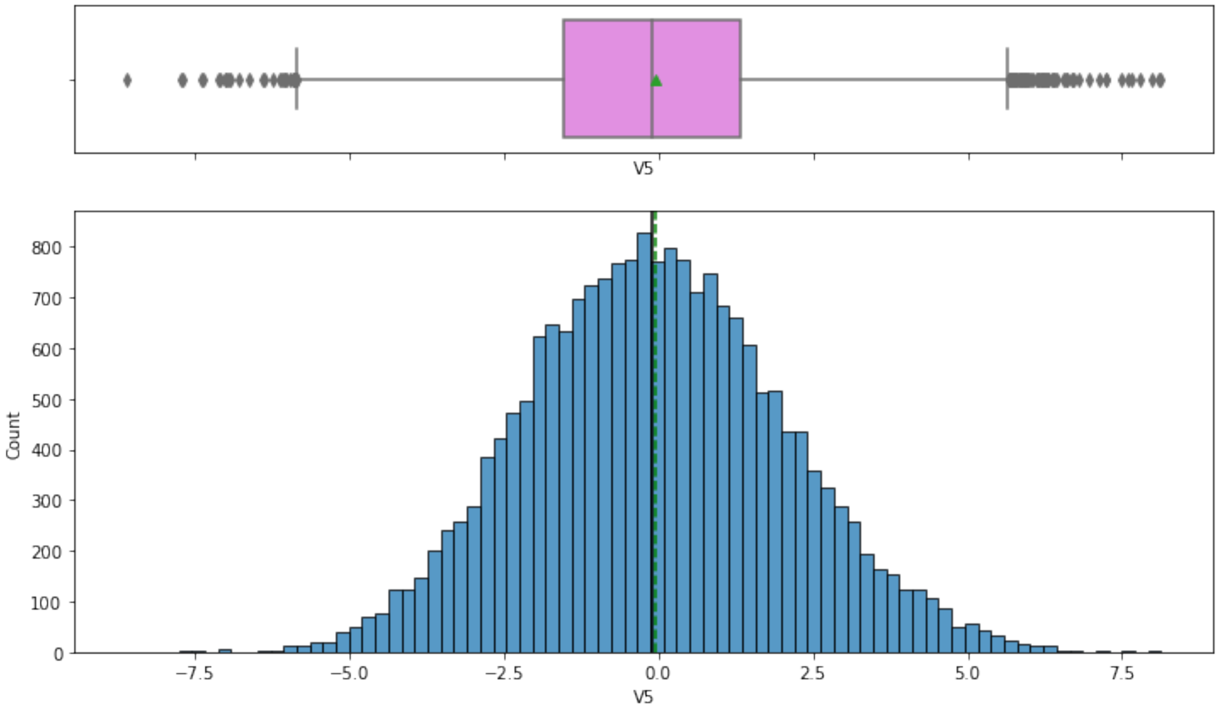
In [96]:
```
for feature in df.columns:
    histogram_boxplot(data, feature, figsize=(12, 7), kde=False, bins=Non
```

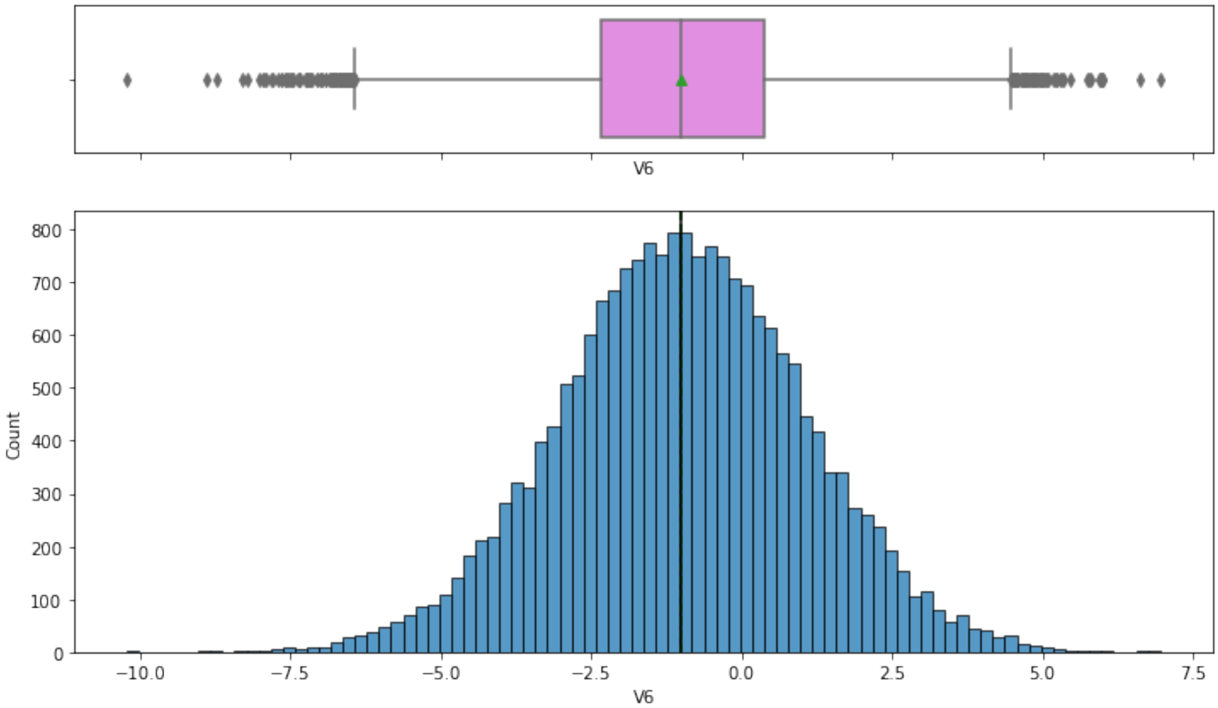


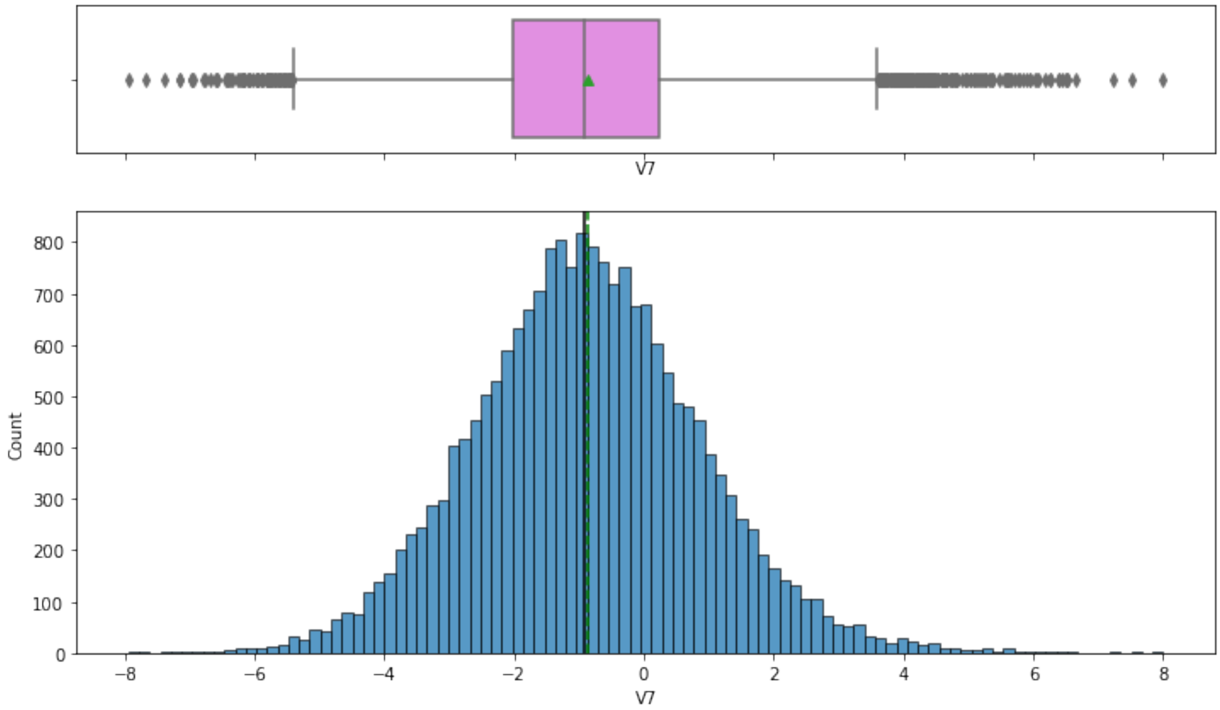



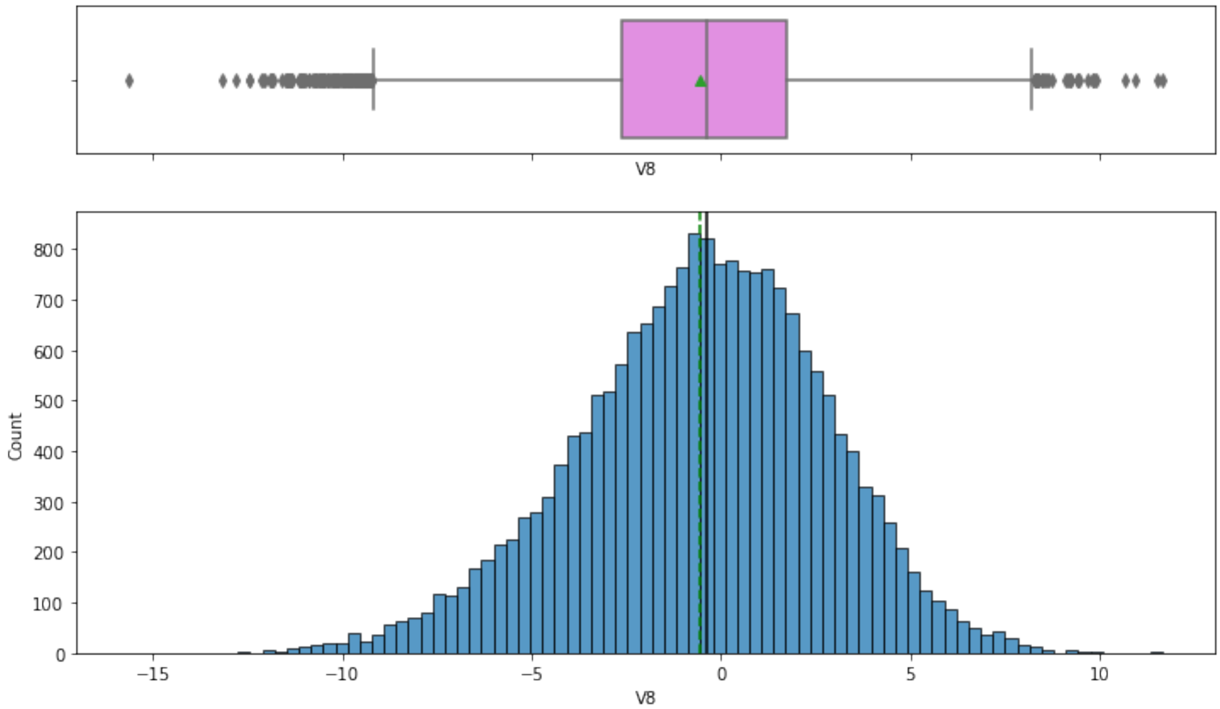

V3
Count
800
700
600
500
400
300
200
100
0
−10
−5
0
5
10
15
V3

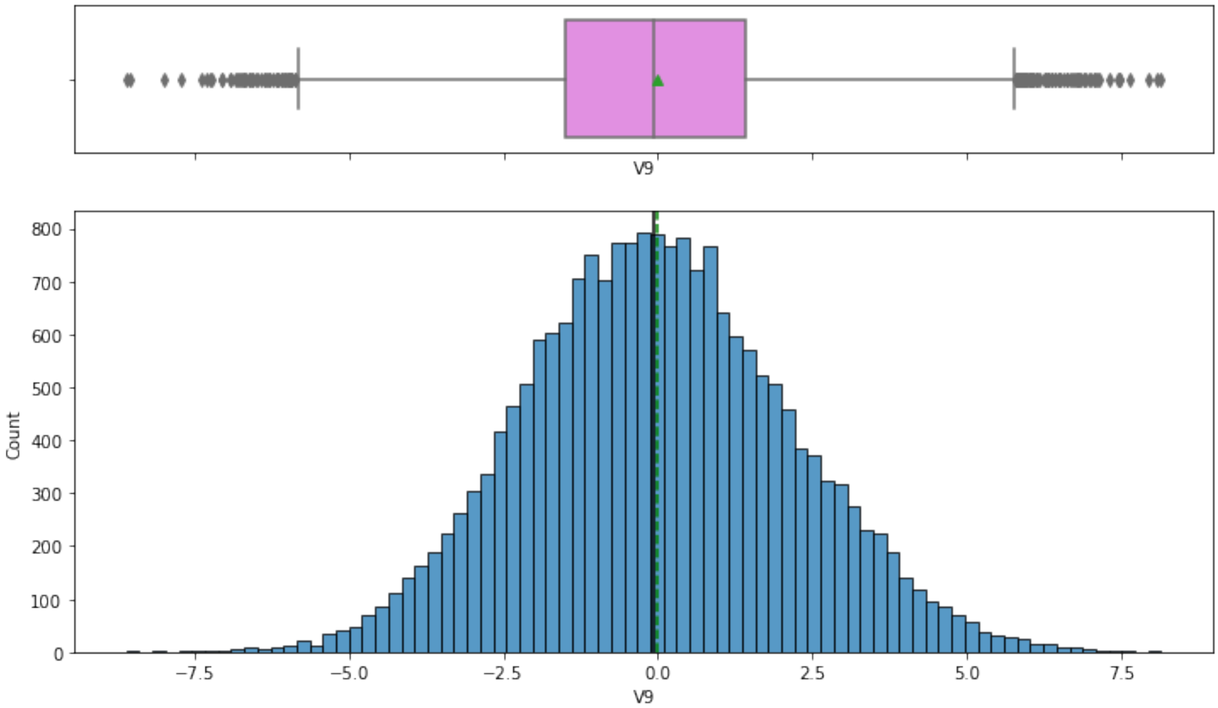

V4
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
V4

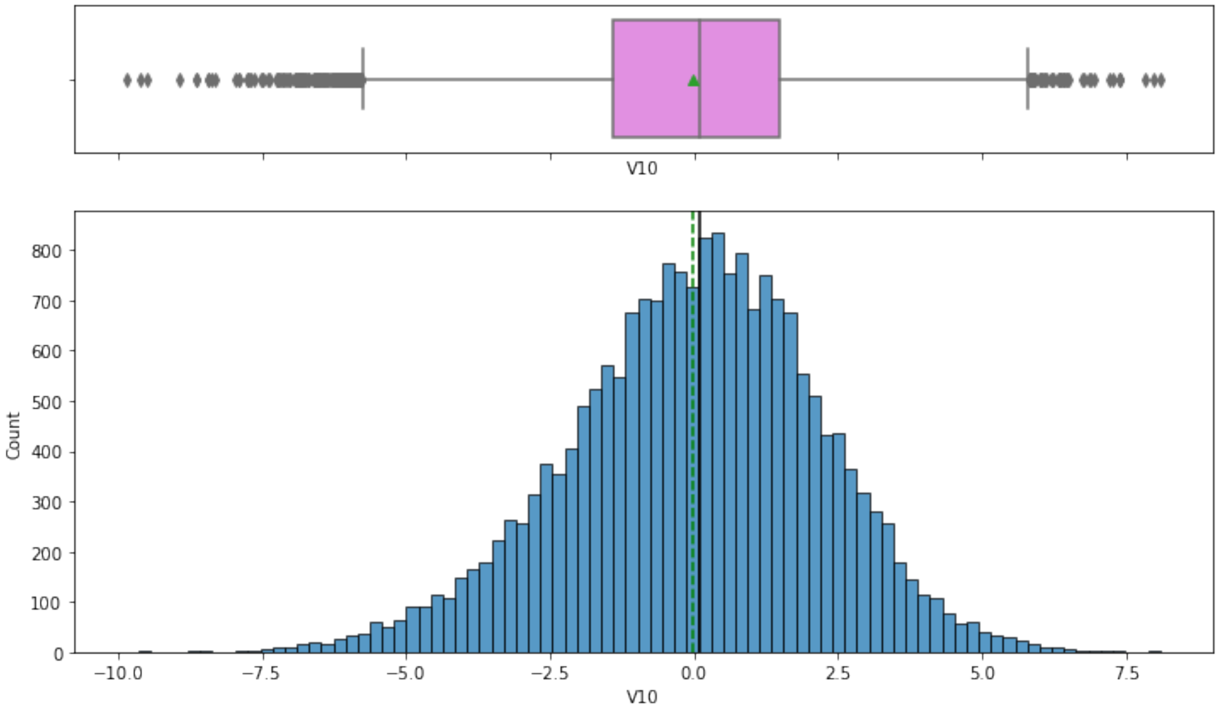

V5
Count
0
100
200
300
400
500
600
700
800
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V5

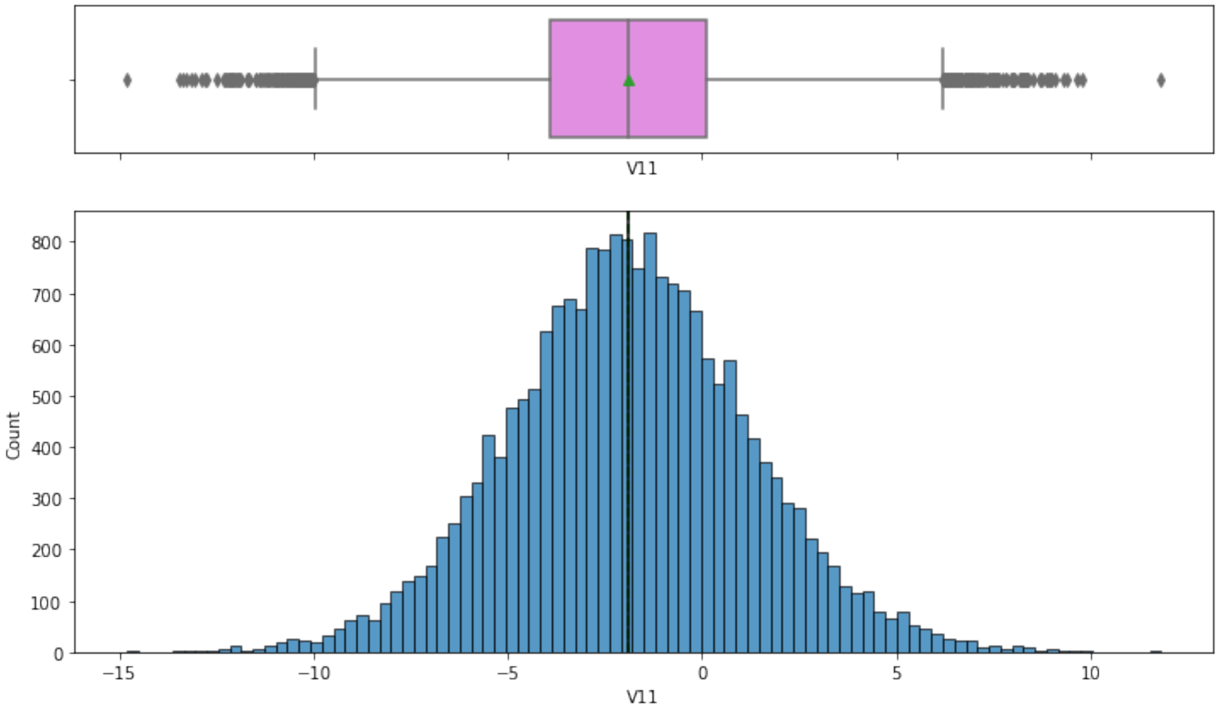

V6
Count
0
100
200
300
400
500
600
700
800
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V6

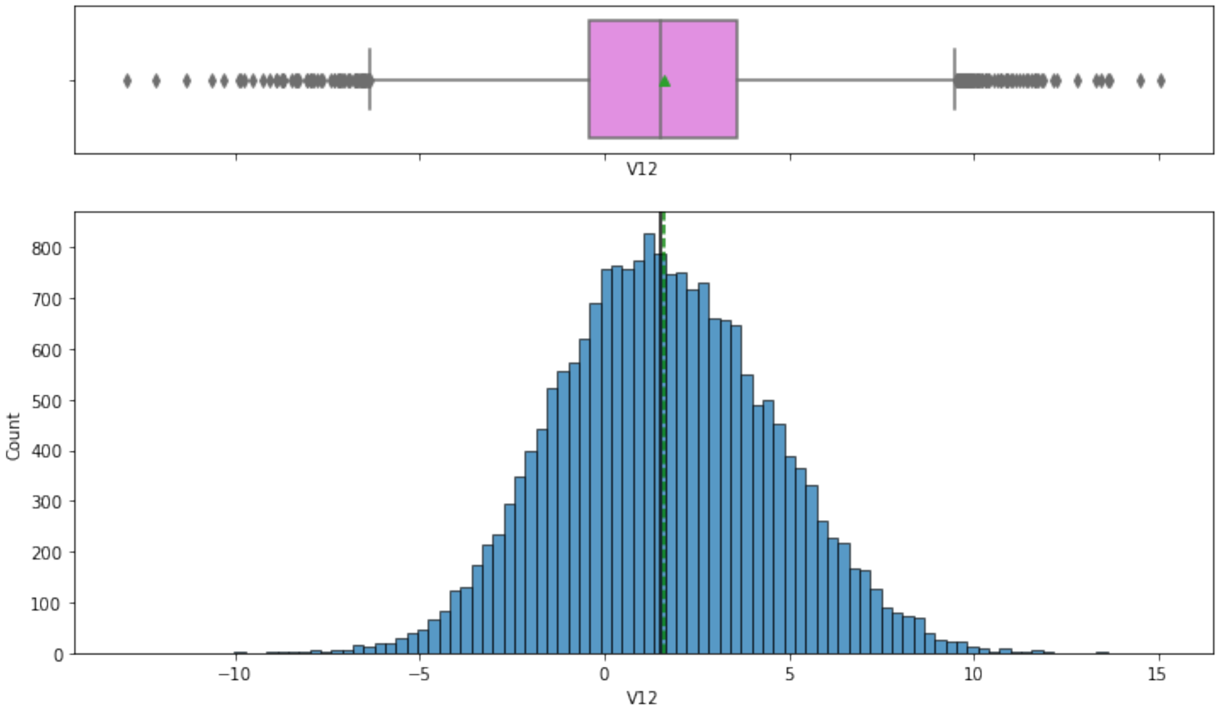

V7
Count
V7

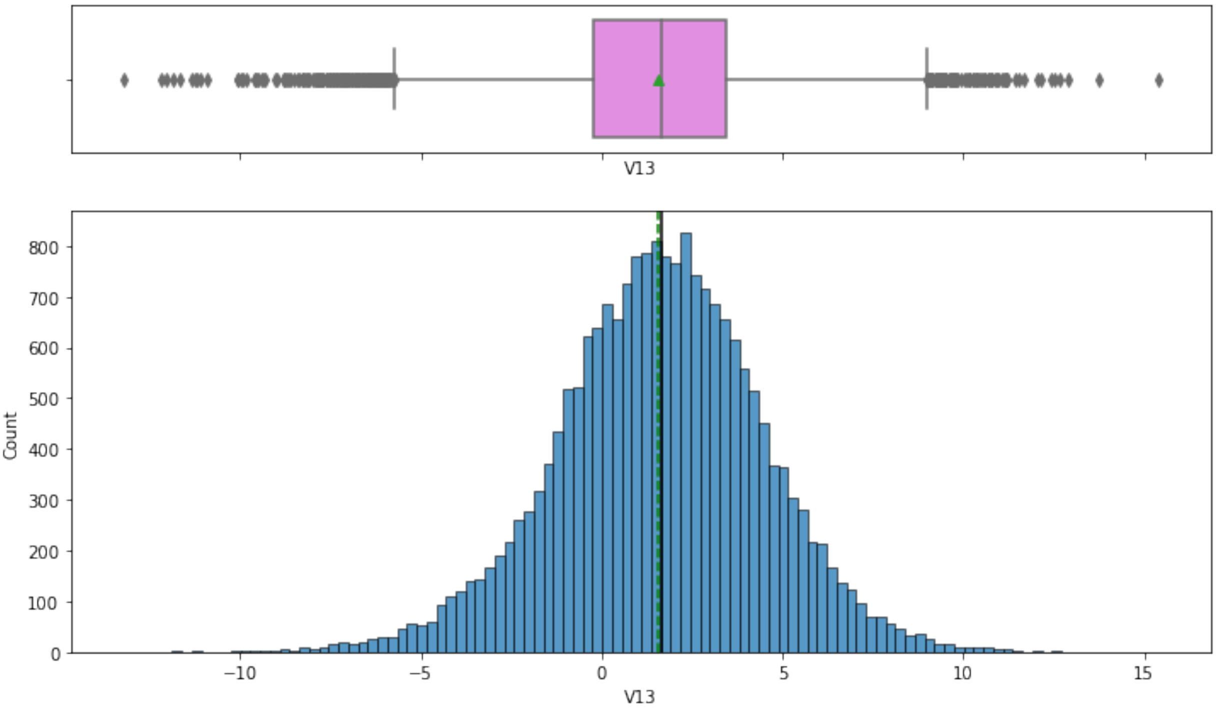

V8
Count
V8

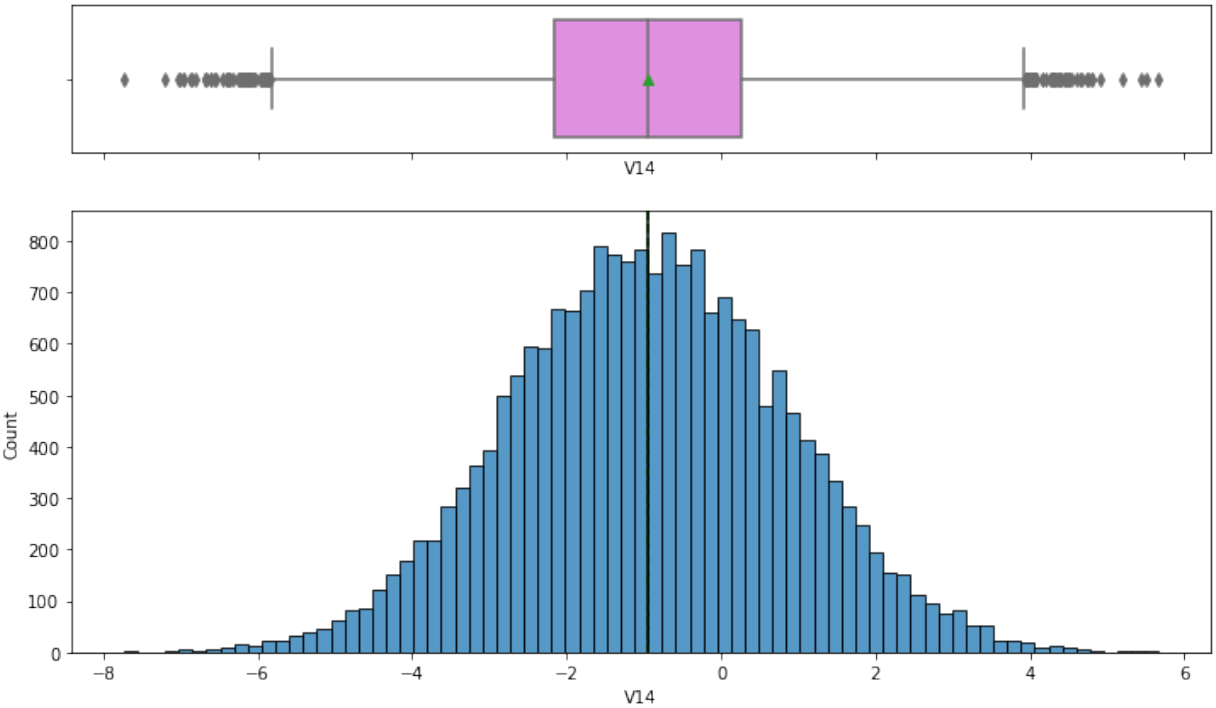

V9
Count
0
100
200
300
400
500
600
700
800
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V9

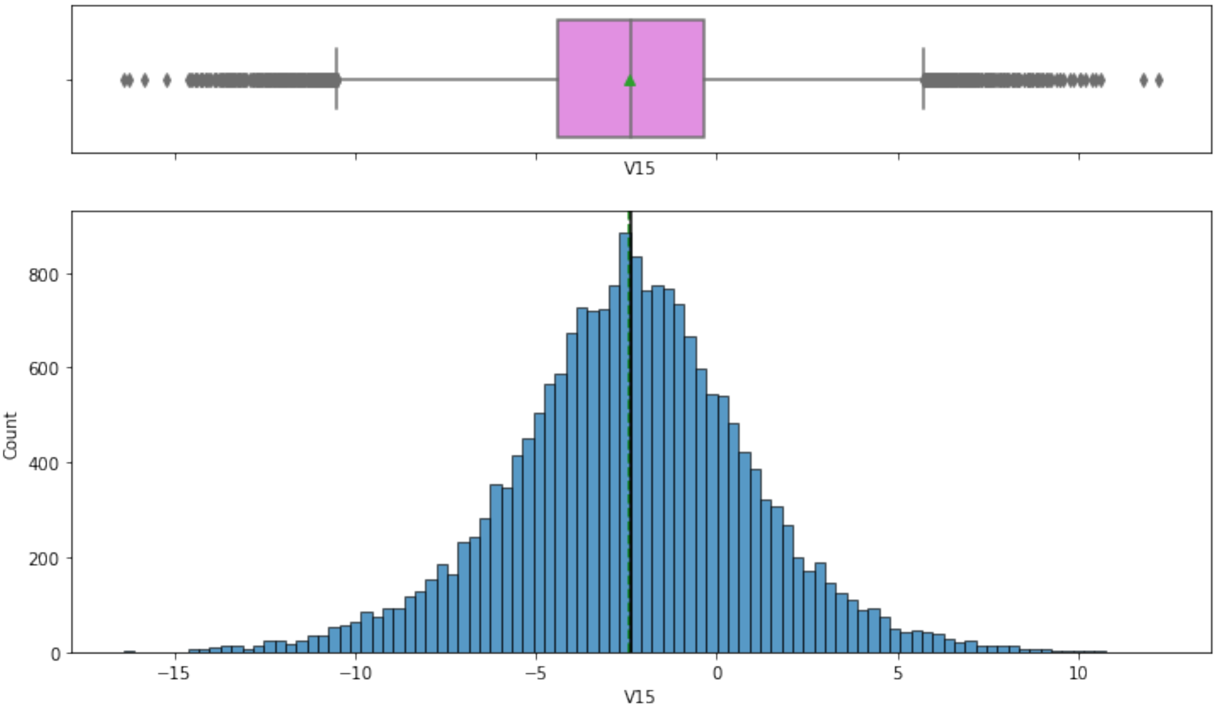

V10
Count
0
100
200
300
400
500
600
700
800
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V10

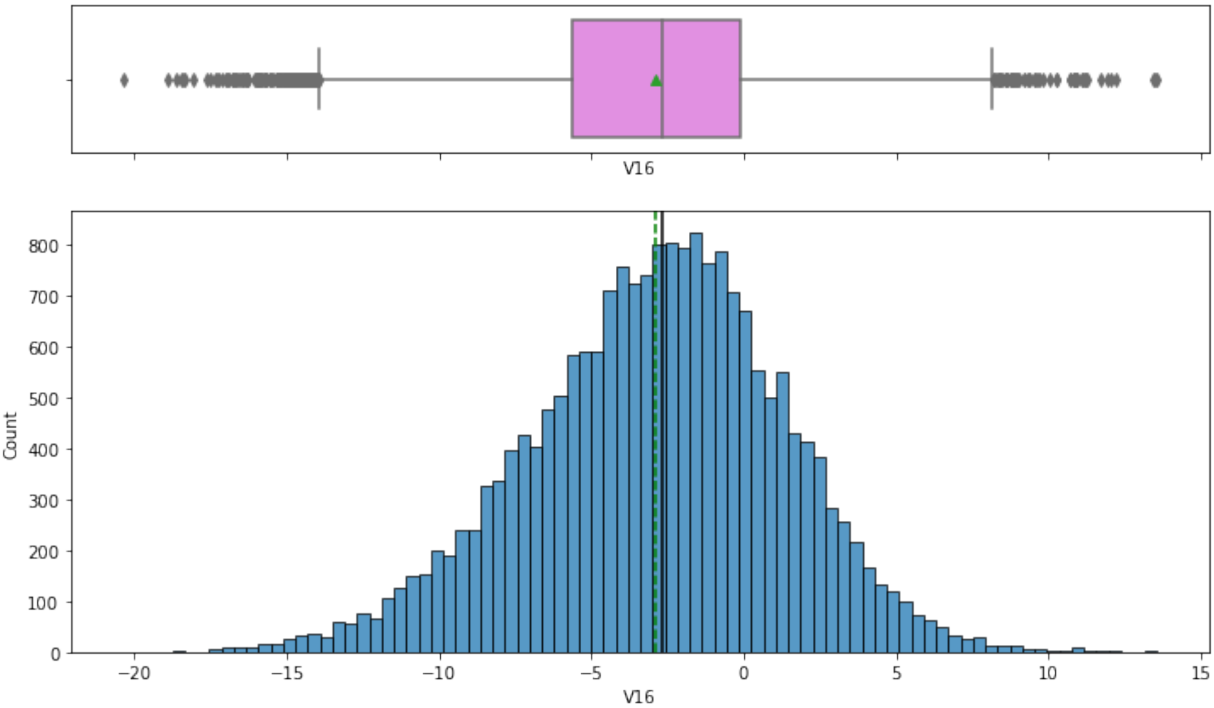

V11
Count
V11

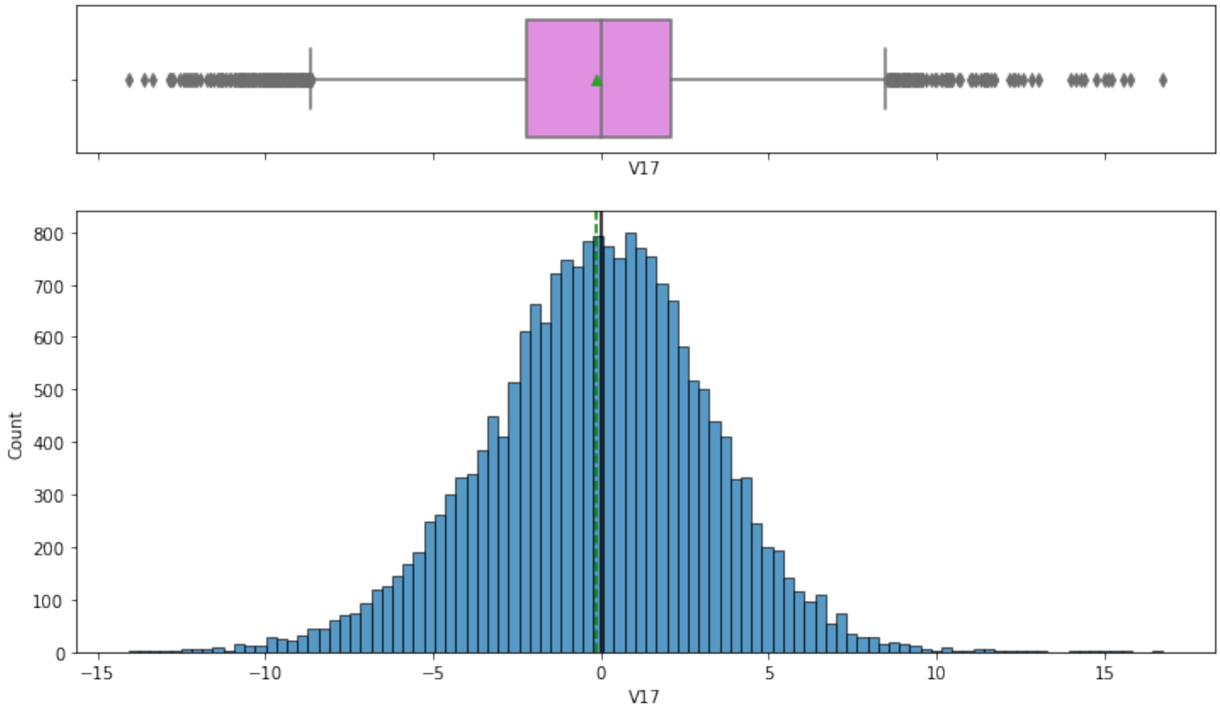

V12
Count
V12

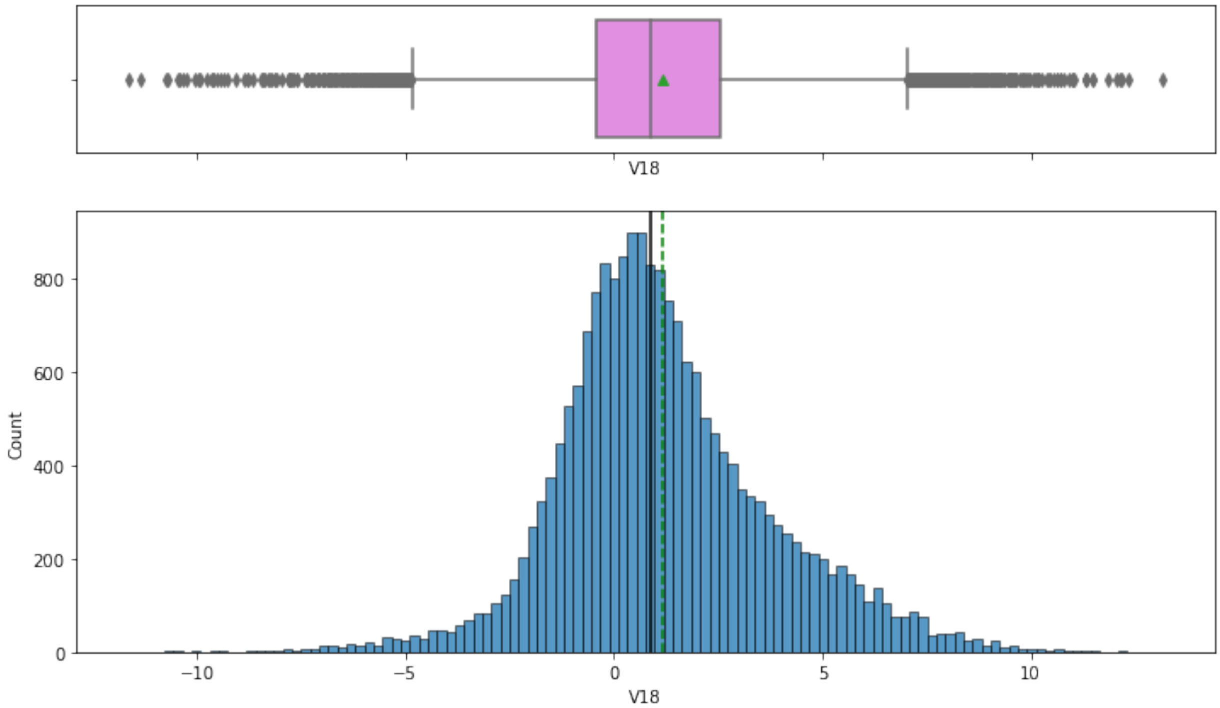

V13
Count
0
100
200
300
400
500
600
700
800
−10
−5
0
5
10
15
V13

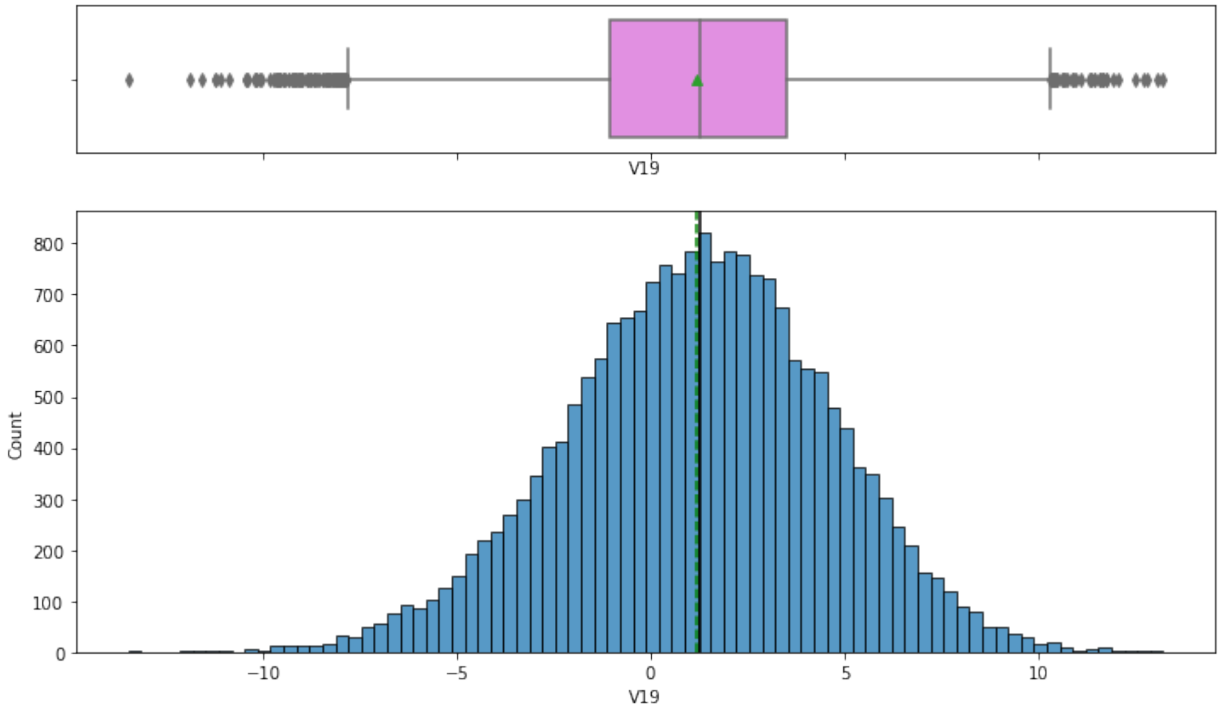

V14
Count
0
100
200
300
400
500
600
700
800
−8
−6
−4
−2
0
2
4
6
V14

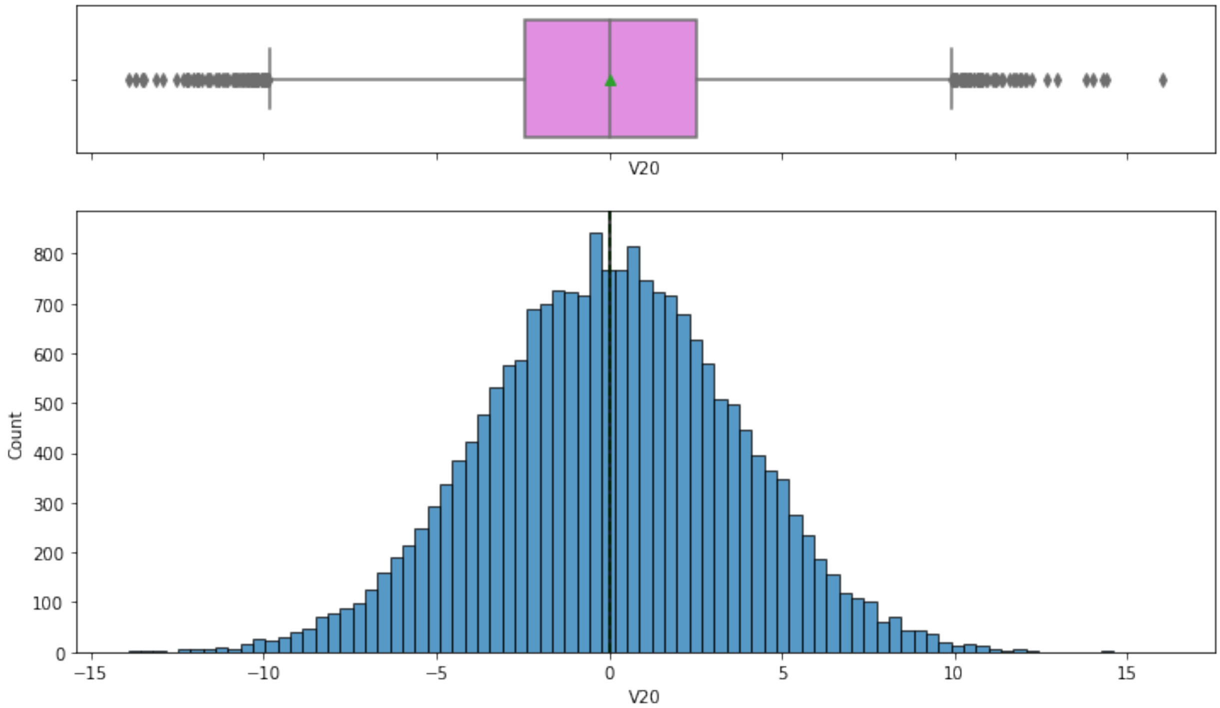

V15
800
600
400
200
0
Count
−15
−10
−5
0
5
10
V15

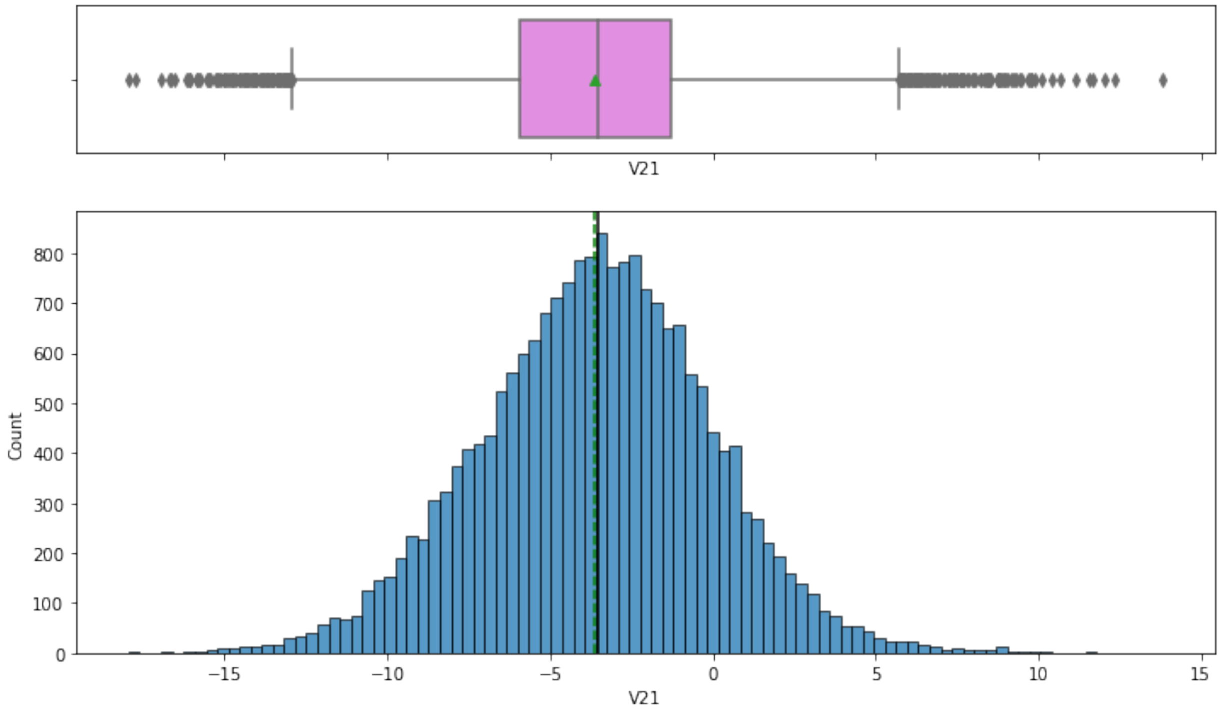

V16
800
700
600
500
400
300
200
100
0
Count
−20
−15
−10
−5
0
5
10
15
V16

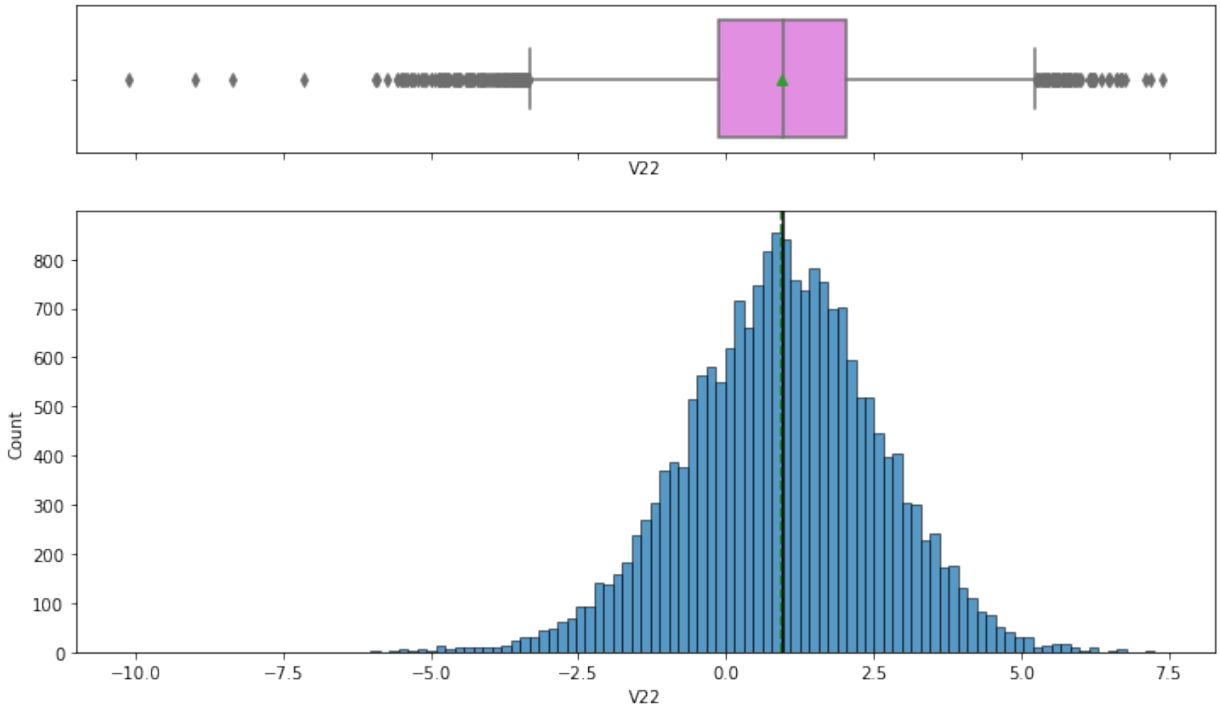

V17
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V17

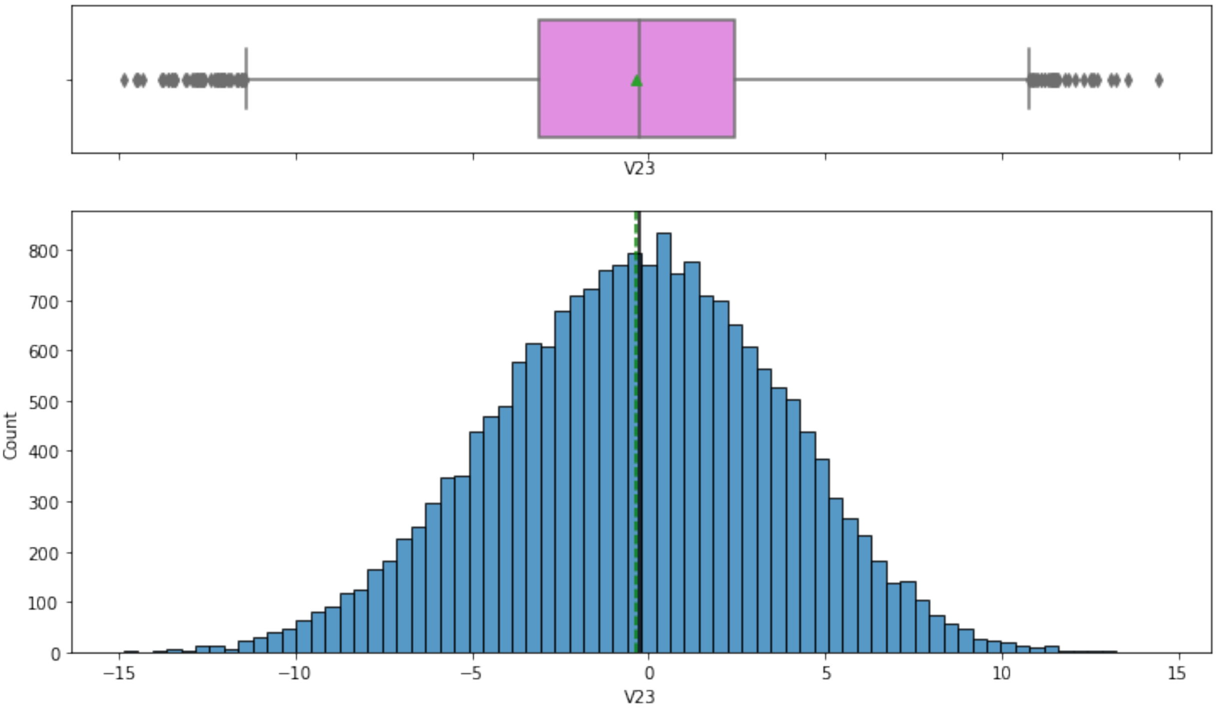

V18
Count
800
600
400
200
0
−10
−5
0
5
10
V18

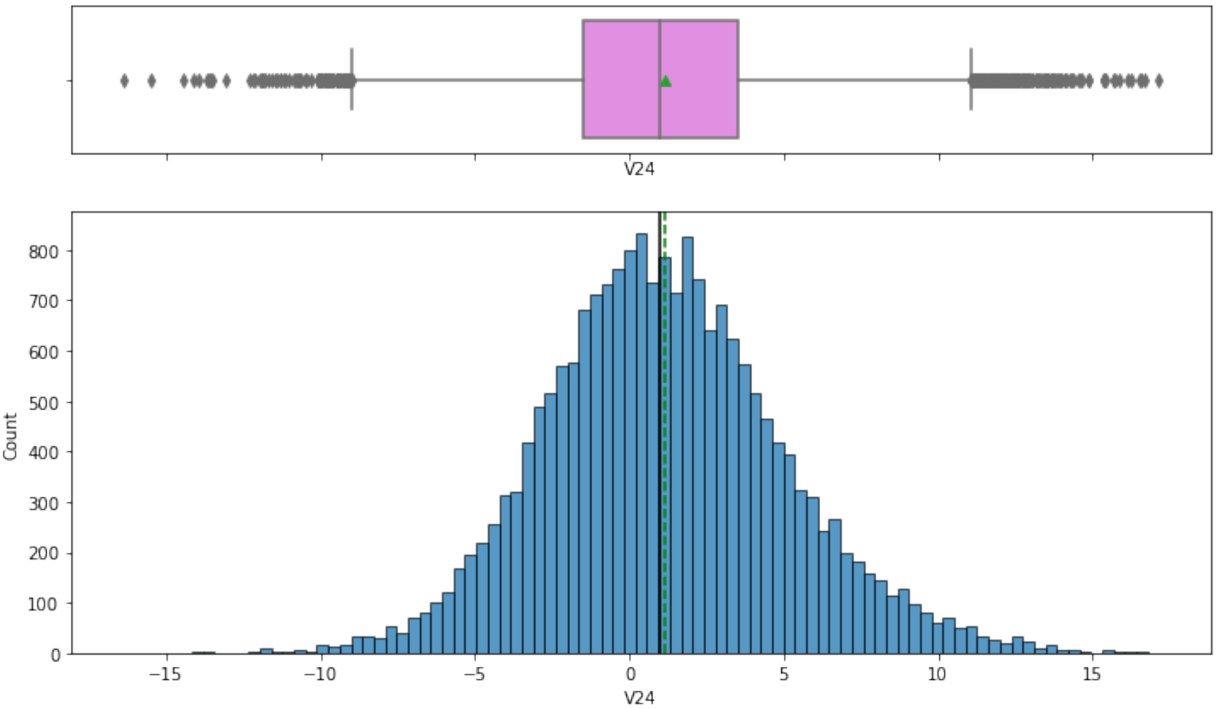

V19
Count
800
700
600
500
400
300
200
100
0
−10
−5
0
5
10
V19

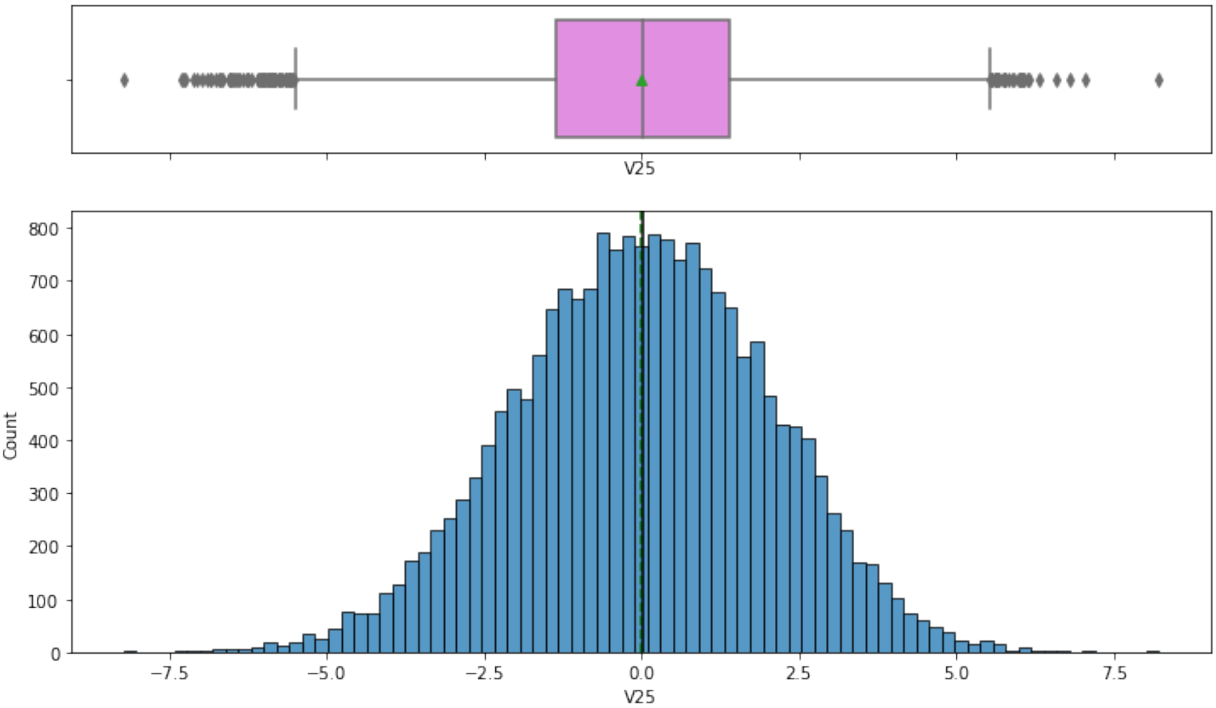

V20
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V20

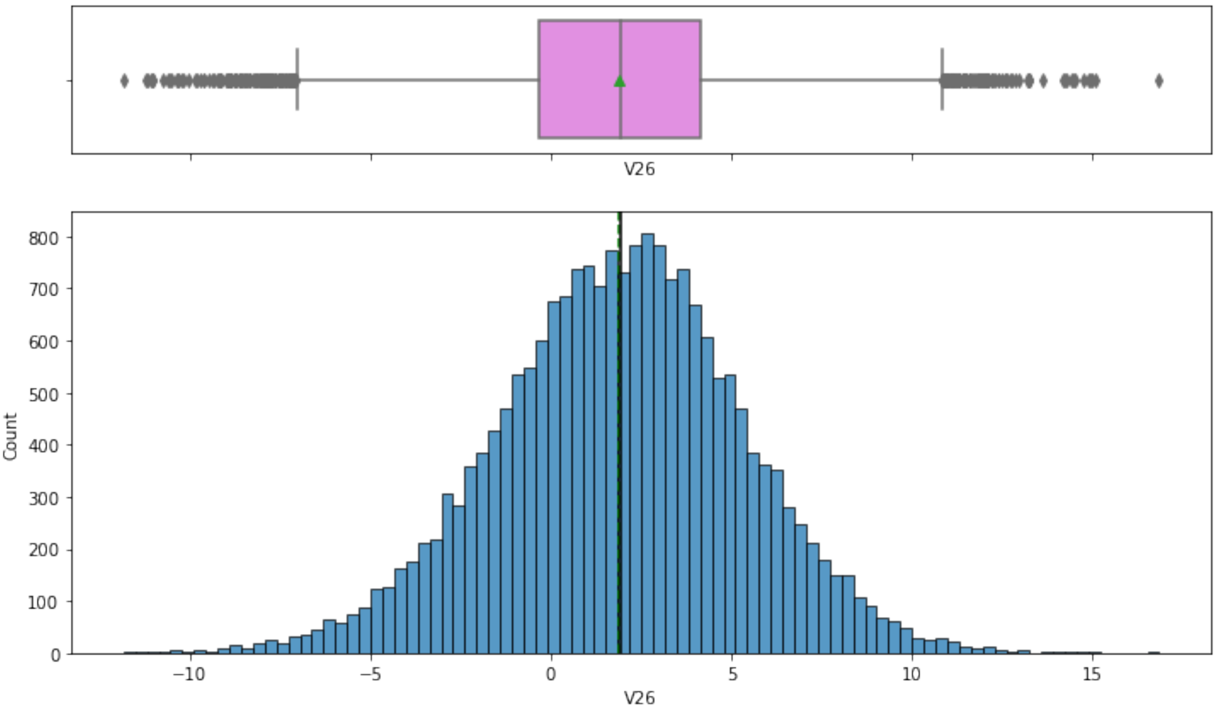

V21
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V21

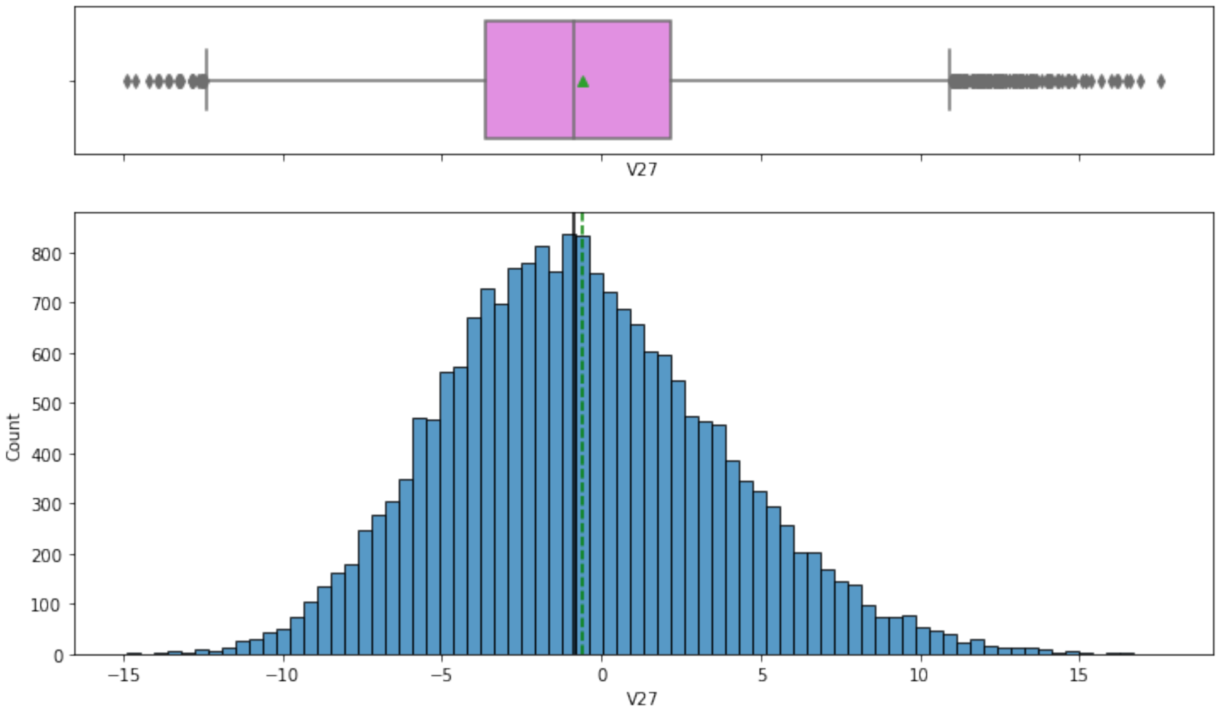

V22
Count
800
700
600
500
400
300
200
100
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V22

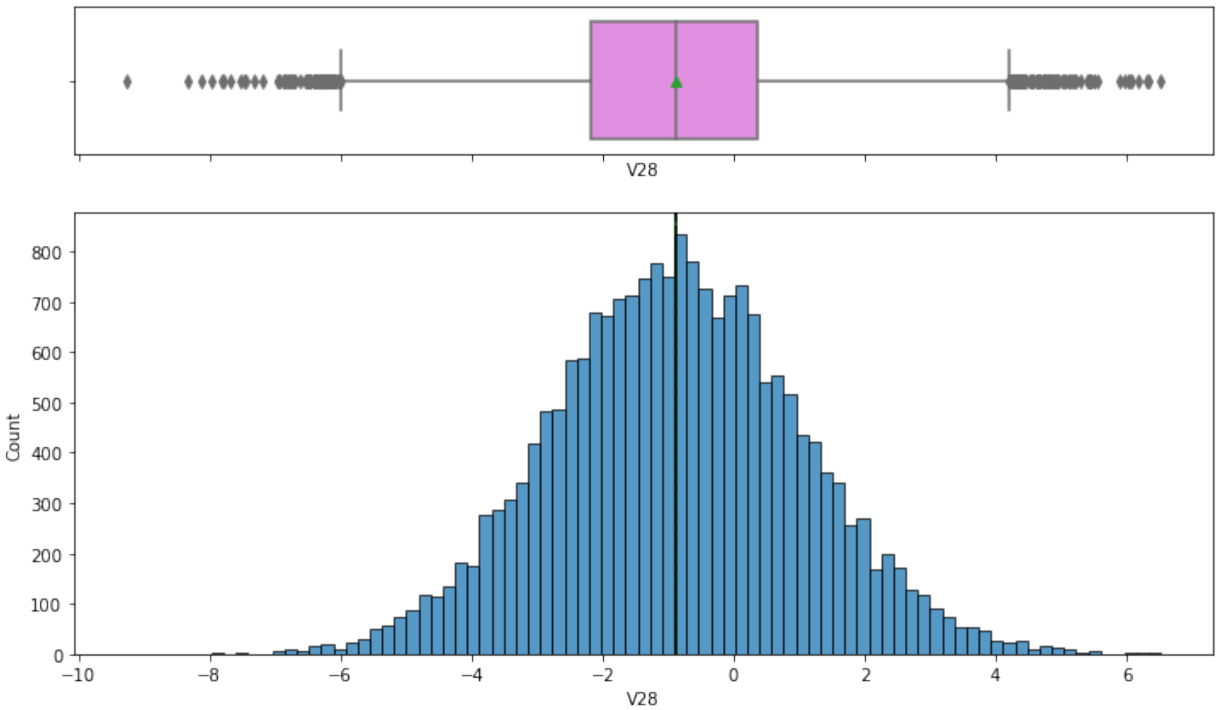

V23
Count
V23

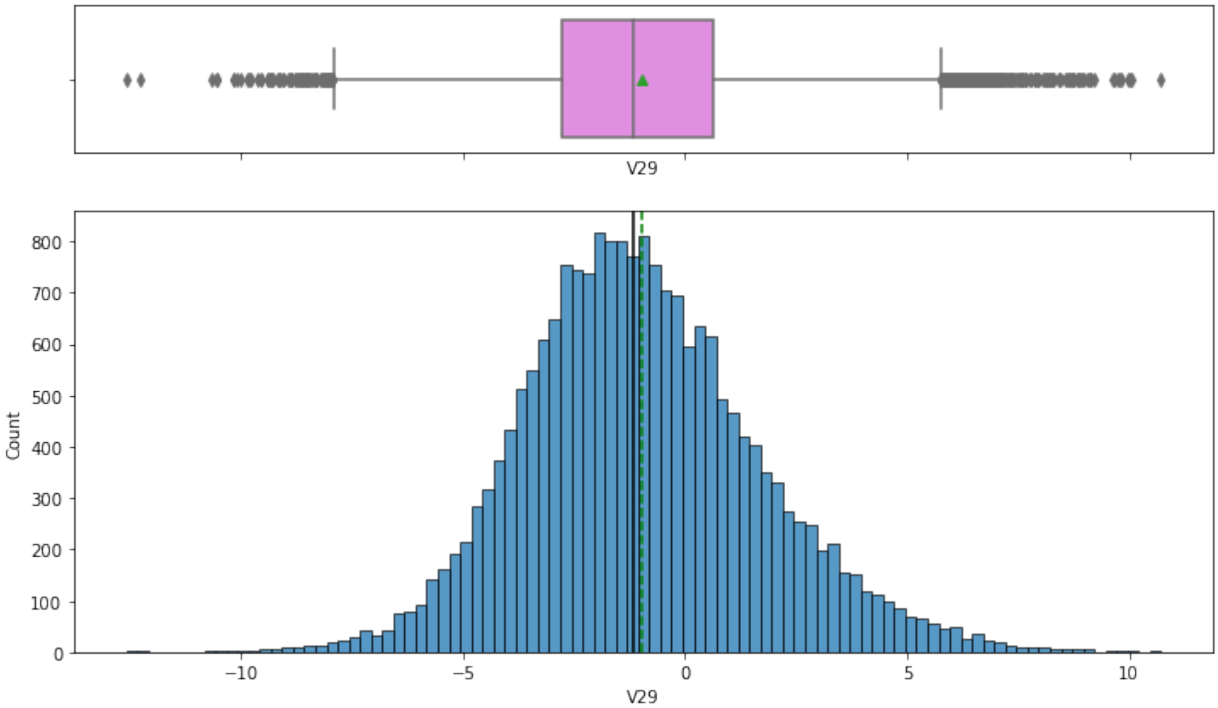

V24
Count
V24

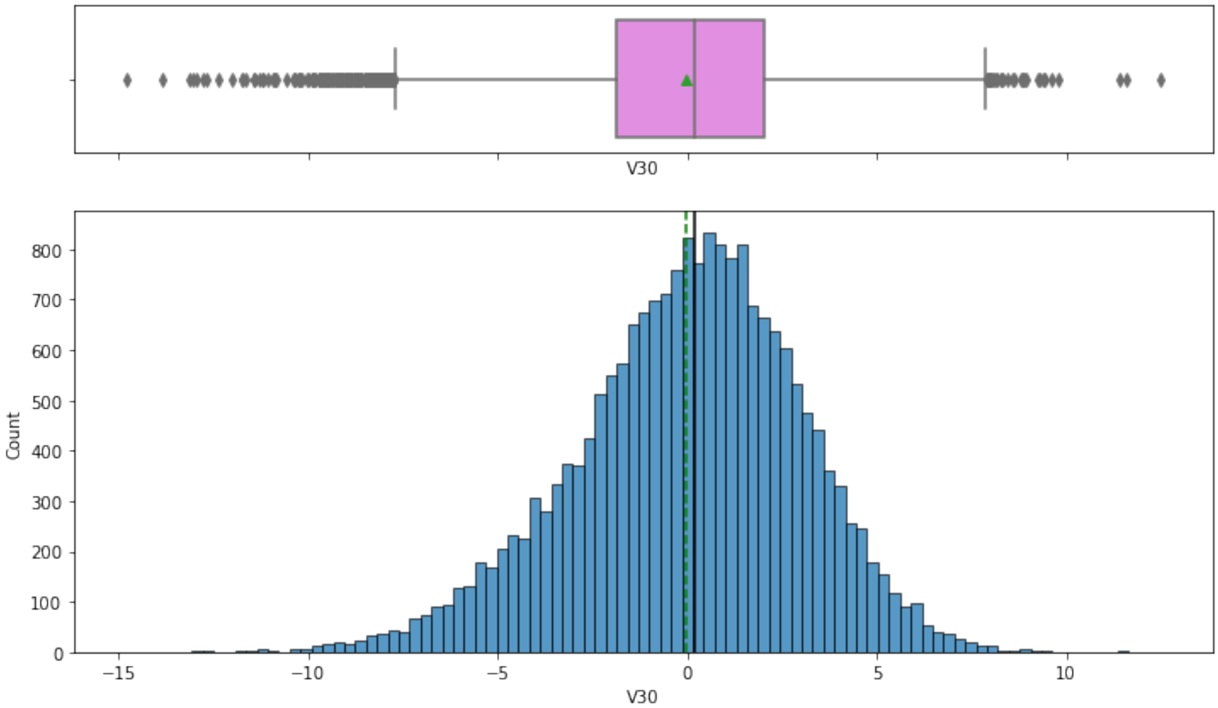

V25
Count
800
700
600
500
400
300
200
100
0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V25

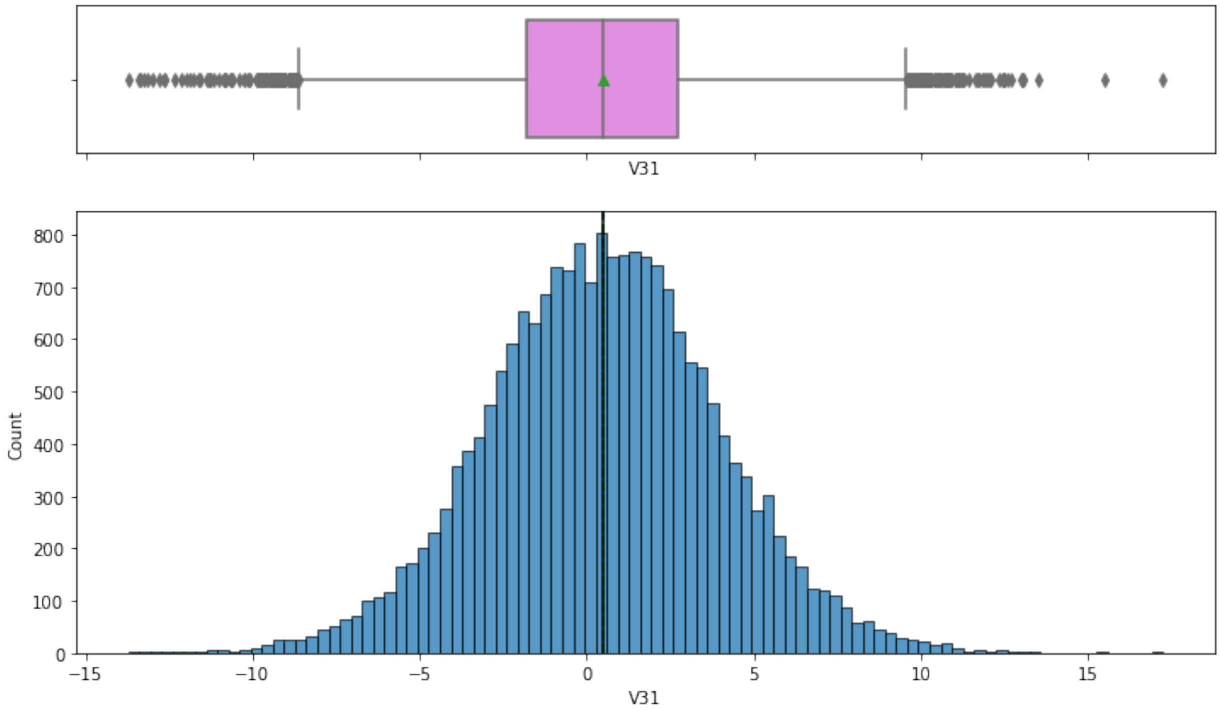

V26
Count
800
700
600
500
400
300
200
100
0
−10
−5
0
5
10
15
V26

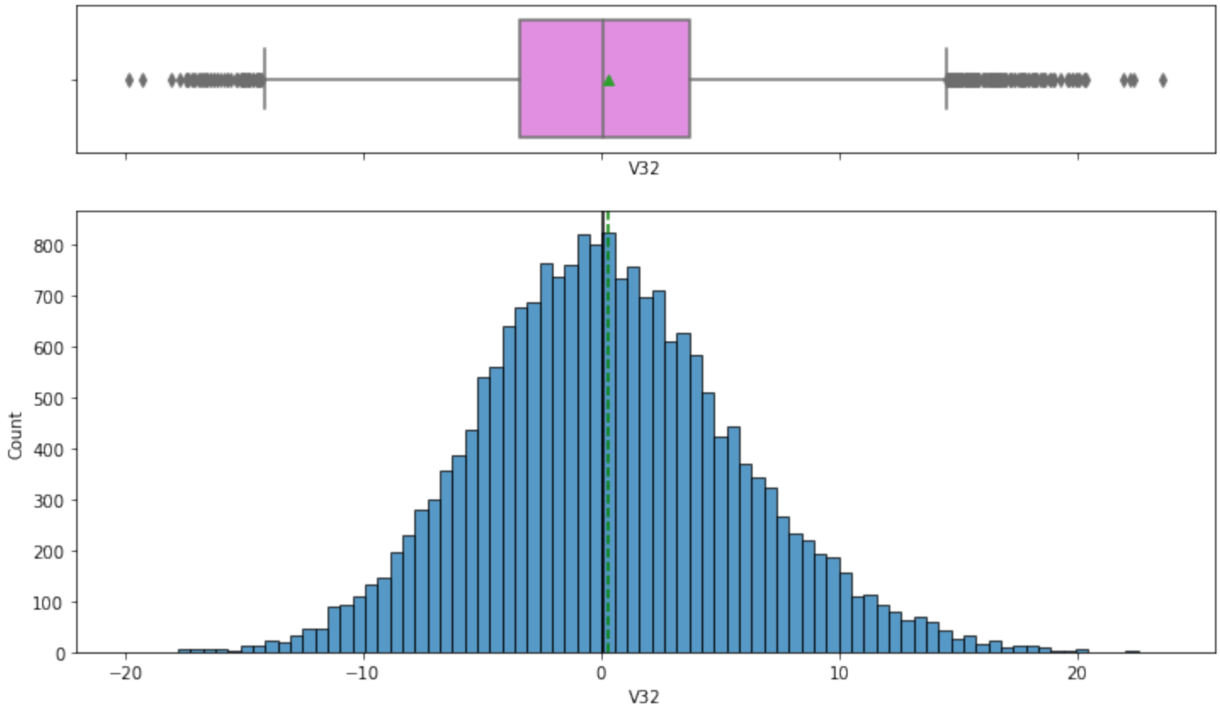

V27
Count
V27

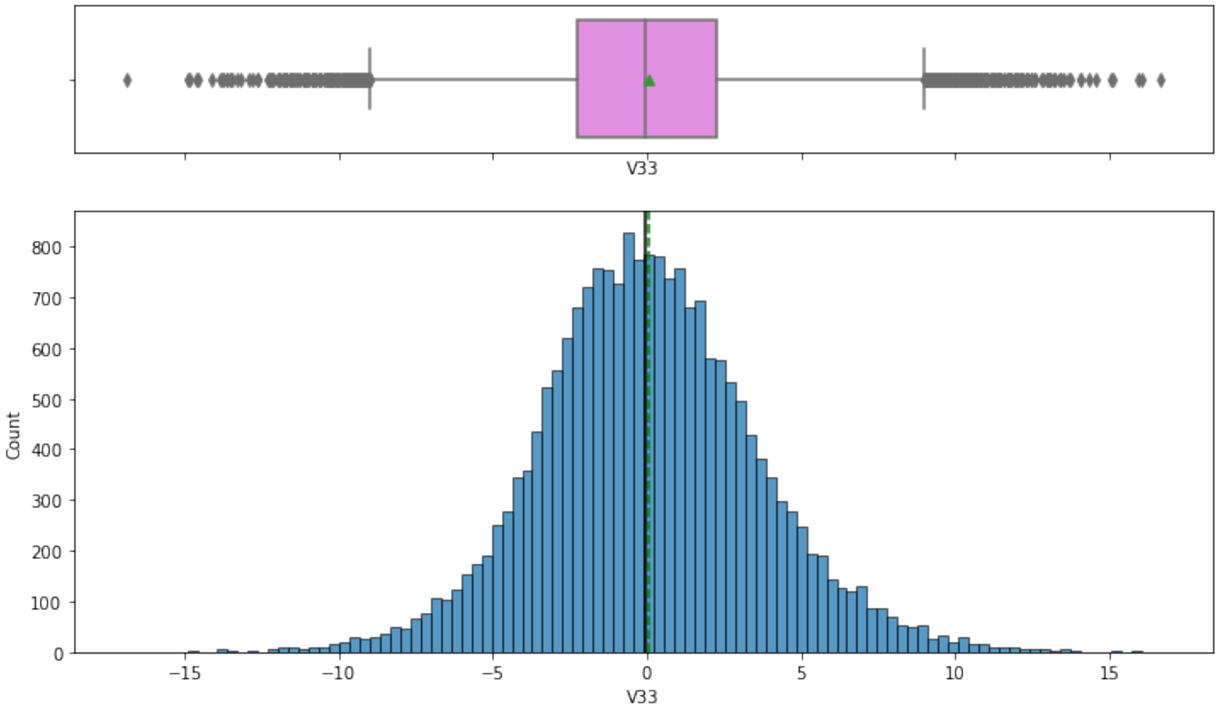

V28
Count
V28

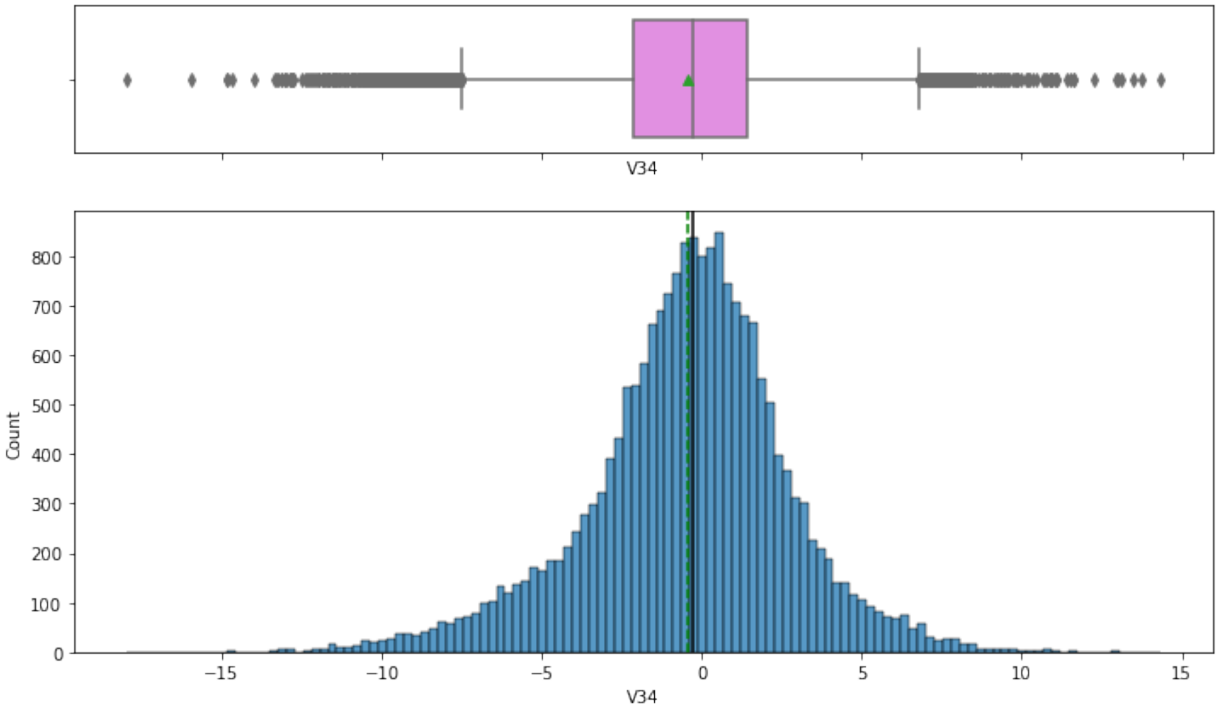

V29
Count
V29

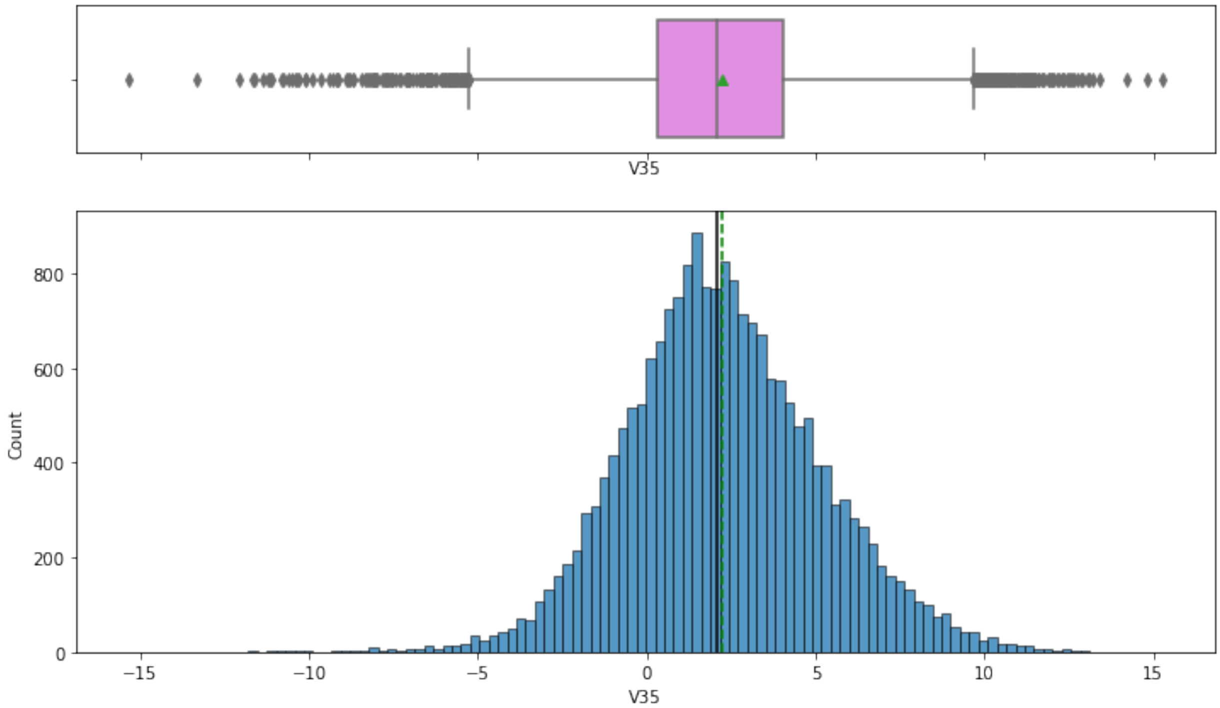

V30
Count
V30

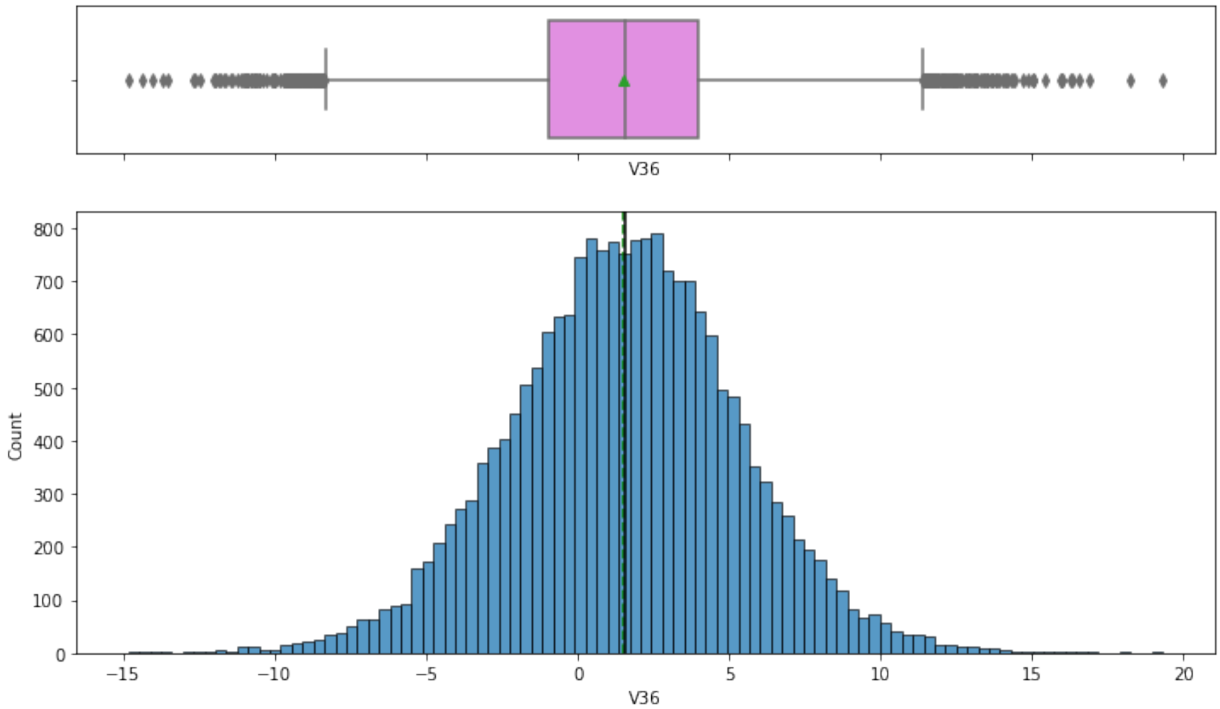

V31
Count
800
700
600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
V31

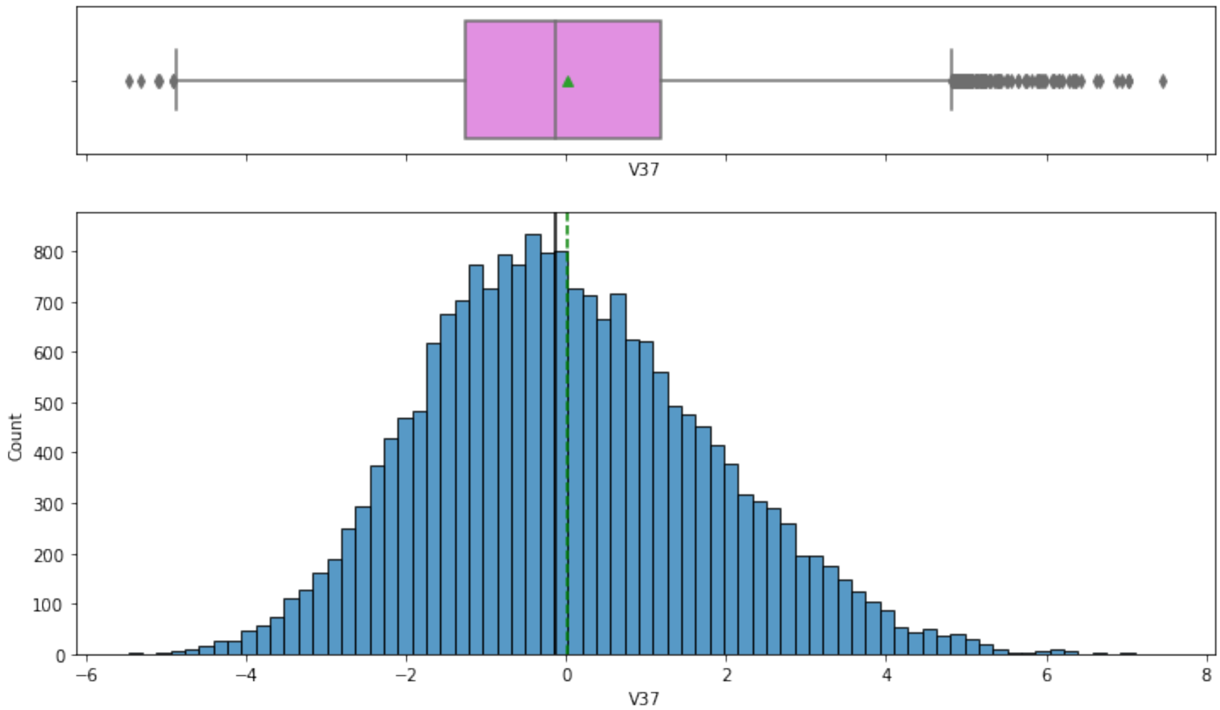

V32
Count
800
700
600
500
400
300
200
100
0
−20
−10
0
10
20
V32

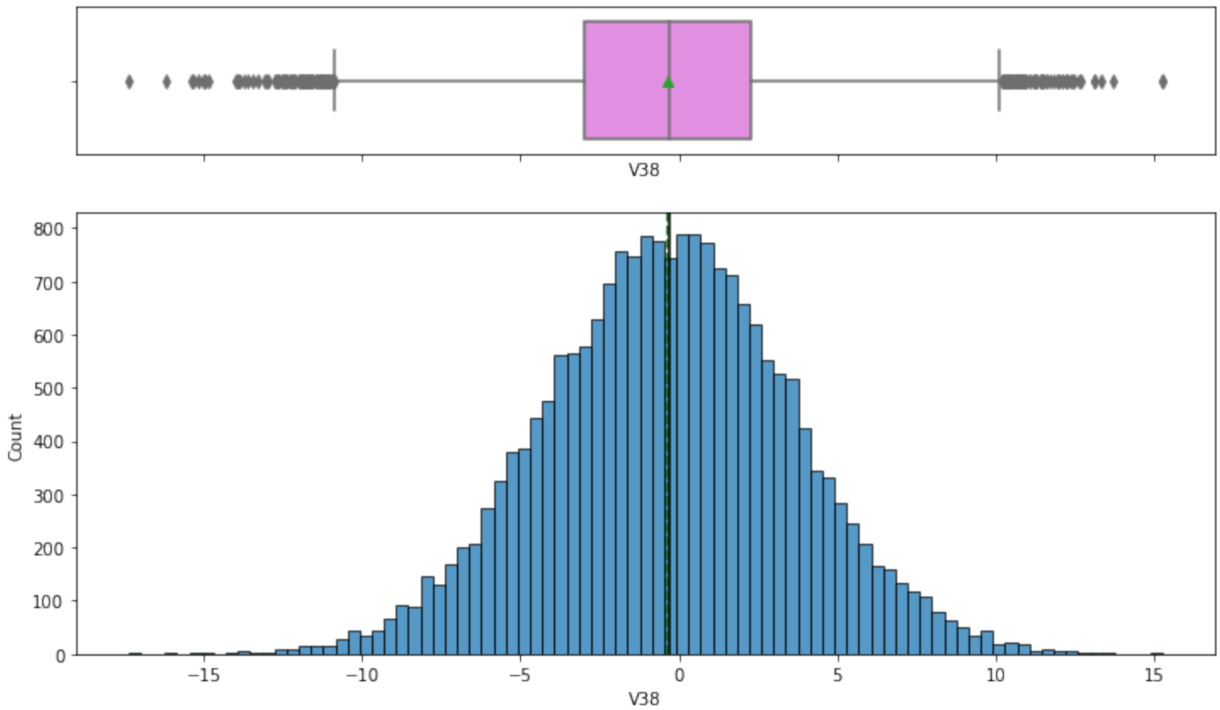

V33
Count
0 100 200 300 400 500 600 700 800
−15 −10 −5 0 5 10 15
V33

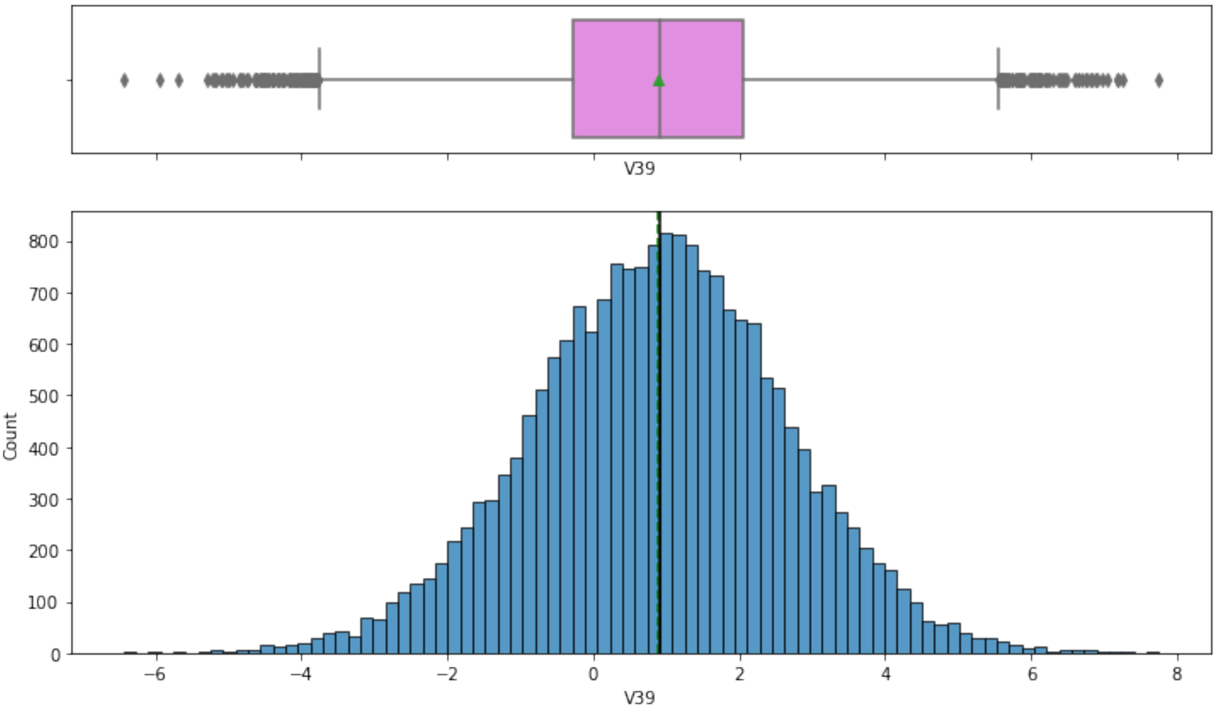

V34
Count
0 100 200 300 400 500 600 700 800
−15 −10 −5 0 5 10 15
V34

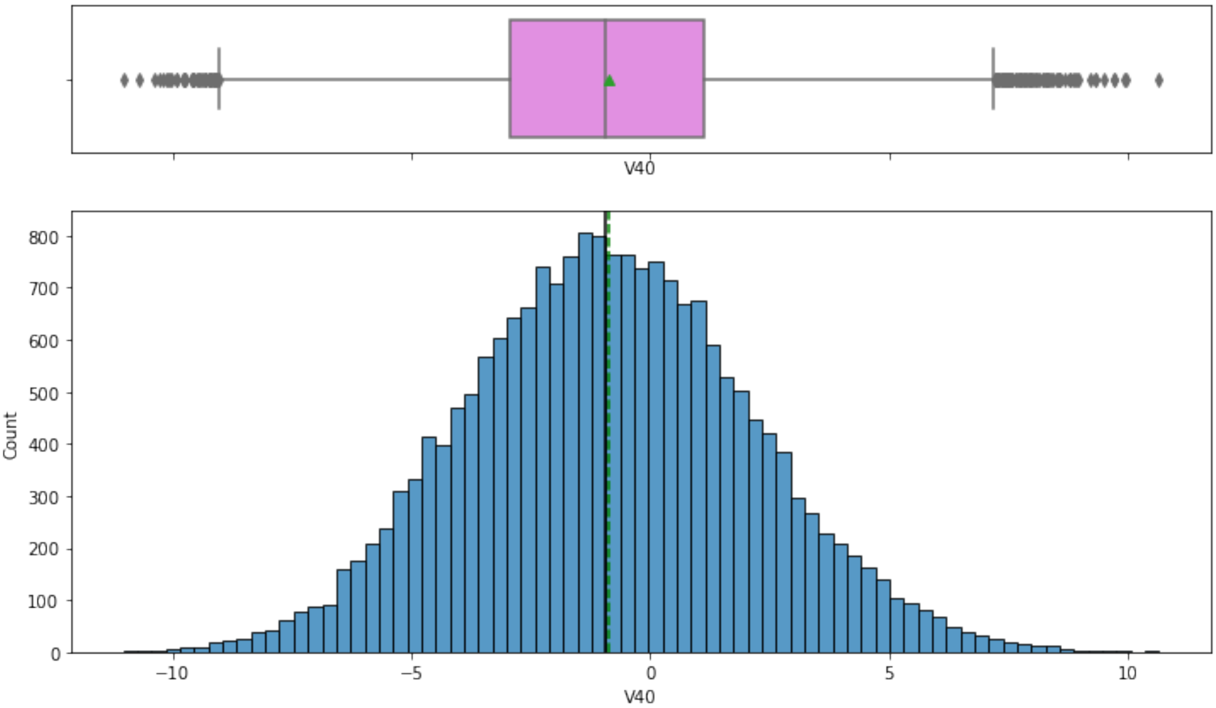

V35
Count
0
200
400
600
800
−15
−10
−5
0
5
10
15
V35

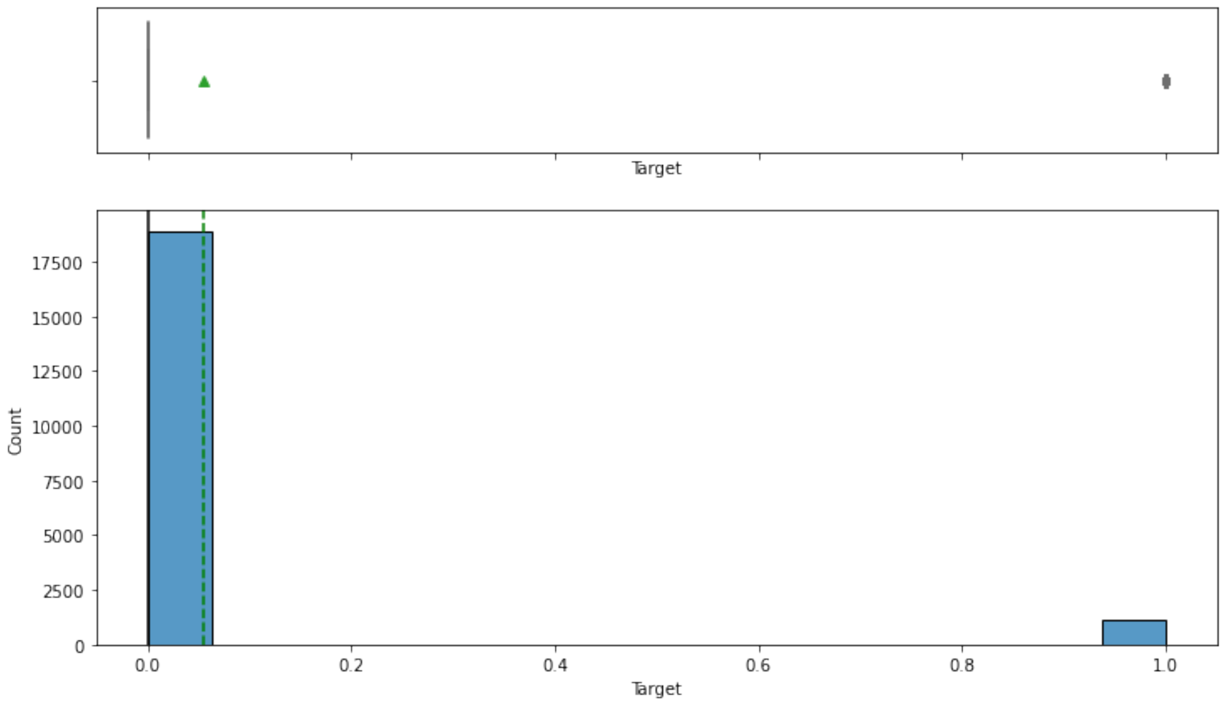

V36
Count
0
100
200
300
400
500
600
700
800
−15
−10
−5
0
5
10
15
20
V36


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


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


V39
Count
V39


V40
Count
V40




## Let's look at the values in target variable

In [97]:
```
data["Target"].value_counts() ##  Complete the code to check the class di
```

Out[97]:
```
0    18890
1     1110
Name: Target, dtype: int64
```

In [98]:
```
data_test["Target"].value_counts() ##  Complete the code to check the cla
```

Out[98]:
```
0    4718
1     282
Name: Target, dtype: int64
```

# Data Pre-Processing

In [99]:
```
# Dividing train data into X and y
X = data.drop(["Target"], axis=1)
y = data["Target"]
```

**Since we already have a separate test set, we don't need to divide data into train, valiation and test**

In [100…
```
# Splitting train dataset into training and validation set

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.25, r
```

In [101…
```
# Checking the number of rows and columns in the X_train data
X_train.shape ##  Complete the code to view dimensions of the X_train dat

# Checking the number of rows and columns in the X_val data
X_val.shape ##  Complete the code to view dimensions of the X_val data
```

Out[101]:
```
(5000, 40)
```

In [102…
```
# Dividing test data into X_test and y_test

X_test = data_test.drop(["Target"], axis=1) ##  Complete the code to drop
y_test = data_test['Target'] ##  Complete the code to store target variab
```

In [103…
```
# Checking the number of rows and columns in the X_test data
X_test.shape##  Complete the code to view dimensions of the X_test data
```

Out[103]:
```
(5000, 40)
```

# Missing value imputation

In [104…
```
# creating an instace of the imputer to be used
imputer = SimpleImputer(strategy="median")
```

In [105…
```
# Fit and transform the train data
X_train = pd.DataFrame(imputer.fit_transform(X_train), columns=X_train.co

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

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

In [106…
```
# Checking that no column has missing values in train or test sets
print(X_train.isna().sum())
print("-" * 30)

print(X_val.isna().sum())
print("-" * 30)

print(X_test.isna().sum())
print("-" * 30)
```

```
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 [107…

```
# defining a function to compute different metrics to check performance o
def model_performance_classification_sklearn(model, predictors, target):
    """
    Function to compute different metrics to check classification model p

    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.

In [108…

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

**We are now done with pre-processing and evaluation criterion, so let's start building the model.**

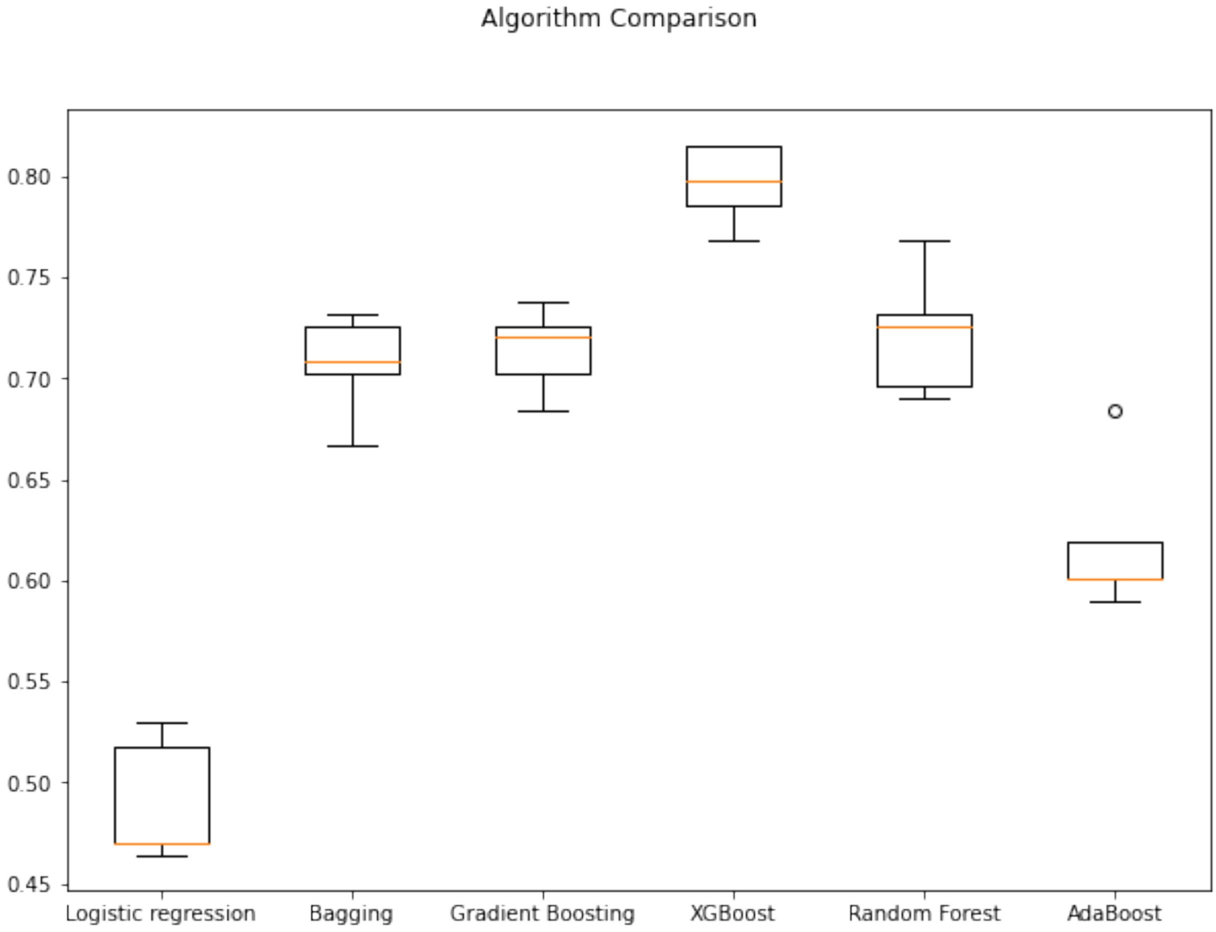
## Model Building on original data

In [109…

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

# Appending models into the list
models.append(("Logistic regression", LogisticRegression(random_state=1))
models.append(("Bagging", BaggingClassifier(random_state=1)))
models.append(("Gradient Boosting", GradientBoostingClassifier(random_sta
models.append(("XGBoost", XGBClassifier(random_state=1)))
models.append(("Random Forest", RandomForestClassifier(random_state=1)))
models.append(("AdaBoost", AdaBoostClassifier(random_state=1))) ## Comple

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:

Logistic regression: 0.4904761904761905
Bagging: 0.7071428571428572
Gradient Boosting: 0.7142857142857142
XGBoost: 0.7964285714285715
Random Forest: 0.7226190476190476
AdaBoost: 0.6190476190476192

Validation Performance:

Logistic regression: 0.48148148148148145
Bagging: 0.7222222222222222
Gradient Boosting: 0.6888888888888889
XGBoost: 0.7925925925925926
Random Forest: 0.6962962962962963
AdaBoost: 0.5777777777777777
```

In [110…
```
# 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 with oversampled data

In [111…

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

# 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("After OverSampling, counts of label '1': {}".format(sum(y_train_ov
print("After OverSampling, counts of label '0': {} \n".format(sum(y_train_


print("After OverSampling, the shape of train_X: {}".format(X_train_over.
print("After OverSampling, the shape of train_y: {} \n".format(y_train_ov
```

```
Before OverSampling, counts of label '1': 840
Before OverSampling, counts of label '0': 14160

After OverSampling, counts of label '1': 14160
After OverSampling, counts of label '0': 14160

After OverSampling, the shape of train_X: (28320, 40)
After OverSampling, the shape of train_y: (28320,)
```

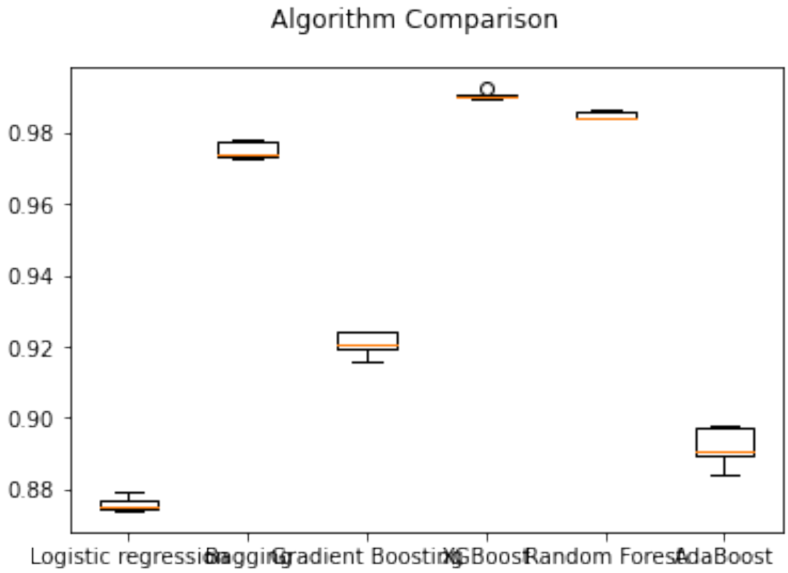
In [112…

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

# Appending models into the list
models.append(("Logistic regression", LogisticRegression(random_state=1))
models.append(("Bagging", BaggingClassifier(random_state=1)))
models.append(("Gradient Boosting", GradientBoostingClassifier(random_sta
models.append(("XGBoost", XGBClassifier(random_state=1)))
models.append(("Random Forest", RandomForestClassifier(random_state=1)))
models.append(("AdaBoost", AdaBoostClassifier(random_state=1)))
 ## Complete the code to append remaining 4 models in the list models

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,
    )  ## Complete the code to build models on oversampled data
    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)## Complete the code to build mo
    scores = recall_score(y_val, model.predict(X_val))
    print("{}: {}".format(name, scores))
```

```
Cross-Validation performance on training dataset:

Logistic regression: 0.8759180790960451
Bagging: 0.975
Gradient Boosting: 0.9206920903954803
XGBoost: 0.9903954802259886
Random Forest: 0.9848870056497174
AdaBoost: 0.8918079096045199

Validation Performance:

Logistic regression: 0.8518518518518519
Bagging: 0.8148148148148148
Gradient Boosting: 0.8629629629629629
XGBoost: 0.8666666666666667
Random Forest: 0.8407407407407408
AdaBoost: 0.8555555555555555
```

In [113…

```
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results1)
ax.set_xticklabels(names)
plt.show()
```




## Model Building with undersampled data

In [114…

```
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
print("Before UnderSampling, counts of label '0': {} \n".format(sum(y_tra


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


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

```
Before UnderSampling, counts of label '1': 840
Before UnderSampling, counts of label '0': 14160

After UnderSampling, counts of label '1': 840
After UnderSampling, counts of label '0': 840

After UnderSampling, the shape of train_X: (1680, 40)
After UnderSampling, the shape of train_y: (1680,)
```

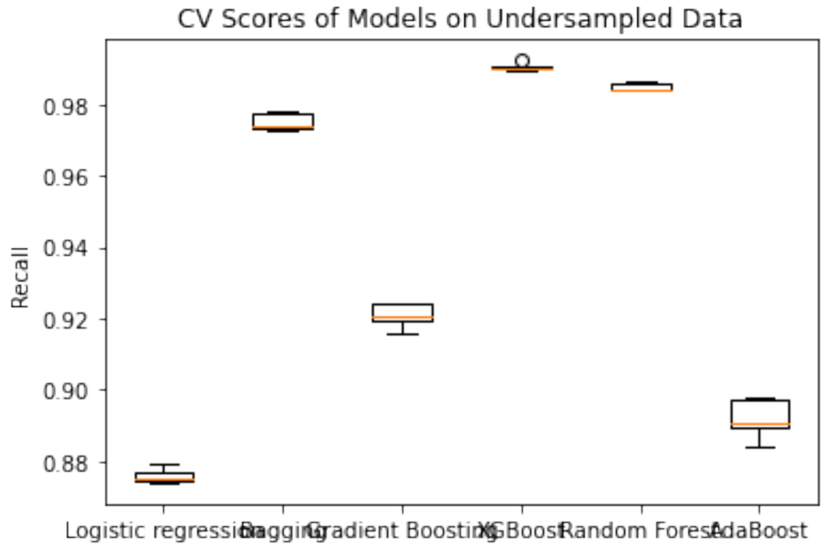
In [115…

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

# Appending models into the list
models.append(("Logistic regression", LogisticRegression(random_state=1))
models.append(("Bagging", BaggingClassifier(random_state=1)))
models.append(("Gradient Boosting", GradientBoostingClassifier(random_sta
models.append(("XGBoost", XGBClassifier(random_state=1)))
models.append(("Random Forest", RandomForestClassifier(random_state=1)))
models.append(("AdaBoost", AdaBoostClassifier(random_state=1))) ## Comple

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,
    )  ## Complete the code to build models on undersampled data
    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=X_train_over, y=y_train_over)## Complete the code to buil
    scores = recall_score(y_val, model.predict(X_val))
    print("{}: {}".format(name, scores))
```

```
Cross-Validation performance on training dataset:

Logistic regression: 0.8759180790960451
Bagging: 0.975
Gradient Boosting: 0.9206920903954803
XGBoost: 0.9903954802259886
Random Forest: 0.9848870056497174
AdaBoost: 0.8918079096045199

Validation Performance:

Logistic regression: 0.8518518518518519
Bagging: 0.8148148148148148
Gradient Boosting: 0.8629629629629629
XGBoost: 0.8666666666666667
Random Forest: 0.8407407407407408
AdaBoost: 0.8555555555555555
```

In [116…

```
plt.boxplot(results1, labels=names)
plt.title("CV Scores of Models on Undersampled Data")
plt.ylabel("Recall")
plt.show()
```




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

**Note**: You can choose to tune some other model if XGBoost gives error.

# Hyperparameter Tuning

## Note

1. Sample parameter grid has been provided to do necessary hyperparameter tuning. One can extend/reduce the parameter grid based on execution time and system configuration to try to improve the model performance further wherever needed.
2. The models chosen in this notebook are based on test runs. One can update the best models as obtained upon code execution and tune them for best performance.

### Tuning AdaBoost using oversampled data

In [117…

```
%%time

# 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
    ]
}


#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=p

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over,y_train_over) ## Complete the code to fit

print("Best parameters are {} with CV score={}:" .format(randomized_cv.be
```

```
Best parameters are {'n_estimators': 200, 'learning_rate': 0.2, 'base_est
imator': DecisionTreeClassifier(max_depth=3, random_state=1)} with CV sco
re=0.9715395480225988:
CPU times: user 1min 1s, sys: 378 ms, total: 1min 1s
Wall time: 5min 56s
```

In [118…

```
# Creating new pipeline with best parameters
tuned_ada = AdaBoostClassifier(
    n_estimators= 150, learning_rate= 0.05, base_estimator= DecisionTreeC
) ## Complete the code with the best parameters obtained from tuning

tuned_ada.fit(X_train_over, y_train_over) ## Complete the code to fit the
```

Out[118]:

```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=2,
                                                         random_state=1)
,
                   learning_rate=0.05, n_estimators=150)
```

In [119…

```
ada_train_perf = model_performance_classification_sklearn(tuned_ada, X_tr
ada_train_perf
```

Out[119]:

| | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|
| **0** | 0.926 | 0.901 | 0.948 | 0.924 |

In [120…

```
ada_val_perf = recall_score(y_val, tuned_ada.predict(X_val))
ada_val_perf
```

Out[120]:

```
0.8555555555555555
```

## Tuning Random forest using undersampled data

In [123…

```
%%time

# 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=p

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_un, y_train_un) ## Complete the code to fit the

print("Best parameters are {} with CV score={}:" .format(randomized_cv.be
```

```
Best parameters are {'n_estimators': 250, 'min_samples_leaf': 1, 'max_sam
ples': 0.6, 'max_features': 'sqrt'} with CV score=0.8964285714285714:
CPU times: user 1.96 s, sys: 248 ms, total: 2.21 s
Wall time: 15.1 s
```

In [125…

```
# Creating new pipeline with best parameters
tuned_rf2 = RandomForestClassifier(
    max_features=randomized_cv.best_params_['max_features'],
    random_state=1,
    max_samples=randomized_cv.best_params_['max_samples'],
    n_estimators=randomized_cv.best_params_['n_estimators'],
    min_samples_leaf=randomized_cv.best_params_['min_samples_leaf'],
)

tuned_rf2.fit(X_train_un, y_train_un)
```

Out[125]:

```
RandomForestClassifier(max_features='sqrt', max_samples=0.6, n_estimator
s=250,
                       random_state=1)
```

In [127…

```
rf2_train_perf = model_performance_classification_sklearn(tuned_rf2, X_tr
rf2_train_perf
```

Out[127]:

| | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|
| **0** | 0.990 | 0.980 | 1.000 | 0.990 |

In [129…

```
rf2_val_perf = model_performance_classification_sklearn(tuned_rf2, X_val,
rf2_val_perf
```

Out[129]:

| | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|
| **0** | 0.942 | 0.874 | 0.478 | 0.618 |

## Tuning Gradient Boosting using oversampled data

In [130…

```
%%time

# 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,

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=p

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

print("Best parameters are {} with CV score={}:" .format(randomized_cv.be
```

```
Best parameters are {'subsample': 0.7, 'n_estimators': 125, 'max_features
': 0.5, 'learning_rate': 1} with CV score=0.9709039548022599:
CPU times: user 12.4 s, sys: 205 ms, total: 12.6 s
Wall time: 2min 28s
```

In [131…

```
tuned_gbm = GradientBoostingClassifier(
max_features=randomized_cv.best_params_['max_features'],
random_state=1,
learning_rate=randomized_cv.best_params_['learning_rate'],
n_estimators=randomized_cv.best_params_['n_estimators'],
subsample=randomized_cv.best_params_['subsample'],
)

tuned_gbm.fit(X_train_over, y_train_over)
```

Out[131]:

```
GradientBoostingClassifier(learning_rate=1, max_features=0.5, n_estimato
rs=125,
                           random_state=1, subsample=0.7)
```

In [132…

```
gbm_train_perf = model_performance_classification_sklearn(tuned_gbm, X_tr
gbm_train_perf
```

Out[132]:

| | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|
| **0** | 0.993 | 0.993 | 0.993 | 0.993 |

In [133…

```
gbm_val_perf = model_performance_classification_sklearn(tuned_gbm, X_val,
gbm_val_perf
```

Out[133]:

| | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|
| **0** | 0.965 | 0.844 | 0.630 | 0.722 |

## Tuning XGBoost using oversampled data

**Note**: You can choose to skip this section if XGBoost gives error.

In [152…

```
%%time

# defining model
Model = XGBClassifier(random_state=1,eval_metric='logloss')

#Parameter grid to pass in RandomSearchCV
param_grid={'n_estimators':[150,200,250],'scale_pos_weight':[5,10], 'lear

#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=Model, param_distributions=p

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over, y_train_over)
 ## Complete the code to fit the model on over sampled data

print("Best parameters are {} with CV score={}:" .format(randomized_cv.be
```

```
Best parameters are {'subsample': 0.8, 'scale_pos_weight': 10, 'n_estimat
ors': 250, 'learning_rate': 0.1, 'gamma': 3} with CV score=0.996610169491
5255:
CPU times: user 2min 19s, sys: 2.18 s, total: 2min 21s
Wall time: 51min 42s
```

In [153…

```
xgb2 = XGBClassifier(
    random_state=1,
    eval_metric="logloss",
    subsample=0.8,
    scale_pos_weight=10,
    n_estimators=250,
    learning_rate=0.2,
    gamma=3,
)

xgb2.fit(X_train_over, y_train_over)
```

Out[153]:

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=3, gpu_id=None, grow_policy=None
,
              importance_type=None, interaction_constraints=None,
              learning_rate=0.2, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=Non
e,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, n_estimators=250, n_jobs=None,
              num_parallel_tree=None, predictor=None, random_state=1, ..
.)
```

In [154…

```
xgb2_train_perf = model_performance_classification_sklearn(xgb2, X_train_
xgb2_train_perf
```

Out[154]:

| | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|
| **0** | 1.000 | 1.000 | 0.999 | 1.000 |

In [155…

```
xgb2_val_perf = model_performance_classification_sklearn(xgb2, X_val, y_v
xgb2_val_perf
```

Out[155]:

| | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|
| **0** | 0.979 | 0.885 | 0.768 | 0.823 |

**We have now tuned all the models, let's compare the performance of all tuned models and see which one is the best.**

# Model performance comparison and choosing the final model

In [156…

```
# training performance comparison

models_train_comp_df = pd.concat(
    [
        gbm_train_perf.T,
        ada_train_perf.T,
        rf2_train_perf.T,
        xgb2_train_perf.T,
    ],
    axis=1,
)
models_train_comp_df.columns = [
    "Gradient Boosting tuned with oversampled data",
    "AdaBoost classifier tuned with oversampled data",
    "Random forest tuned with undersampled data",
    "XGBoost tuned with oversampled data"
]
print("Training performance comparison:")
models_train_comp_df
```

```
Training performance comparison:
```

Out[156]:

| | Gradient Boosting tuned with oversampled data | AdaBoost classifier tuned with oversampled data | Random forest tuned with undersampled data | XGBoost tuned with oversampled data |
|---|---|---|---|---|
| **Accuracy** | 0.993 | 0.926 | 0.990 | 1.000 |
| **Recall** | 0.993 | 0.901 | 0.980 | 1.000 |
| **Precision** | 0.993 | 0.948 | 1.000 | 0.999 |
| **F1** | 0.993 | 0.924 | 0.990 | 1.000 |

In [158…

```
model_names = ['Random Forest', 'Adaboost', 'Gradient Boosting','XGboost'
model_val_perfs = [rf2_val_perf, ada_val_perf, gbm_val_perf ,xgb2_val_per

val_perf_df = pd.DataFrame({'Model': model_names, 'Validation Performance
val_perf_df
```

Out[158]:

| | Model | Validation Performance |
|---|---|---|
| **0** | Random Forest | Accuracy Recall Precision F1 0 0.9... |
| **1** | Adaboost | 0.856 |
| **2** | Gradient Boosting | Accuracy Recall Precision F1 0 0.9... |
| **3** | XGboost | Accuracy Recall Precision F1 0 0.9... |

**Now we have our final model, so let's find out how our final model is performing on unseen test data.**

In [169…

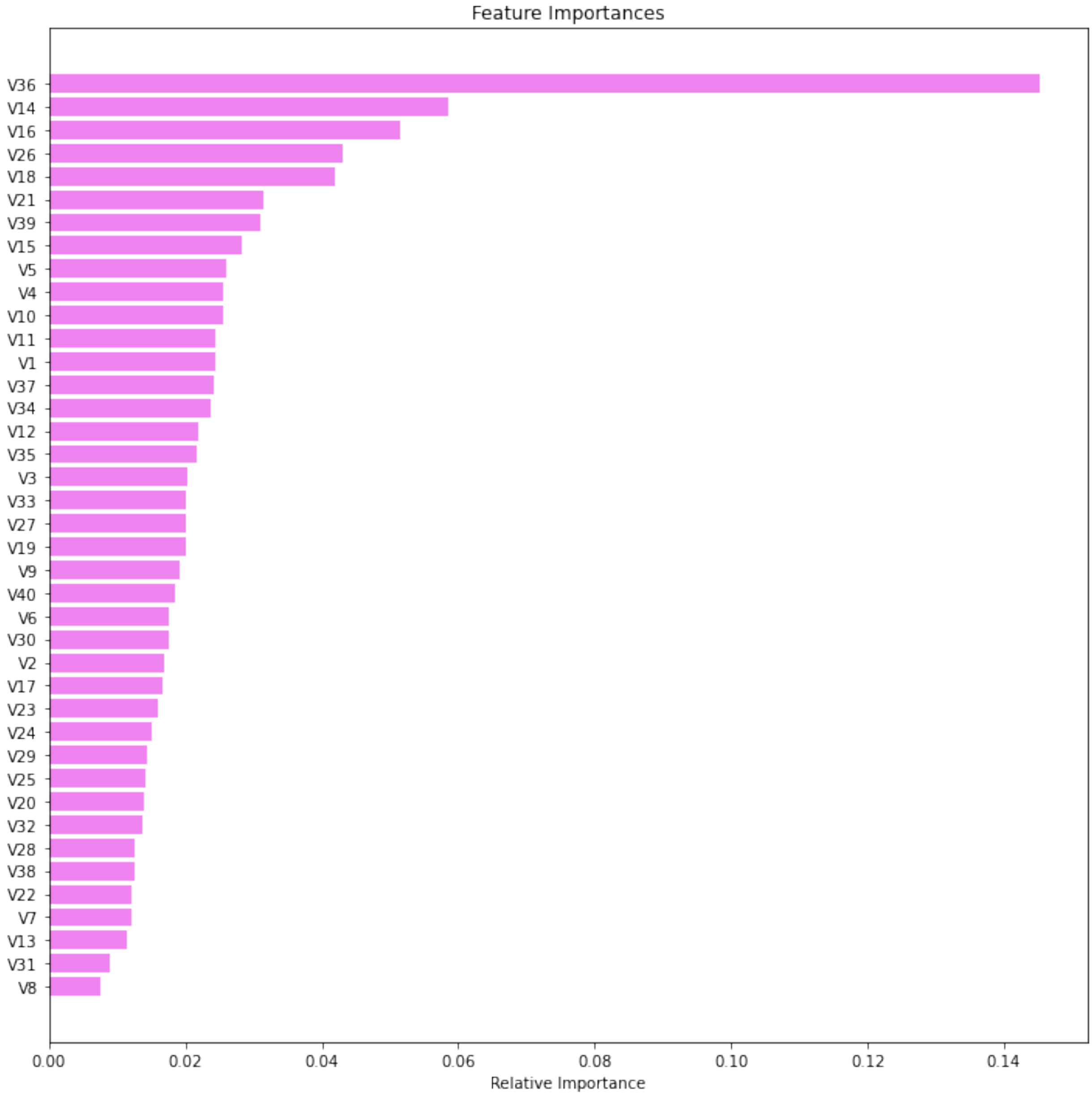
```
# training performance comparison

models_test_comp_df = pd.concat(
    [
        gbm_test_perf.T,
        ada_test_perf.T,
        rf2_test_perf.T,
        xgb2_test_perf.T,
    ],
    axis=1,
)
models_train_comp_df.columns = [
    "Gradient Boosting tuned with oversampled data",
    "AdaBoost classifier tuned with oversampled data",
    "Random forest tuned with undersampled data",
    "XGBoost tuned with oversampled data"
]
print("test performance comparison:")
models_test_comp_df
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [169], in <cell line: 3>()
      1 # training performance comparison
      3 models_test_comp_df = pd.concat(
      4     [
----> 5         gbm_test_perf.T,
      6         ada_test_perf.T,
      7         rf2_test_perf.T,
      8         xgb2_test_perf.T,
      9     ],
     10     axis=1,
     11 )
     12 models_train_comp_df.columns = [
     13     "Gradient Boosting tuned with oversampled data",
     14     "AdaBoost classifier tuned with oversampled data",
     15     "Random forest tuned with undersampled data",
     16     "XGBoost tuned with oversampled data"
     17 ]
     18 print("test performance comparison:")

NameError: name 'gbm_test_perf' is not defined
```

## Feature Importances

In [161…

```
feature_names = X_train.columns
importances = xgb2.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
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel("Relative Importance")
plt.show()
```




# Let's use Pipelines to build the final model

- Since we have only one datatype in the data, we don't need to use column transformer here

In [162…

```
Pipeline_model = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler()),
    ('classifier', XGBClassifier(random_state=1,eval_metric='logloss',
                                 subsample=0.8, scale_pos_weight=10,
                                 n_estimators=250, learning_rate=0.2, gam
])
```

In [163...
```
# Separating target variable and other variables
X1 = data.drop(columns="Target")
Y1 = data["Target"]

# Since we already have a separate test set, we don't need to divide data

X_test1 = df_test.drop(columns='Target')
y_test1 = df_test['Target']
 ##  Complete the code to store target variable in y_test1
```

In [164...
```
# We can't oversample/undersample data without doing missing value treatm
imputer = SimpleImputer(strategy="median")
X1 = imputer.fit_transform(X1)

# We don't need to impute missing values in test set as it will be done i
```

**Note:** Please perform either oversampling or undersampling based on the final model chosen.

If the best model is built on the oversampled data, uncomment and run the below code to perform oversampling

In [165...
```
#code for oversampling on the data
# Synthetic Minority Over Sampling Technique
sm = SMOTE(sampling_strategy=1, k_neighbors=5, random_state=1)
X_over1, y_over1 = sm.fit_resample(X1, Y1)
```

If the best model is built on the undersampled data, uncomment and run the below code to perform undersampling

In [166...
```
# # code for undersampling on the data
# # Under Sampling Technique
rus = RandomUnderSampler(random_state=1, sampling_strategy=1)
X_train_un, y_train_un = rus.fit_resample(X_train, y_train)
```

In [167...
```
Pipeline_model.fit(X_over1, y_over1)
```

Out[167]:
```
Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
                ('scaler', StandardScaler()),
                ('classifier',
                 XGBClassifier(base_score=None, booster=None, callbacks=
None,
                               colsample_bylevel=None, colsample_bynode=
None,
                               colsample_bytree=None,
                               early_stopping_rounds=None,
                               enable_categorical=False, eval_metric='lo
gloss',
                               feature_types=None, gamma=3, gpu_id=None,
                               grow_policy=None, importance_type=None,
                               interaction_constraints=None, learning_ra
te=0.2,
                               max_bin=None, max_cat_threshold=None,
                               max_cat_to_onehot=None, max_delta_step=No
ne,
                               max_depth=None, max_leaves=None,
                               min_child_weight=None, missing=nan,
                               monotone_constraints=None, n_estimators=2
50,
                               n_jobs=None, num_parallel_tree=None,
                               predictor=None, random_state=1, ...))])
```

In [168…
```
Pipeline_model_test = Pipeline_model.score(X_test1, y_test1)
Pipeline_model_test
```

Out[168]:
```
0.9772
```

# Business Insights and Conclusions

- Best model and its performance
- Important features
- Additional points

---