ReneWind

Model tuning

Date 4/14/2023

Elena Korzilova

Contents / Agenda

Executive Summary

Actionable Insights & Recommendations:

- Feature Importance: Utilize the feature importances extracted from the XGBoost model to identify the most influential factors driving the predictions. Focus on leveraging these key features for further analysis and decision-making.
- Oversampling Strategy: The Synthetic Minority Over Sampling Technique (SMOTE) was employed to address class imbalance. Continuously monitor the effectiveness of this strategy and consider experimenting with different oversampling techniques to enhance model performance further.
- Hyperparameter Tuning: Fine-tuning model hyperparameters significantly contributed to enhancing predictive accuracy. Continue to explore hyperparameter optimization techniques to extract the maximum performance from the model.
- Model Maintenance: Regularly monitor the model's performance and retrain it as needed with new data to ensure its effectiveness over time. Stay vigilant for changes in the data distribution or underlying patterns that may impact model performance.
- Validation Performance: While the model performed admirably on the training data, there was a slight drop in performance on the validation set. Investigate potential causes for this discrepancy and consider refining the model or data preprocessing steps to address any underlying issues.

Business Problem Overview and Solution Approach

- Business Problem:

  Renewable energy, particularly wind energy, is pivotal in the global energy landscape due to environmental concerns. To optimize wind turbine efficiency, predictive maintenance is essential. However, the challenge lies in accurately predicting component failures before they occur, to minimize maintenance costs.

- Solution Approach:

  ReneWind aims to enhance wind energy production through machine learning. By leveraging sensor data, predictive maintenance models will be developed to anticipate turbine failures. The objective is to classify failures accurately, distinguishing between true positives, false negatives, and false positives. This approach enables proactive maintenance, reducing repair costs, and optimizing turbine performance.

EDA Results

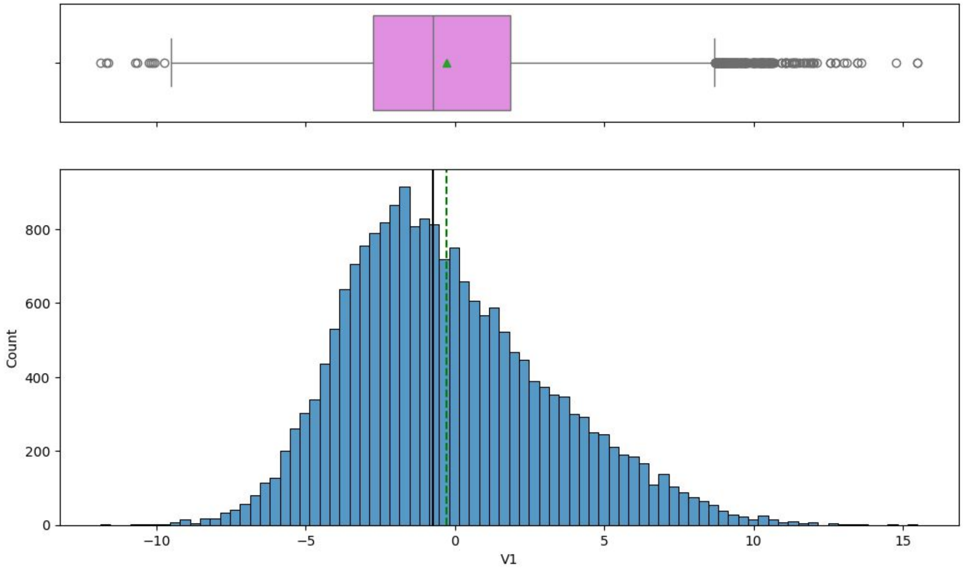
The statistical summary of the training data reveals key insights:

- Data Completeness: Most variables are fully populated with 20,000 entries each, except for V1 and V2 which each have 18 missing values.
- Distribution: The data shows a broad range of means and standard deviations, indicating varied scales and significant dispersion among the features.
- Extremes: Minimum and maximum values suggest some features have extreme outliers.
- Target Variable: The dataset is imbalanced with only 5.6% of cases indicating 'failure' (1).

Link to Appendix slide on data background check

EDA Results

- Histogram:
  - The distribution of V1 appears to be approximately normal, centered around zero.
  - There is a visible skew to the right, indicated by a longer tail extending towards the positive values.
- Boxplot:
  - The boxplot on the top indicates a relatively symmetrical spread around the median, which is marked by the green triangle.
  - There are numerous outliers on both sides, but more pronounced on the positive side of the scale.
  - The interquartile range (IQR) is compact, suggesting that the middle 50% of the data points are clustered within a small range of values.

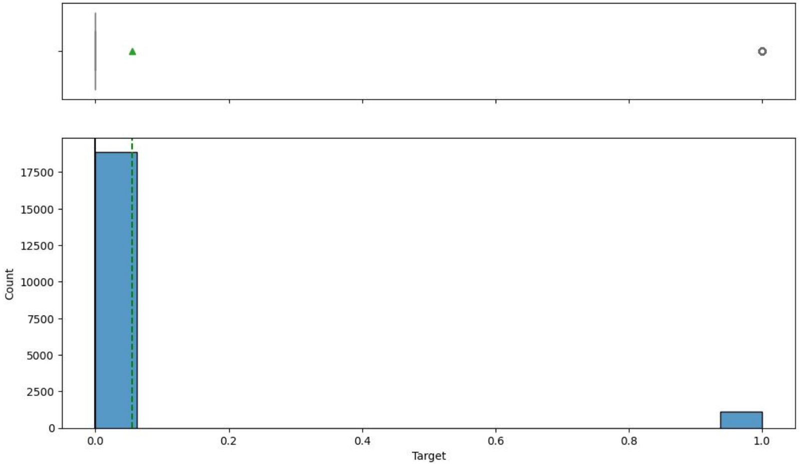


Similar results for the rest variables. **More EDA results in the attachment**

Slide 17: EDA Results

EDA Results

- Histogram Observations:
  - The data is highly imbalanced with a vast majority of the observations categorized as '0' (no failure).
  - There's a very small number of observations categorized as '1' (failure).
- Boxplot Observations:
  - The boxplot shows that almost all data falls into the '0' category with a few outliers represented as '1'.
  - The median is at '0', indicating that more than half of the observations are non-failures.
  - The data points for '1' appear as outliers, emphasizing the imbalance.



Link to Appendix slide on data background check

Data Preprocessing

- Duplicate value check
- Missing value treatment
- Outlier check (treatment if needed)
- Feature engineering
- Data preparation for modeling

Data Preprocessing

For the training data:

- Class 0 (non-failure) has 18,890 observations.
- Class 1 (failure) has 1,110 observations.

For the test data:

- Class 0 (non-failure) has 4,718 observations.
- Class 1 (failure) has 282 observations.

Missing values: For the test data:

- There are 5 missing values in column V1.
- There are 6 missing values in column V2.

For the training data:

- There are 18 missing values in column V1.
- There are 18 missing values in column V2.

Since we already have a separate test set, we don't need to divide data into train, validation and test.
We split the train dataset into train and validation set in the ratio 75:25

- `X_train` has 15,000 rows and 40 columns.
- `X_val` has 5,000 rows and 40 columns.

Link to Appendix slide on data background check

Data Preprocessing

Train-Validation Split: we split the training data into training and validation sets with a ratio of 75:25. This was done using the `train_test_split()` function, resulting in `X_train` and `X_val` datasets with dimensions `(15000, 40)` and `(5000, 40)` respectively.

Test Data Preparation: we divided the test data into features (`X_test`) and the target variable (`y_test`).

The features were obtained by dropping the target variable from the test dataset, resulting in `X_test` with dimensions `(5000, 40)`. You stored the target variable in `y_test`.

We utilized the `SimpleImputer` from scikit-learn to impute missing values with the median.

Applied the imputer to both the training, validation, and test datasets.

Ensured that there are no missing values present in any of the datasets by checking the count of missing values in each column.

Data Preprocessing

Based on your requirement to maximize the correct prediction of generator failures while minimizing false negatives, the metric to optimize is Recall.

Recall measures the ability of a model to correctly identify all positive instances out of all actual positive instances (i.e., true positives divided by the sum of true positives and false negatives). Maximizing recall ensures that the model identifies the maximum number of generator failures correctly, reducing the chances of false negatives

We set up the scoring metric to be used for cross-validation and hyperparameter tuning as Recall. This means that during the process of model evaluation and parameter optimization, the algorithms will prioritize maximizing the **recall score**, which aligns with your objective of reducing false negatives.

Link to Appendix slide on model assumptions

Model Performance Summary

- Summary of performance metrics for training and validation data in tabular format for comparison for tuned models
- Comments on the model performances and choice of final model

***Note**: You can use more than one slide if needed*

[Link to Appendix slide on model assumptions]

Model Performance Summary

Cross-Validation Performance on Training Dataset:

The mean cross-validated Recall scores for each model during the training phase.

- Among the models tested, XGBoost achieved the highest cross-validated Recall score of approximately 0.8012, indicating that it performed the best on average during cross-validation.

Validation Performance:

- Similar to the cross-validation results, XGBoost achieved the highest Recall score of approximately 0.8 on the validation dataset, indicating its strong performance in predicting generator failures.

In summary, both the cross-validation and validation results suggest that XGBoost is the top-performing model among those tested, as it consistently achieved the highest Recall scores on both the training and validation datasets.

Cross-Validation performance on training dataset	
Logistic regression	0.490476
Bagging	0.707143
Random Forest	0.722619
Gradient Boosting	0.714286
AdaBoost	0.619048
XGBoost	0.80119
Validation Performance	
Logistic regression	0.481481
Bagging	0.722222
Random Forest	0.696296
Gradient Boosting	0.688889
AdaBoost	0.577778
XGBoost	0.8

Link to Appendix slide on model assumptions

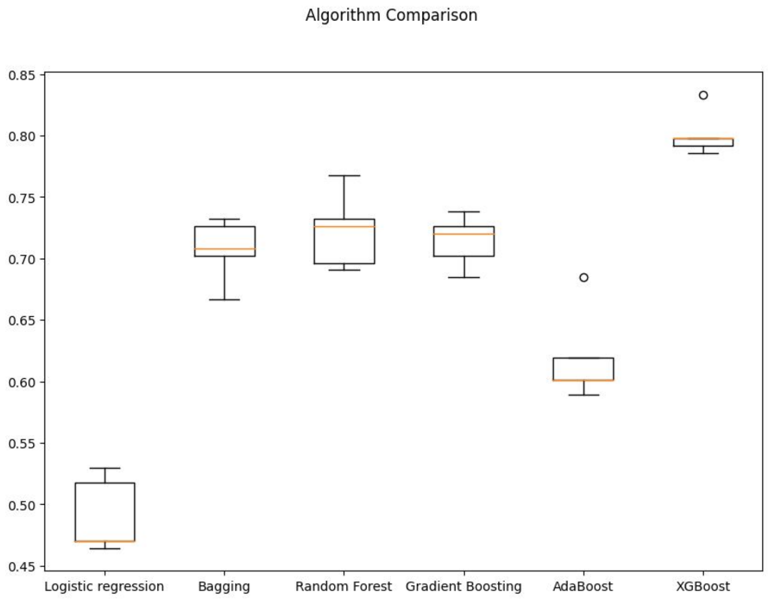
Model Performance Summary

Logistic Regression has the lowest performance with median CV score around 0.50, indicating it may not handle complex patterns in the data well.

Bagging, Random Forest, Gradient Boosting have median CV scores in a tighter range, roughly between 0.70 and 0.75, indicating good performance but with some variability.

AdaBoost has a lower median CV score, closer to 0.60, showing it might not be as effective as other ensemble methods here.

XGBoost stands out with a median CV score near 0.80 and less variability, suggesting it is the most accurate and stable model for the data.



Link to Appendix slide on model assumptions

Model Performance Summary

Model Building with oversampled data

Before OverSampling:

- Initially had 840 instances of label '1' (failures) and 14,160 instances of label '0' (non-failures) in your training data.
- The class distribution was highly imbalanced, with significantly fewer instances of failures compared to non-failures.

After OverSampling:

- SMOTE was applied to synthetically generate new instances of the minority class (label '1') to balance the class distribution.
- After oversampling, both classes now have the same number of instances, with 14,160 instances each.
- The shape of the training data (`X_train_over`) expanded to (28320, 40), indicating 28,320 instances and 40 features after oversampling.
- The shape of the training labels (`y_train_over`) also expanded to (28320,), matching the number of instances in `X_train_over`.

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,)

Link to Appendix slide on model assumptions

Model Performance Summary

Cross-Validation Performance:

-Random Forest and XGBoost have the highest cross-validation performance on the training dataset, with scores of 0.984 and 0.991, respectively.

-Bagging, Gradient Boosting, and AdaBoost also demonstrate strong performance, scoring above 0.875.

-Logistic Regression has the lowest performance among the models, but still maintains a respectable score of 0.876.

Validation Performance:

-XGBoost achieves the highest recall score on the validation set, indicating its effectiveness in identifying positive cases (failures) while minimizing false negatives.

-Gradient Boosting, AdaBoost, and Logistic Regression also perform well on the validation set, with recall scores above 0.85.

-Bagging and Random Forest, while performing strongly in cross-validation, show slightly lower performance on the validation set compared to the boosting algorithms.

Overall, **XGBoost appears to be the top-performing** model based on both cross-validation and validation performance, followed closely by Random Forest and Gradient Boosting.

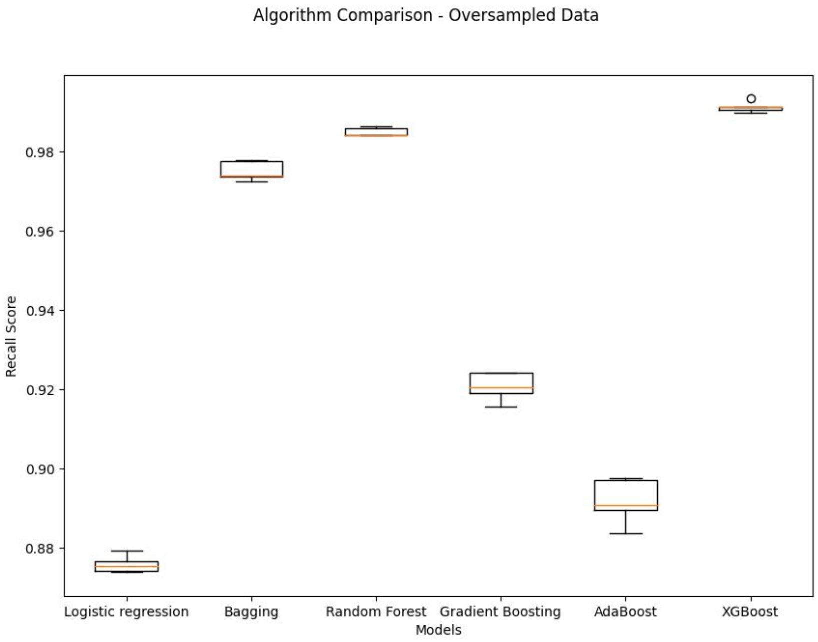
Cross-Validation performance on training dataset	
Logistic regression	0.88
Bagging	0.98
Random Forest	0.98
Gradient Boosting	0.92
AdaBoost	0.89
XGBoost	0.99
Validation Performance	
Logistic regression	0.85
Bagging	0.81
Random Forest	0.84
Gradient Boosting	0.86
AdaBoost	0.86
XGBoost	0.86

Link to Appendix slide on model assumptions

Model Performance Summar

Cross-Validation Performance:

- XGBoost stands out with the highest cross-validation recall score on the training data (0.99) and maintains a strong recall score on the validation set (0.86), making it the most consistent and reliable choice for identifying failures.
- Bagging and Random Forest show excellent recall scores on the training data (both 0.98), but they exhibit a notable performance drop on the validation set (to 0.81 and 0.84, respectively), indicating they may not generalize as well as XGBoost.
- Gradient Boosting and AdaBoost present solid recall scores on training (0.92 and 0.89) and validation (0.86 for both), offering a good balance of performance and potentially more simplicity than XGBoost.
- Logistic Regression has the lowest training score (0.88) but shows an increase in recall for the validation set (0.85), suggesting it is less likely to overfit than more complex models.



XGBoost is likely the best option for minimizing missed failure detections without sacrificing performance on unseen data. Logistic Regression, while not as powerful, could be a simpler and more cost-effective solution that still offers reasonably good recall. Bagging and Random Forest may require additional tweaking to reduce overfitting.

Model Performance Summary

Model Building with undersampled data

Before Undersampling: 840 instances of label '1' (failures) and 14160 instances of label '0' (non-failures) in your training data.

Undersampling: used Random Under Sampler to balance the class distribution by randomly removing instances from the majority class (label '0') so that its count matches that of the minority class (label '1'). After undersampling, both classes have 840 instances each.

After Undersampling: The shape of your training data (features) changed from (28320, 40) to (1680, 40), and the shape of the target labels (y_train_un) changed from (28320,) to (1680,).

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,)

Link to Appendix slide on model assumptions

Model Performance Summary

Undersampled data

- Random Forest and XGBoost achieved the highest cross-validation scores, with scores of 0.899 and 0.906, respectively.
- Gradient Boosting closely followed with a score of 0.895, indicating good generalization performance.
- Bagging showed slightly lower performance compared to other models, with a score of 0.874.
- Logistic Regression and AdaBoost had similar performance, with scores of 0.856 and 0.863, respectively.
- Across the validation set, the performance was consistent with the cross-validation results, confirming the models' robustness.

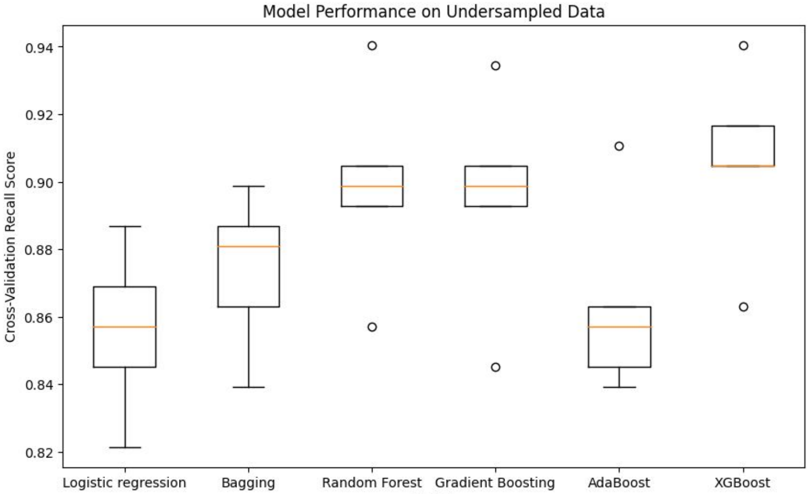
Cross-Validation performance on training dataset	
Logistic regression	0.86
Bagging	0.87
Random Forest	0.90
Gradient Boosting	0.90
AdaBoost	0.86
XGBoost	0.91
Validation Performance	
Logistic regression	0.86
Bagging	0.85
Random Forest	0.88
Gradient Boosting	0.89
AdaBoost	0.86
XGBoost	0.89

Link to Appendix slide on model assumptions

Model Performance Summar

Undersampled data

- Logistic Regression seems to have the broadest range of recall scores, indicating variability in its performance
- Random Forest and Gradient Boosting appear to have a similar range of recall scores with a tight interquartile range, denoting stable performance and with some outliers
- AdaBoost has a smaller interquartile range, implying consistent but lower performance.
- **XGBoost shows** a compact box but with outliers, suggesting mostly consistent high performance with a few exceptions.
- Models with higher and more consistent recall scores, such as XGBoost, Random Forest, and Gradient Boosting, would be preferable as they would reliably detect more true positive cases, which is crucial in scenarios like predictive maintenance to avoid costly downtimes.
- Logistic Regression may not be as reliable due to its wider performance range.



Consistency is important; models with fewer outliers (like Random Forest and Bagging) might be more predictable in their performance and thus more trustworthy for business applications. As models with a narrow interquartile range and few outliers are considered consistent and reliable, reducing the risk of unpredictably poor performance in a business setting.

Link to Appendix slide on model assumptions

Model Performance Summary

The models that consistently demonstrate high recall scores on both the cross-validation and validation datasets are:

Random Forest
Gradient Boosting
XGBoost

Link to Appendix slide on model assumptions

Model Performance Summary

Performed hyperparameter tuning using **RandomizedSearchCV for the AdaBoostClassifier model**.

The best parameters obtained from the random search are: n_estimators: 200, learning_rate: 0.2, base_estimator: DecisionTreeClassifier with max_depth of 3.

AdaBoostClassifier
base_estimator: DecisionTreeClassifier
DecisionTreeClassifier

Train performance

	Accuracy	Recall	Precision	F1
0	0.991	0.986	0.996	0.991

Validation performance

	Accuracy	Recall	Precision	F1
0	0.982	0.856	0.816	0.835

Overall, the model demonstrates high accuracy and precision on the training set, indicating good performance in classifying the target variable. However, there is a slight drop in performance metrics on the validation set, particularly in recall, precision, and F1 score, suggesting some level of overfitting or suboptimal generalization to unseen data.

[Link to Appendix slide on model assumptions]

Model Performance Summary

Tuning Random forest using undersampled data

- Best Parameters:
  - `n_estimators`: 250
  - `min_samples_leaf`: 1
  - `max_samples`: 0.6
  - `max_features`: 'sqrt'
- Cross-Validation Score: 89.76%

```
RandomForestClassifier
RandomForestClassifier(max_samples=0.6, n_estimators=250, random_state=1)
```

Train performance

	Accuracy	Recall	Precision	F1
0	0.990	0.980	1.000	0.990

Validation performance

	Accuracy	Recall	Precision	F1
0	0.941	0.878	0.474	0.616

The model performs exceptionally well on the training set, achieving high accuracy, recall, precision, and F1 score. However, there seems to be a drop in performance on the validation set, particularly in precision and F1 score, indicating that the model might be overfitting to the training data. It's essential to further evaluate and potentially refine the model to generalize better to unseen data.

Link to Appendix slide on model assumptions

Model Performance Summa

Tuning Gradient Boosting using oversampled data

```
GradientBoostingClassifier
GradientBoostingClassifier(learning_rate=1, max_features=0.5, n_estimators=150,
                           random_state=1, subsample=0.7)
```

The model achieves a remarkable training accuracy of 99.3%, along with high recall, precision, and F1-score values, all above 99%. This suggests that the model accurately predicts the target variable on the training dataset.

However, on the validation set, while the accuracy remains decent at 96.3%, there is a noticeable drop in recall, precision, and F1-score, all around 83-62%. This indicates that the model's performance on unseen data is not as robust as on the training data, particularly in correctly identifying positive cases. Therefore, further optimization or fine-tuning may be required to enhance its generalization capabilities.

	Accuracy	Recall	Precision	F1
0	0.993	0.994	0.992	0.993

Train performance

Validation performance

	Accuracy	Recall	Precision	F1
0	0.963	0.837	0.619	0.712

[Link to Appendix slide on model assumptions]

Model Performance Sum

Tuning XGBoost using oversampled data

The XGBoost model achieves a training accuracy of 99.9%, with perfect recall and precision, suggesting excellent performance on the oversampled training data. However, on the validation set, the accuracy drops to 98.3%, with a recall of 88.5% and precision of 81.6%. Despite the slight drop in performance compared to the training set, the model still maintains strong predictive capabilities on unseen data.

XGBClassifier

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=0, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=0.1, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=250,
              n_jobs=None, num_parallel_tree=None, random_state=1, ...)
```

Train performance

	Accuracy	Recall	Precision	F1
0	0.999	1.000	0.999	0.999

Validation performance

Accuracy	Recall	Precision	F1
0.983	0.885	0.816	0.849

[*Link to Appendix slide on model assumptions*]

Productionize and test the final model using pipelines

- Summary of the performance of the model built with pipeline on test dataset

	Accuracy	Recall	Precision	F1
0	0.979	0.855	0.788	0.820

XGBoost tuned with oversampled data model performs reasonably well on the test set, with high accuracy and **decent recall,** precision, and F1 score.

Link to Appendix slide on model assumptions

Model Performance Summary

Training performance comparison:

	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	0.991	0.990	0.999
Recall	0.994	0.986	0.980	1.000
Precision	0.992	0.996	1.000	0.999
F1	0.993	0.991	0.990	0.999

Validation performance comparison:

	Gradient Boosting tuned with oversampled data	AdaBoost classifier tuned with oversampled data	Random forest tuned with undersampled data	XGBoost tuned with oversampled data
Accuracy	0.963	0.982	0.941	0.983
Recall	0.837	0.856	0.878	0.885
Precision	0.619	0.816	0.474	0.816
F1	0.712	0.835	0.616	0.849

Overall, **XGBoost appears** to be the most promising model as it maintains high performance on both the training and validation sets, indicating its suitability for this classification task. Gradient Boosting and AdaBoost also perform well but may require further regularization to prevent overfitting. Random Forest shows a significant drop in performance on the validation set, indicating potential issues with generalization.

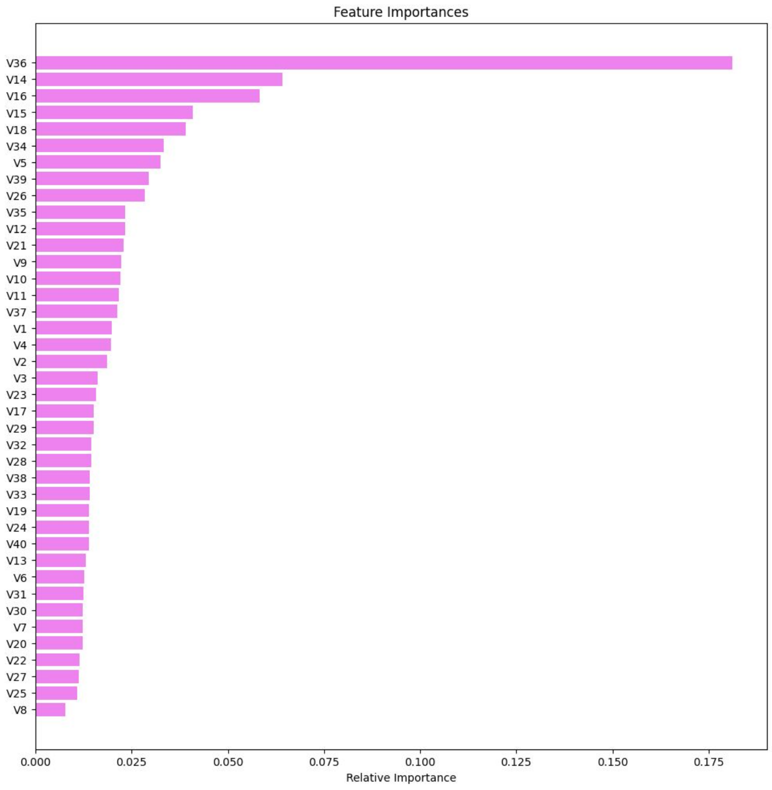
Link to Appendix slide on model assumptions

Productionize and test the final model using pipelines

- Summary of most important factors used by the model built with pipeline for prediction

Most important feature is V36

Least important V8



ions

Productionize and test the final model using pipelines

Steps taken to create a pipeline for the final model

- Imported Necessary Libraries: imported the required libraries, including scikit-learn's Pipeline class and relevant model classes AdaBoostClassifier, RandomForestClassifier, etc.
- Defined Preprocessing Steps (Optional): Since your data preprocessing steps were handled separately (e.g., oversampling or undersampling), didn't explicitly define any preprocessing steps in the pipeline.
- Instantiated Model: instantiated the final model for the analysis. Instantiated RandomForestClassifier with the best parameters obtained from hyperparameter tuning.
- Built Pipeline: used the Pipeline class to create a pipeline. Since there were no preprocessing steps to include, the pipeline only contained the final model.
- Fit Pipeline: fitted the pipeline to training data using the fit method. This sequentially could apply the transformations (none in this case) and then fit the final model.
- Predictions: After fitting the pipeline, use to make predictions on new data using the predict or predict_proba methods, similar to with standalone models.

Leveraged the Pipeline class to encapsulate the final model, making it easier to manage and apply to new data while ensuring consistency in preprocessing steps (not in this case).

[Link to Appendix slide on model assumptions]

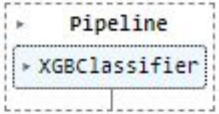
Productionize and test the final model using pipelines

- Summary of most important factors used by the model built with pipeline for prediction
- Get Feature Names: Obtain the names of the features used in the model.
- Extract Feature Importances
- Sort Feature Importances: Sort the feature importances in descending order to identify the most important features.
- Visualize Feature Importances (Optional): Optionally, can create a visualization such as a bar plot to show the relative importance of each feature.

Link to Appendix slide on model assumptions

Productionize and test the final model using pipelines

0.9888 is performance on test set



Best Model and its

Performance:

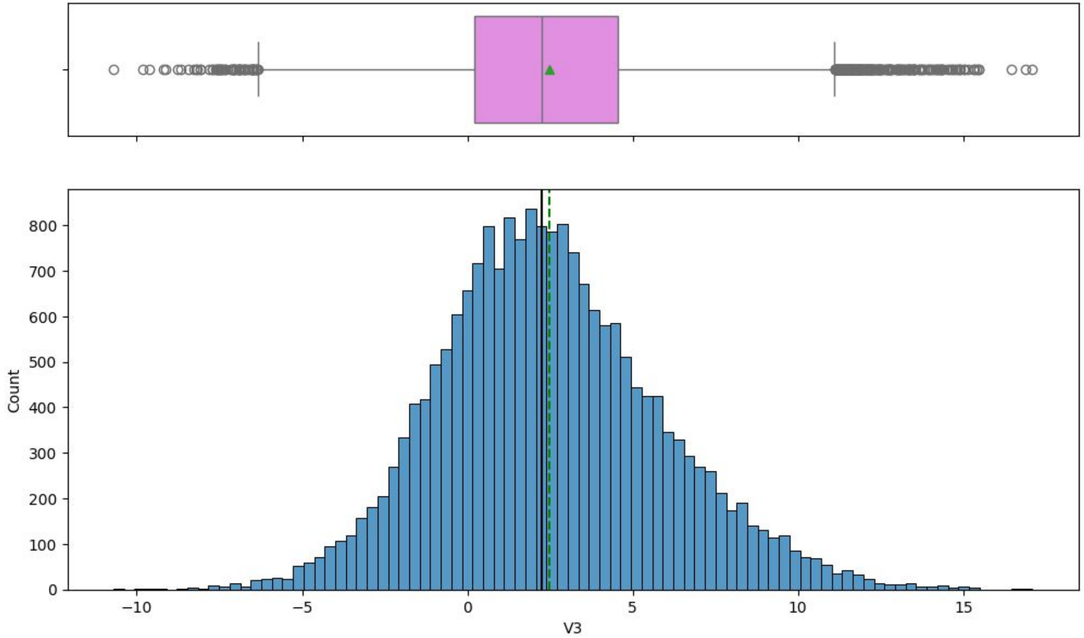
The best model appears to be the XGBoost classifier tuned with oversampled data (`xgb2`). Its performance on the test set is as follows:

- Accuracy: 97.9%
- Recall: 85.5%
- Precision: 78.8%
- F1 Score: 82.0%

Link to Appendix slide on model assumptions

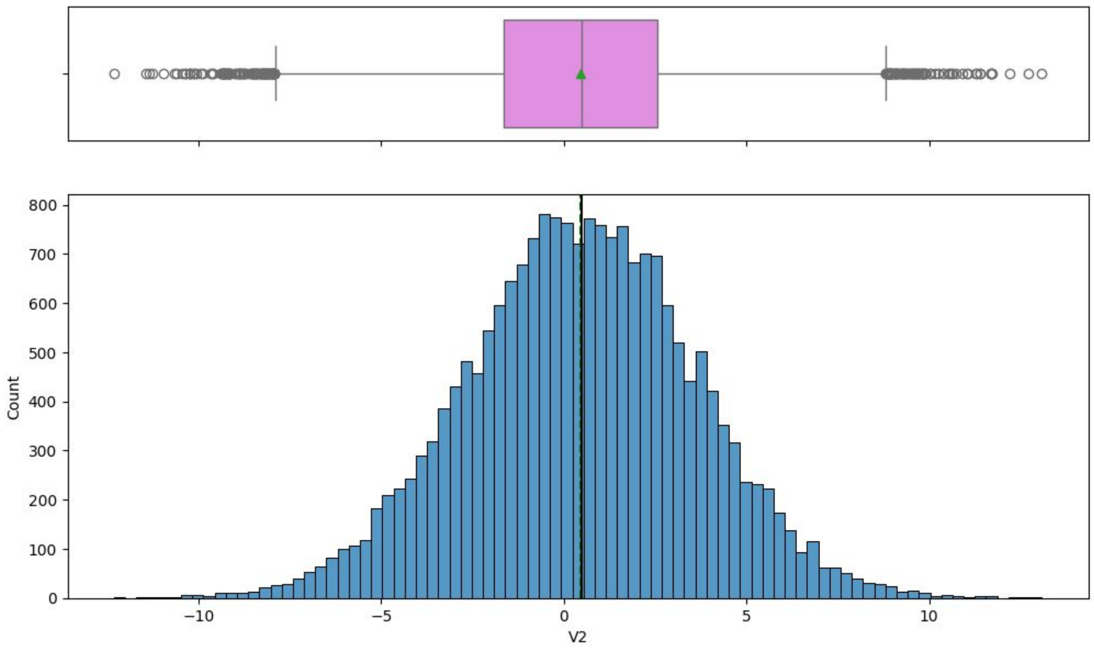
APPENDIX

EDA Results



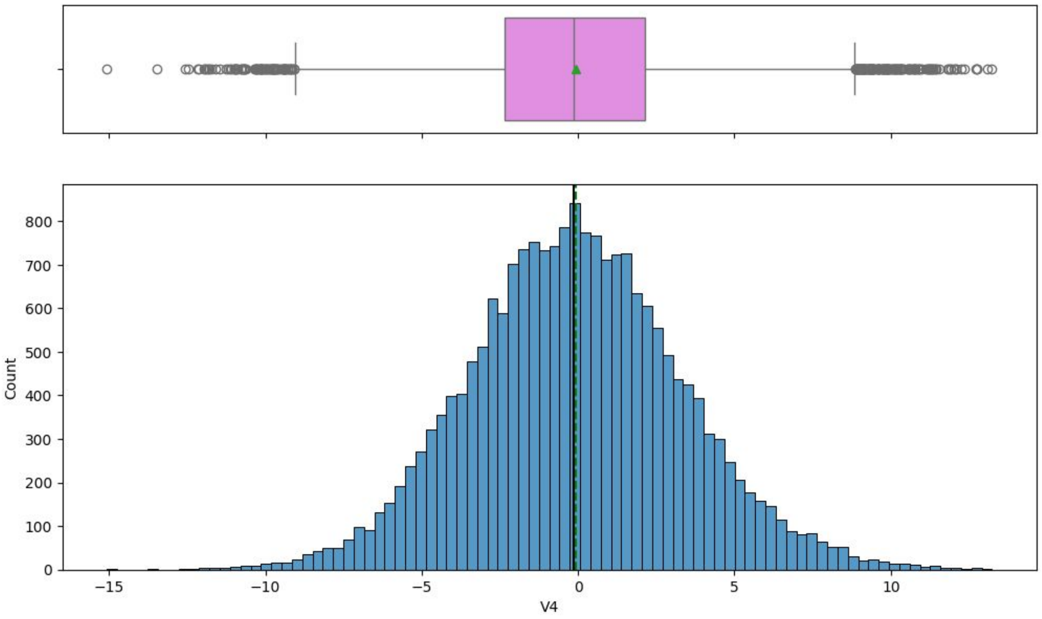
[Link to Appendix slide on data background check]

EDA Results



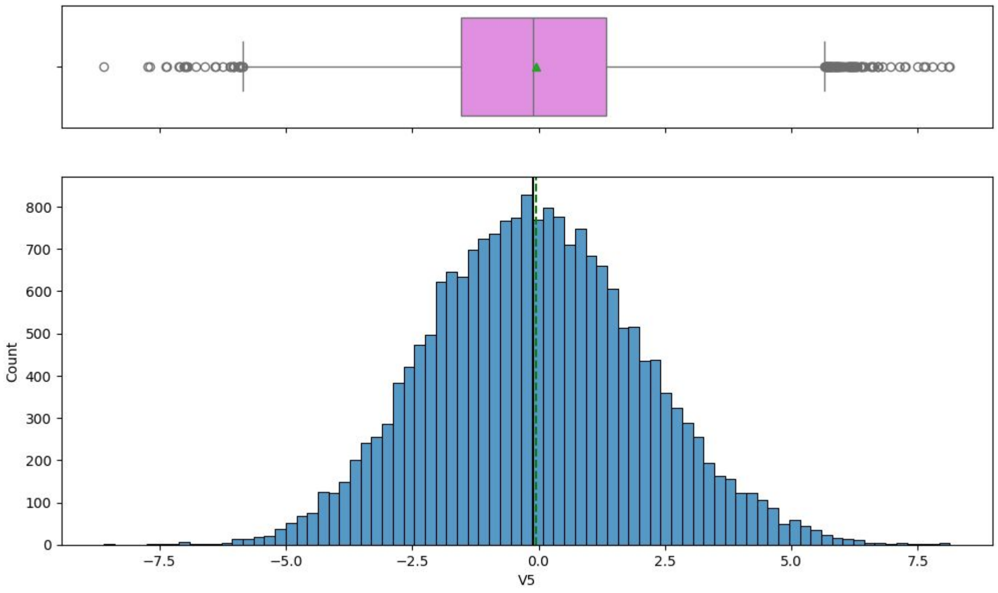
Link to Appendix slide on data background check

EDA Results



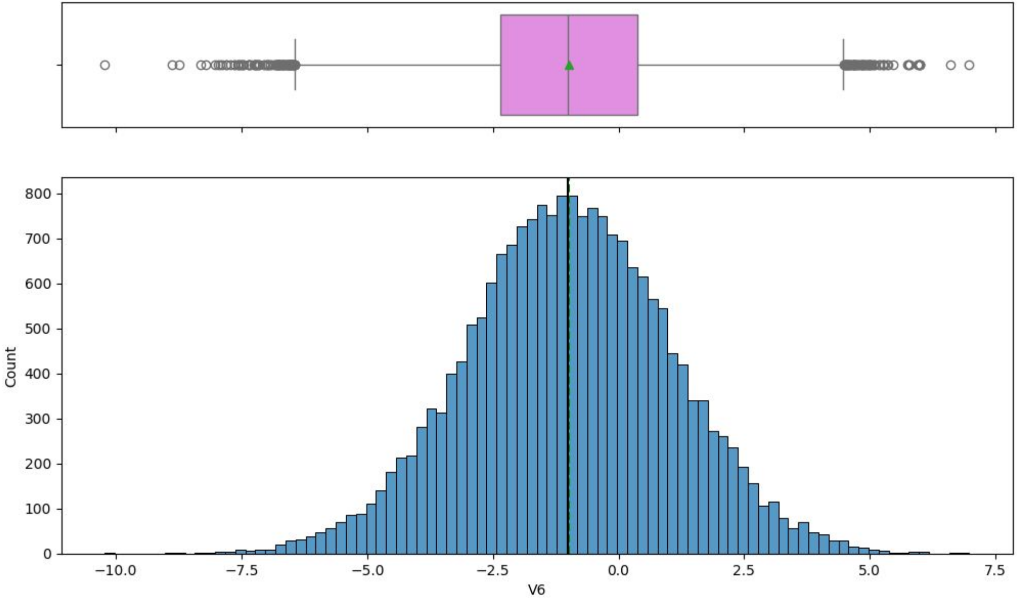
[Link to Appendix slide on data background check]

EDA Results



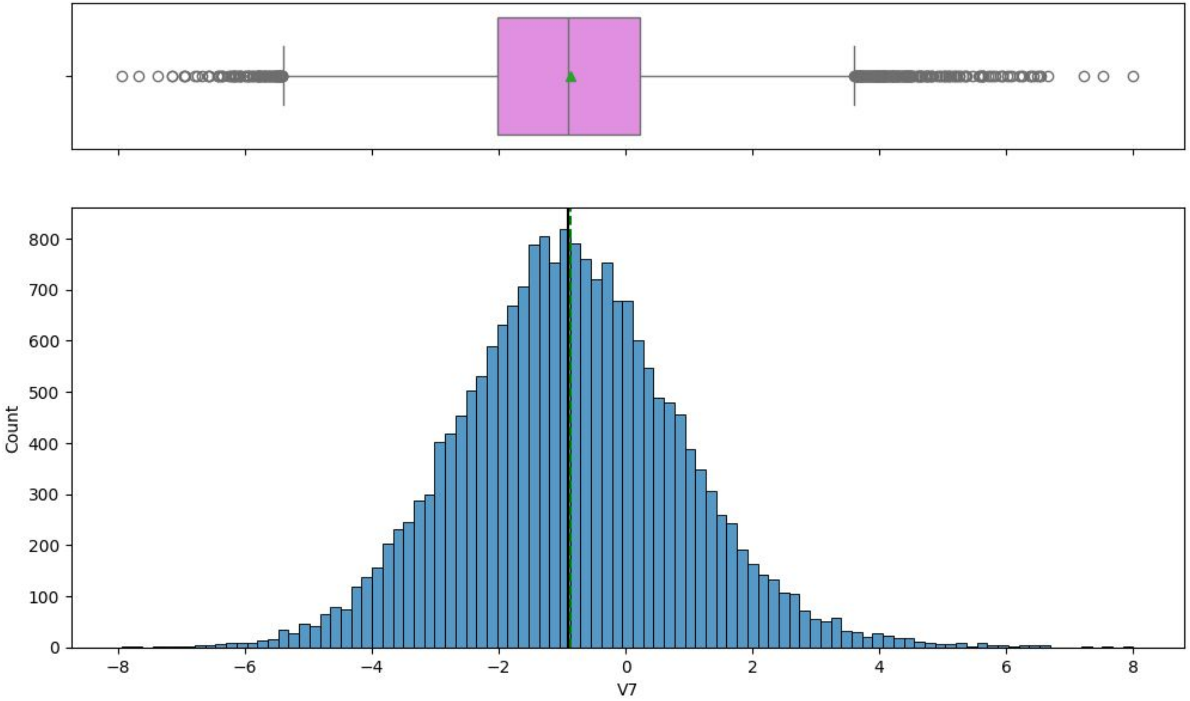
[Link to Appendix slide on data background check]()

EDA Results



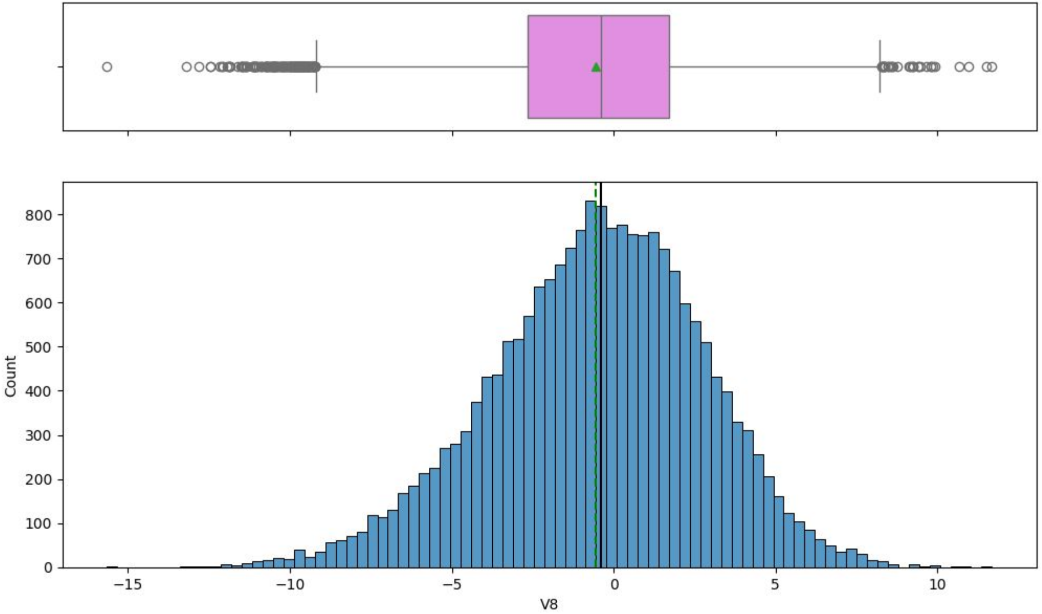
Link to Appendix slide on data background check

EDA Results



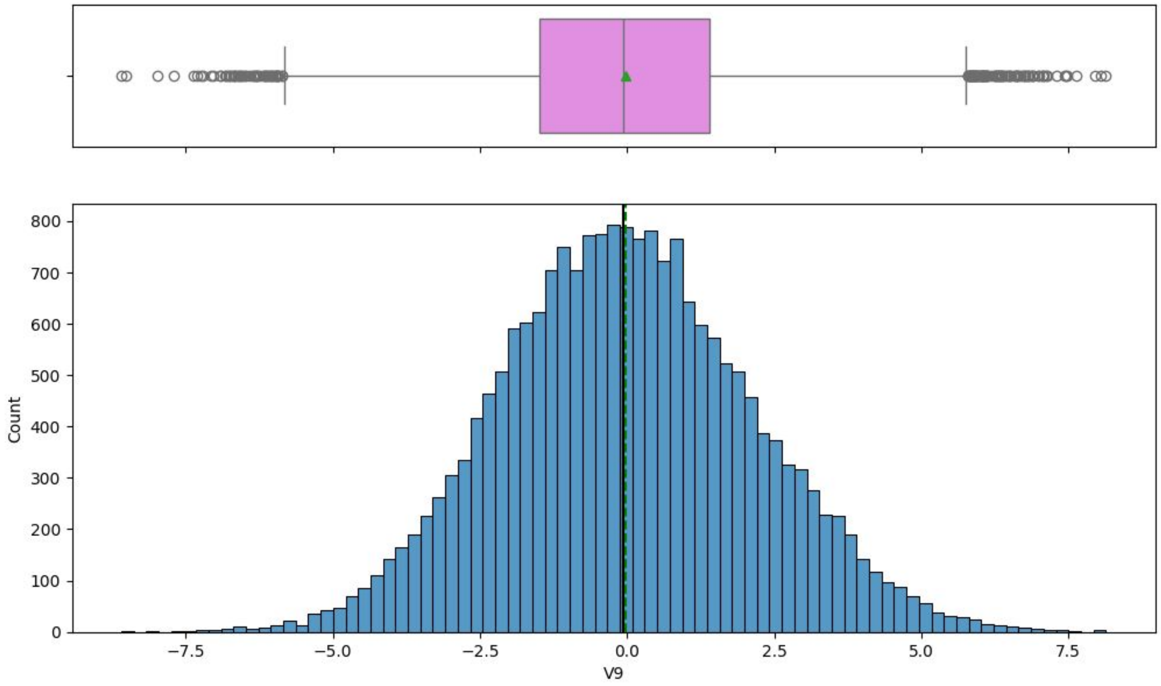
Link to Appendix slide on data background check

EDA Results



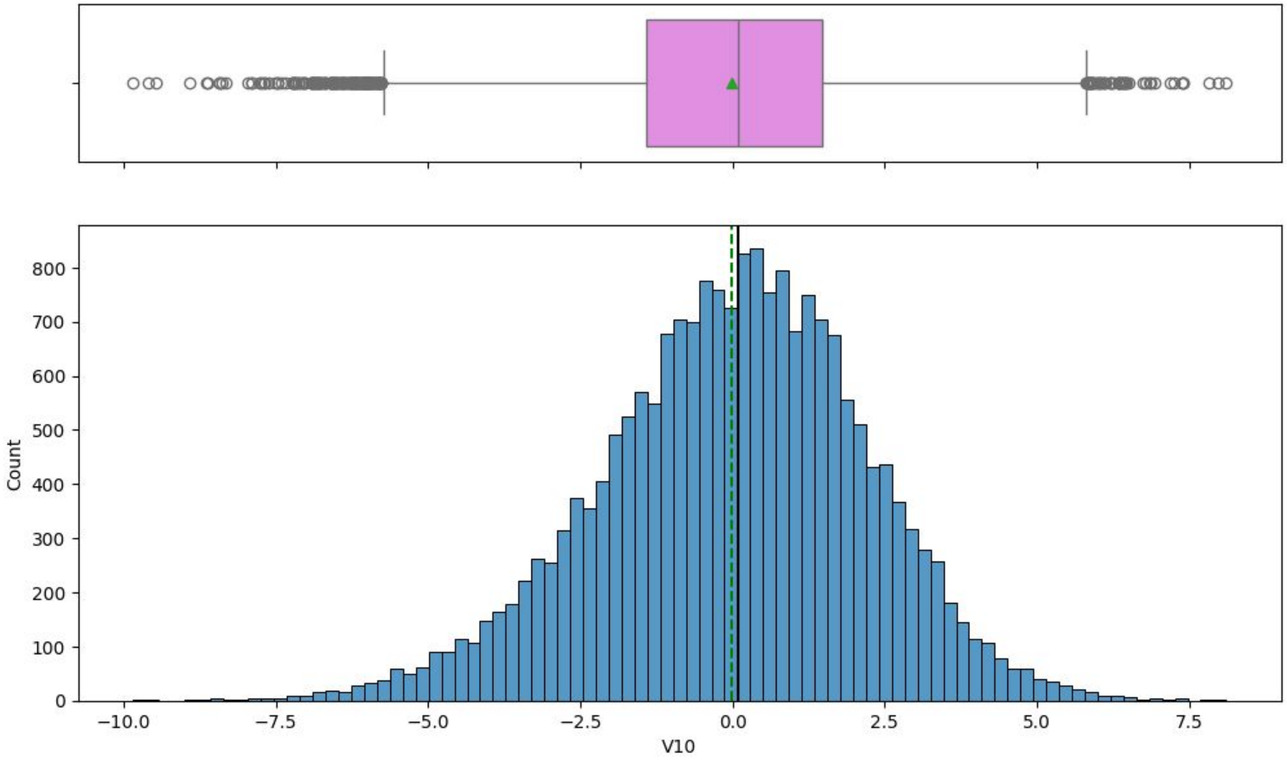
Link to Appendix slide on data background check

EDA Results



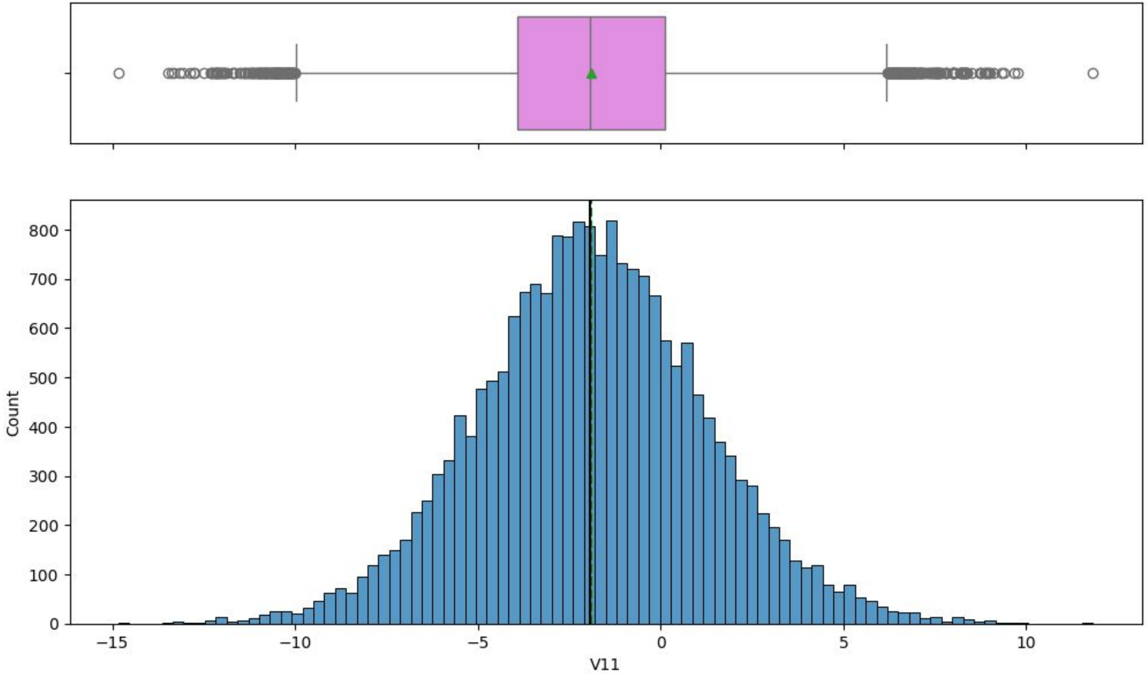
Link to Appendix slide on data background check

EDA Results



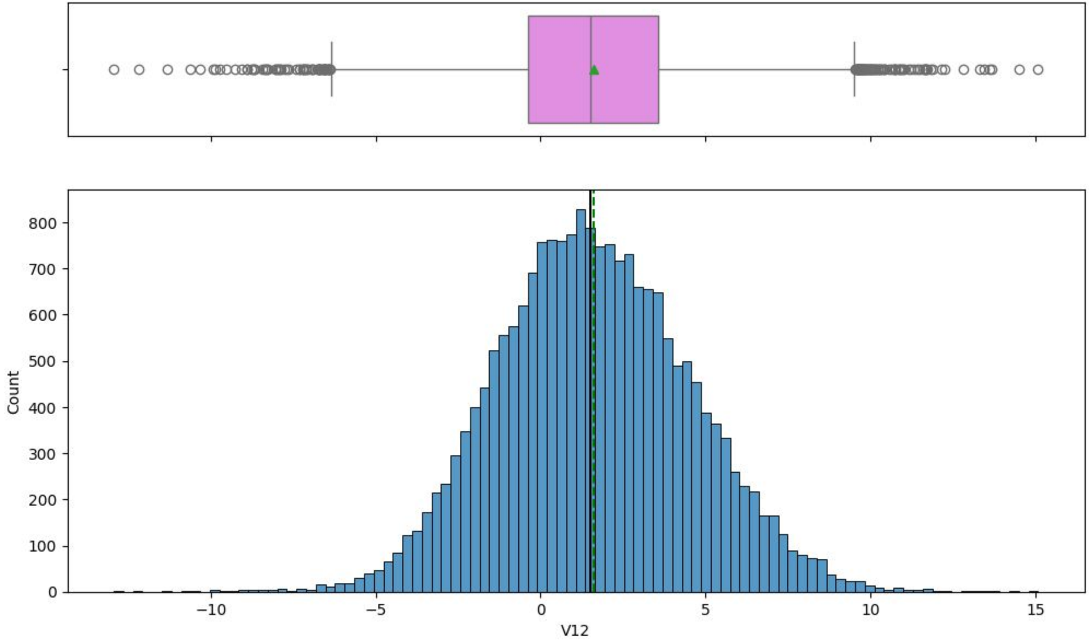
[Link to Appendix slide on data background check](#)

EDA Results



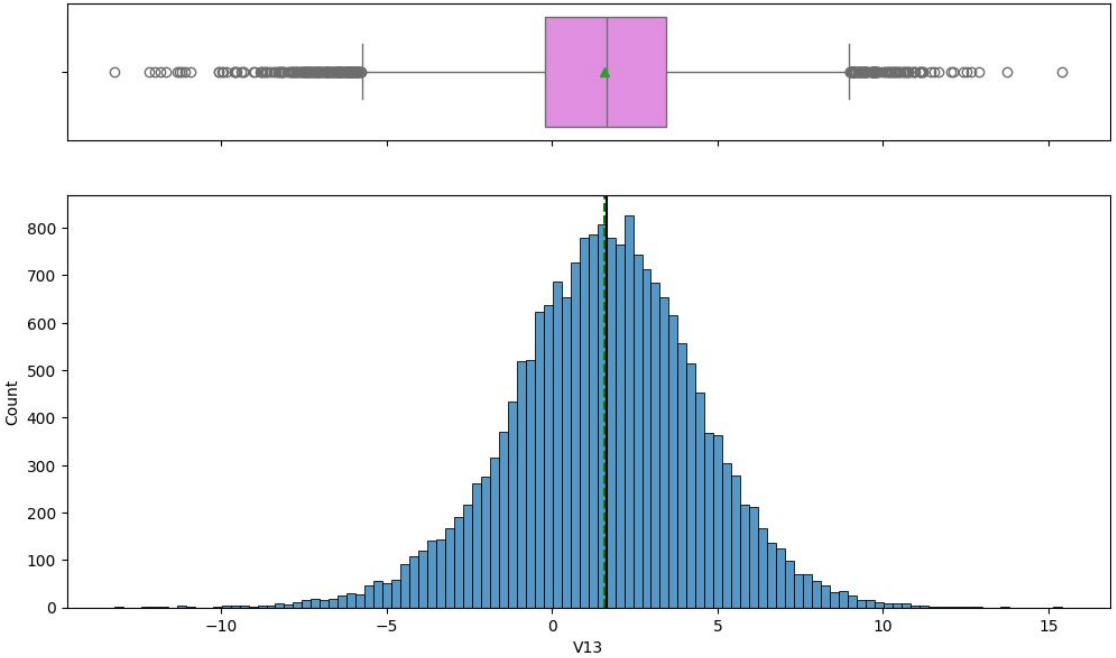
Link to Appendix slide on data background check

EDA Results



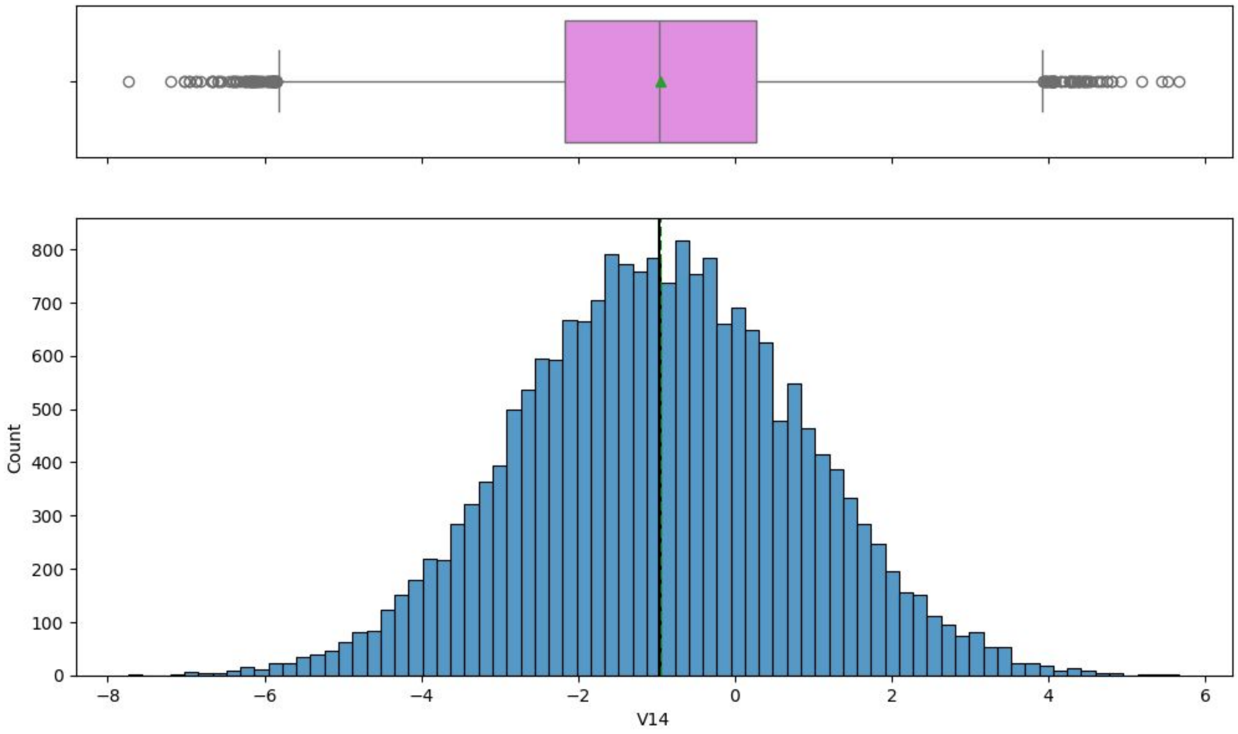
Link to Appendix slide on data background check

EDA Results



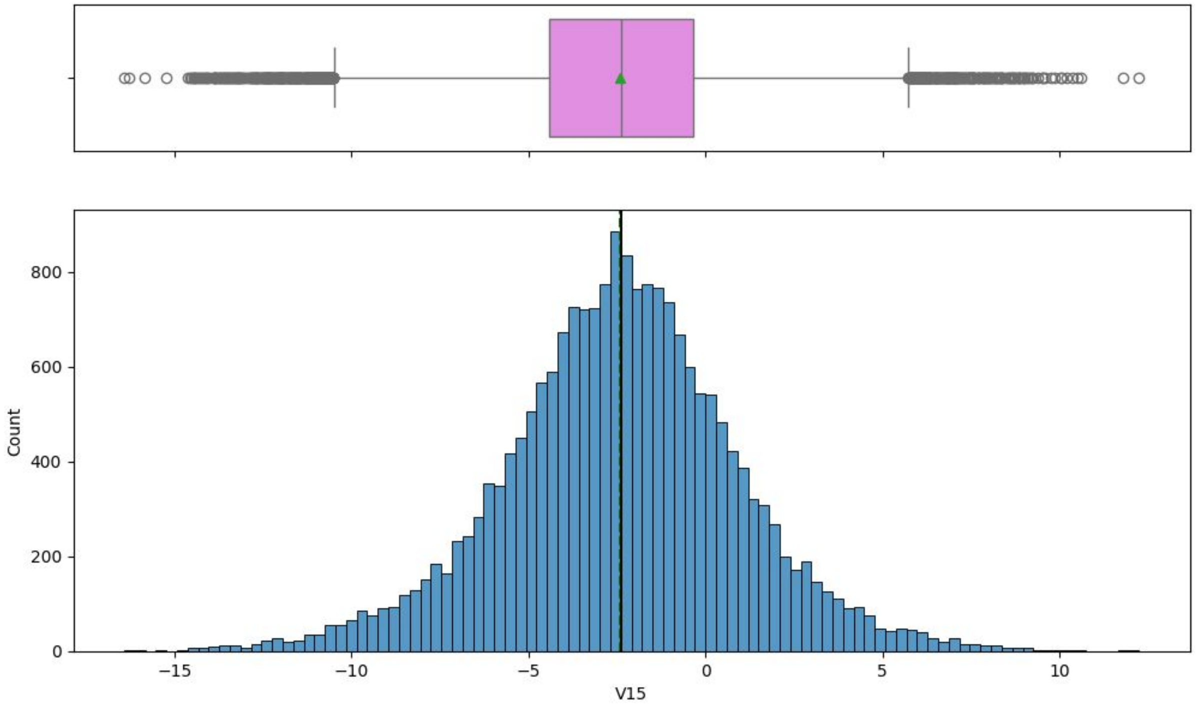
[Link to Appendix slide on data background check]

EDA Results



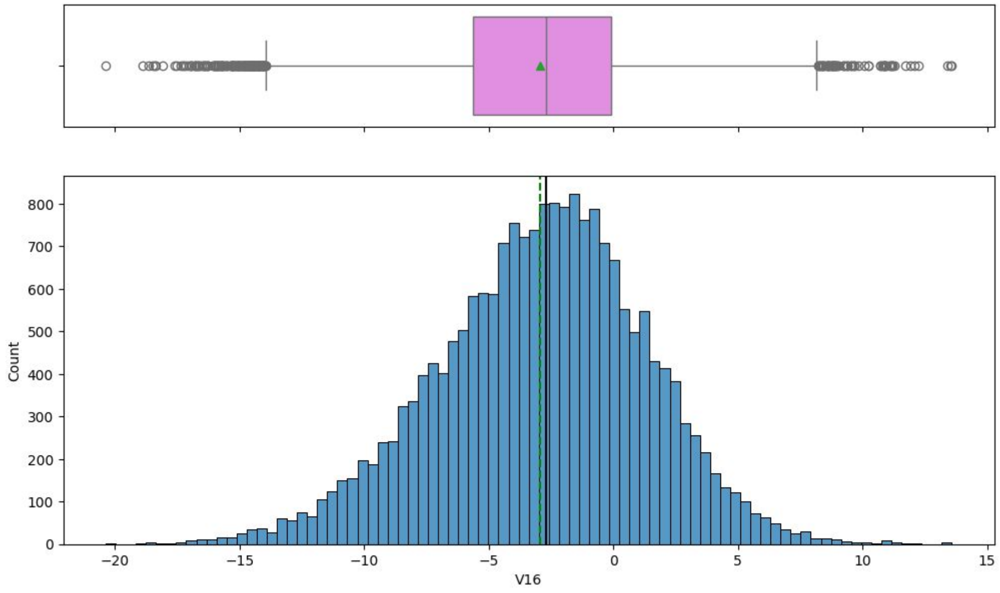
[Link to Appendix slide on data background check]

EDA Results



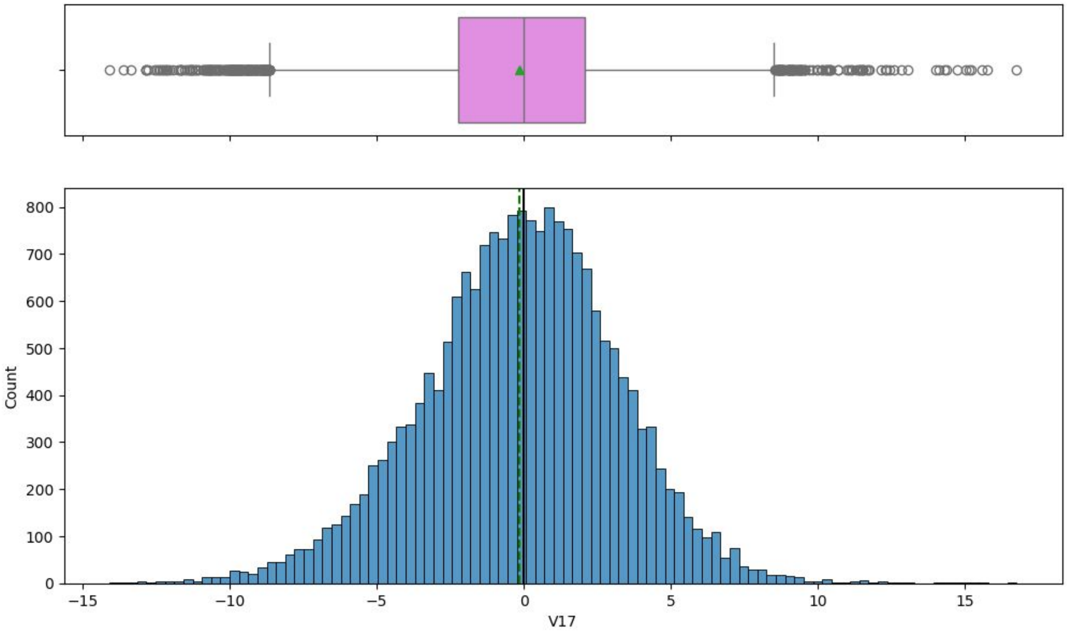
[Link to Appendix slide on data background check]

EDA Results



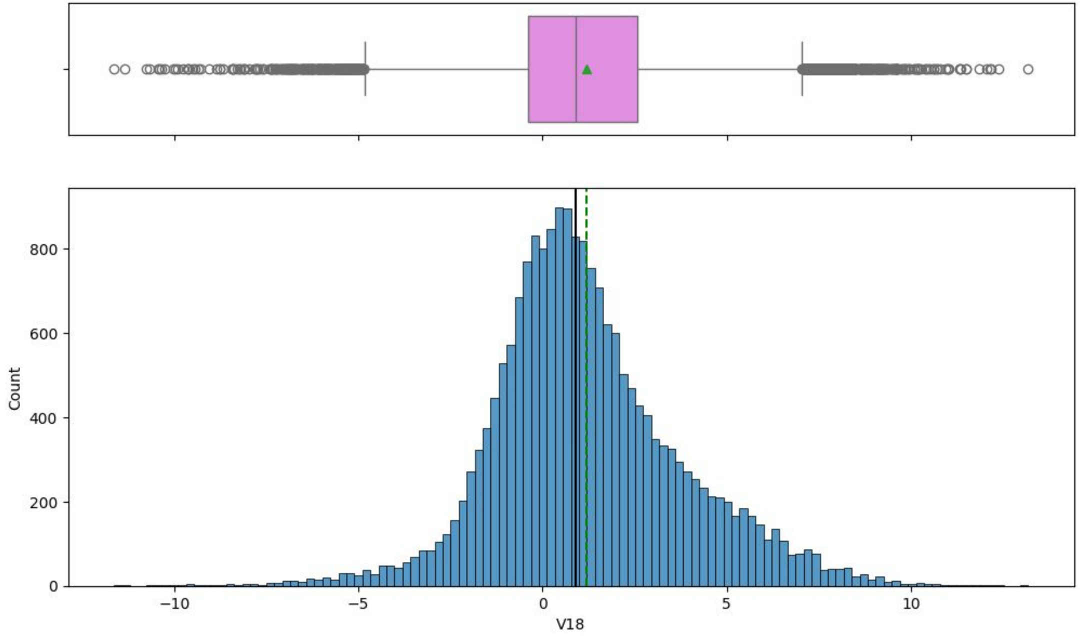
[Link to Appendix slide on data background check]

EDA Results



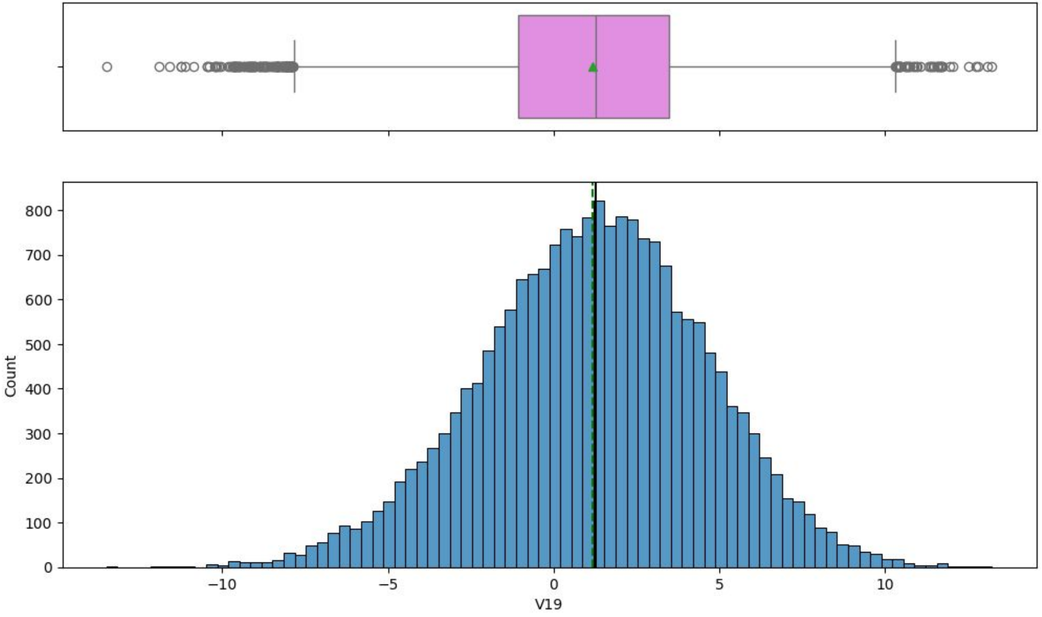
[*Link to Appendix slide on data background check*]

EDA Results



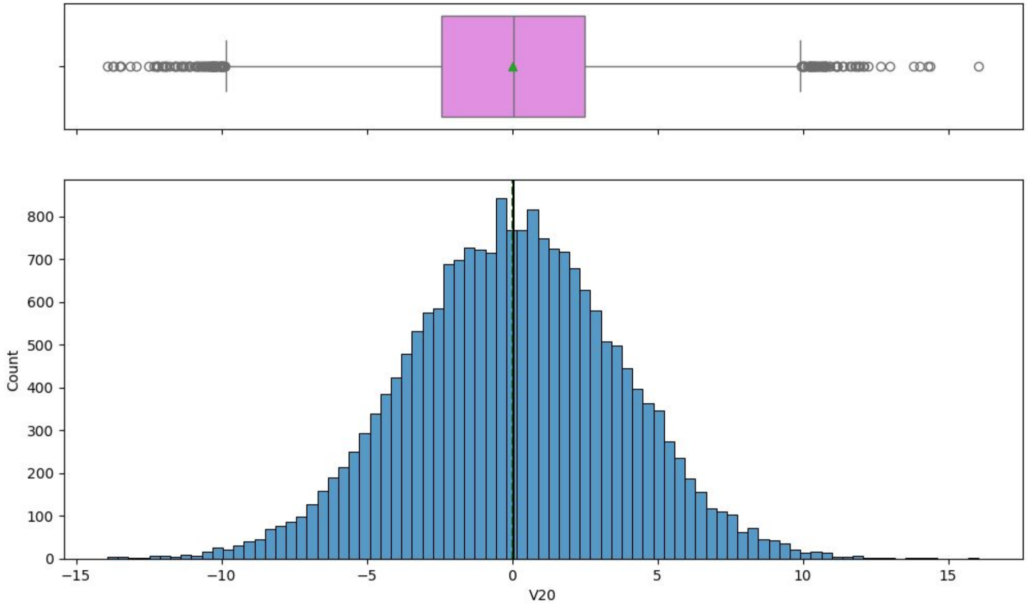
Link to Appendix slide on data background check

EDA Results



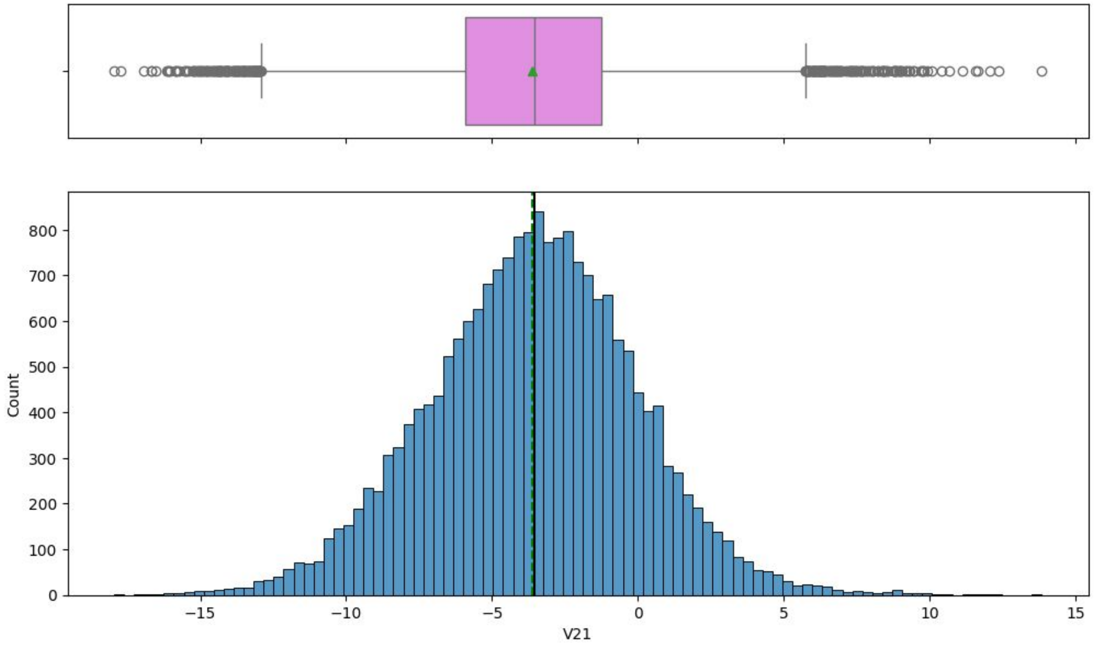
[Link to Appendix slide on data background check]

EDA Results



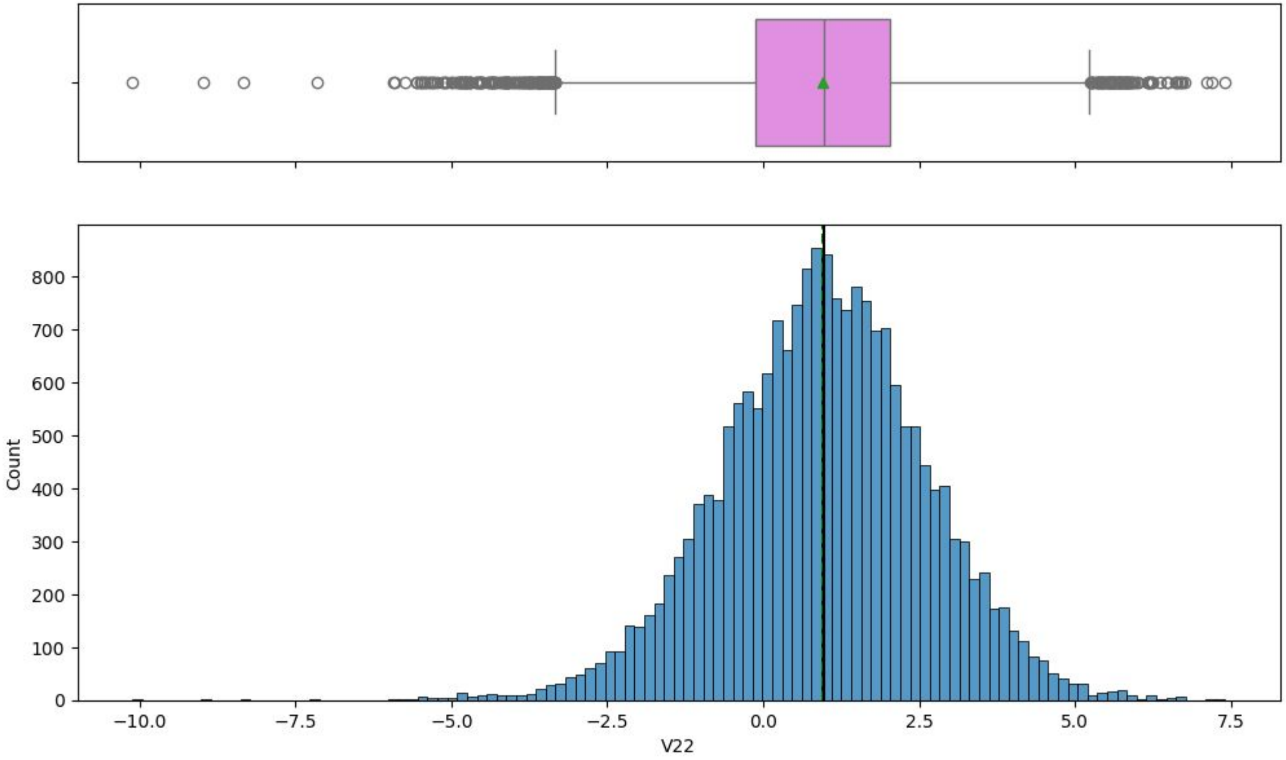
[Link to Appendix slide on data background check]

EDA Results



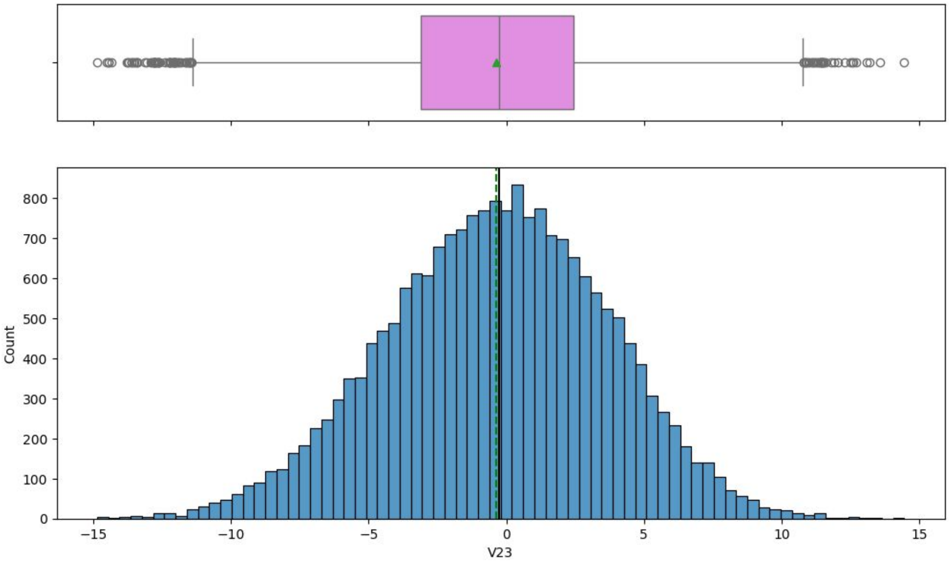
[Link to Appendix slide on data background check]

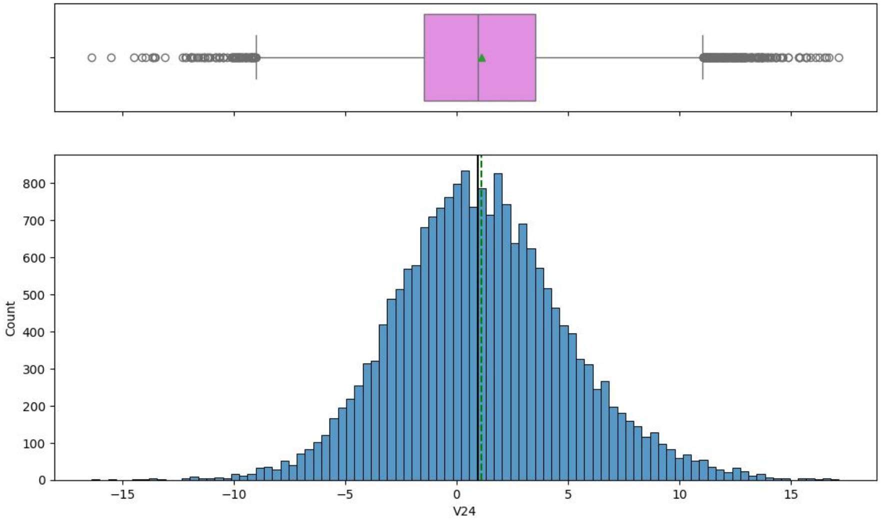
EDA Results



[Link to Appendix slide on data background check](#)

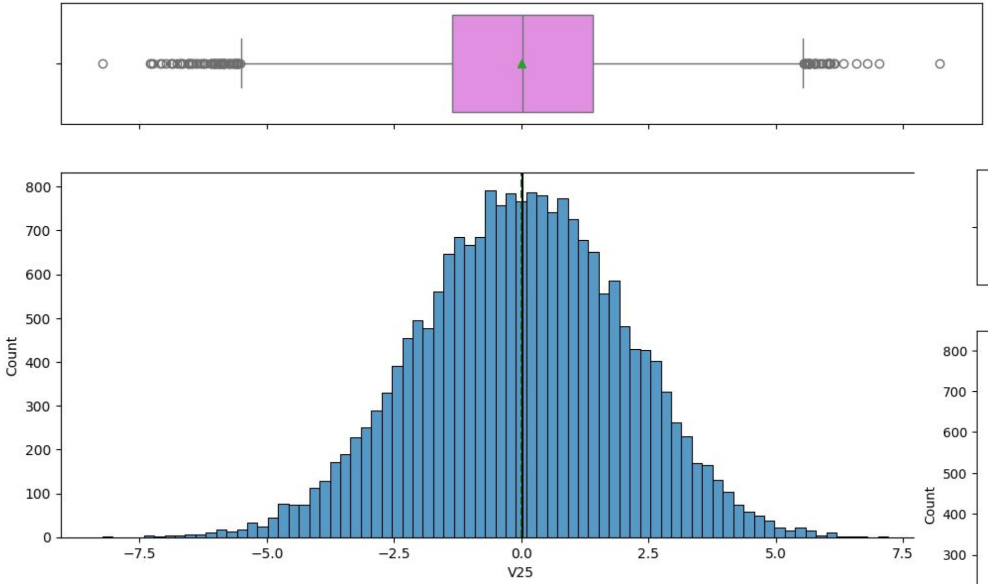
EDA Results

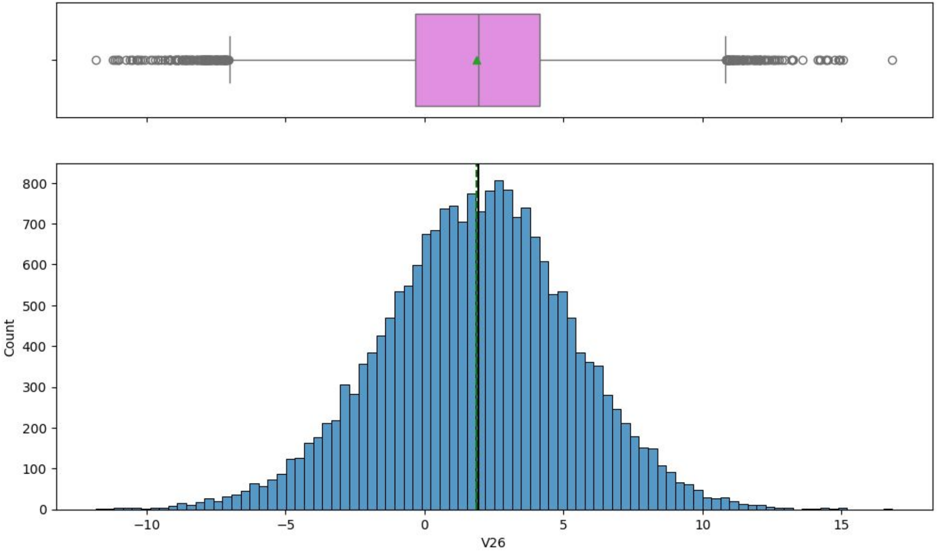




Link to Appendix slide on data background check

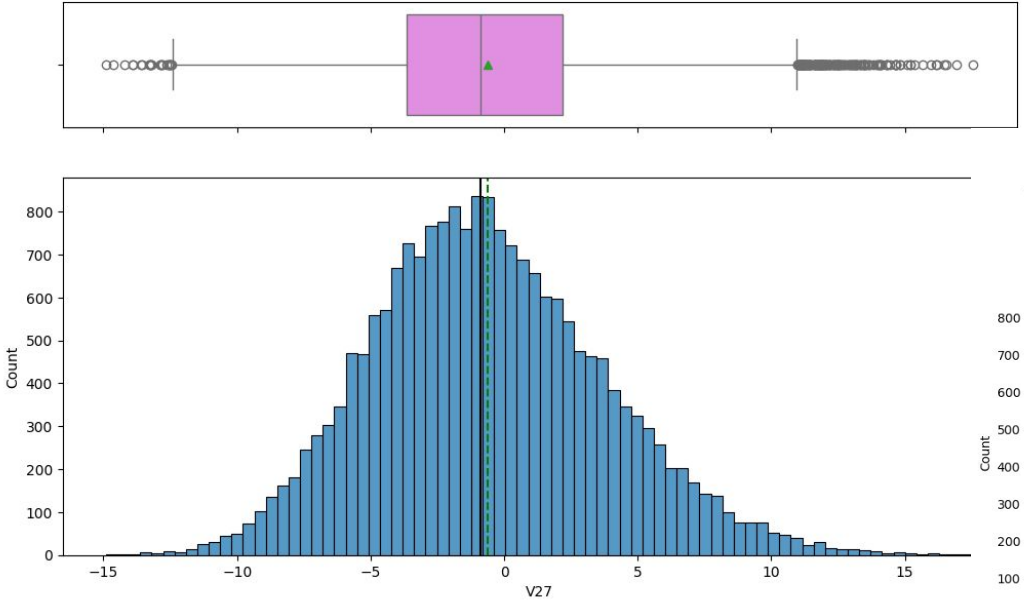
EDA Results

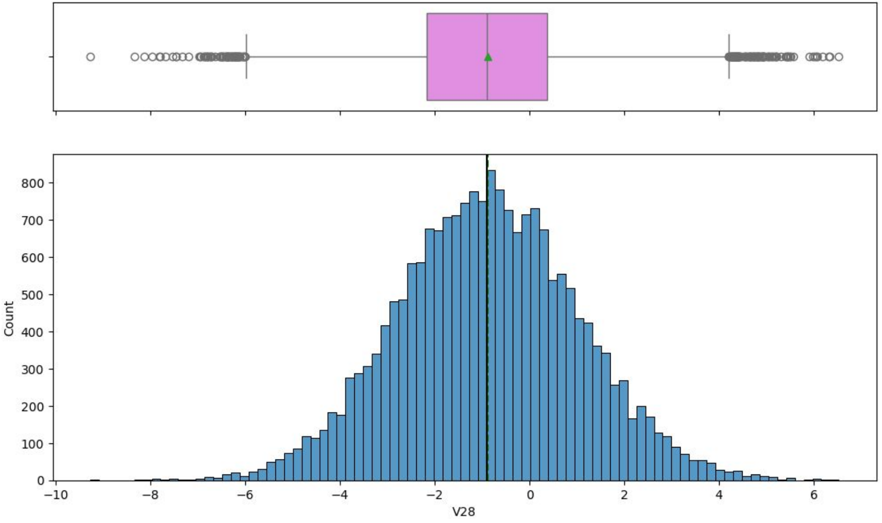




[Link to Appendix slide on data background check]

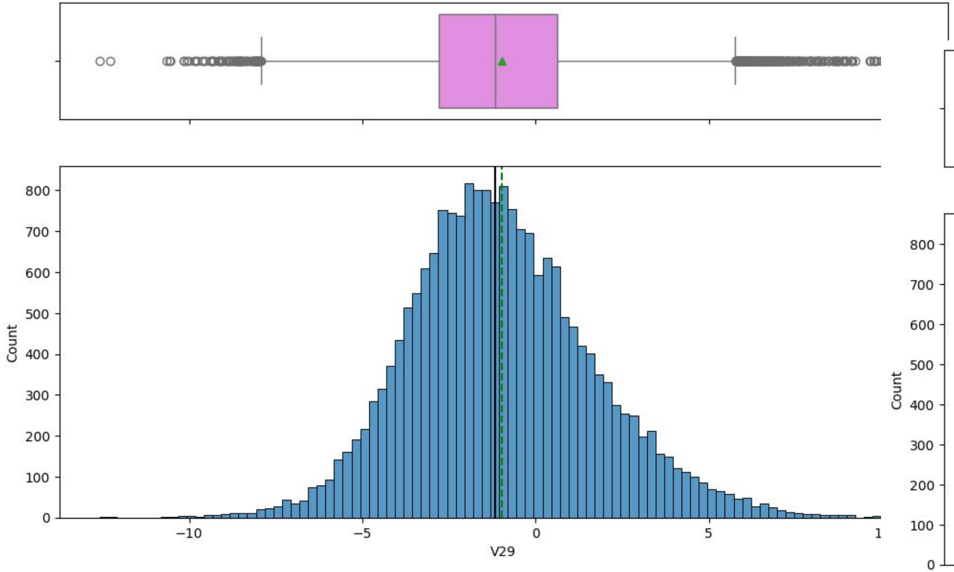
EDA Results

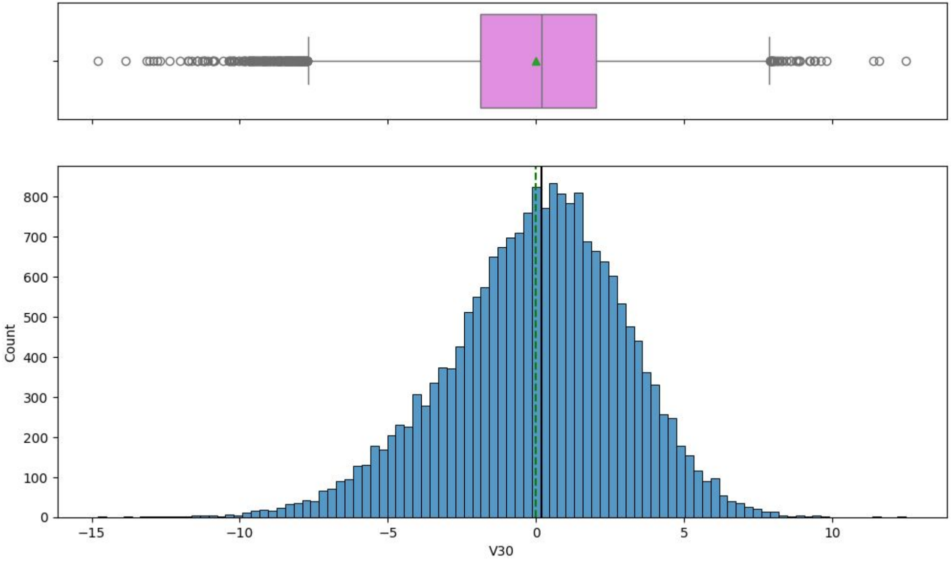




Link to Appendix slide on data background check

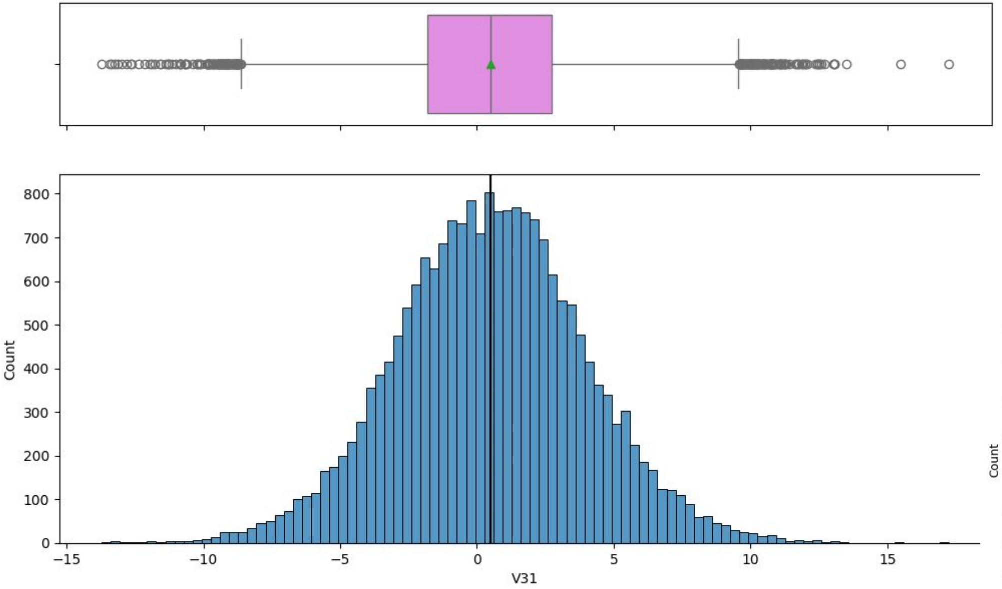
EDA Results

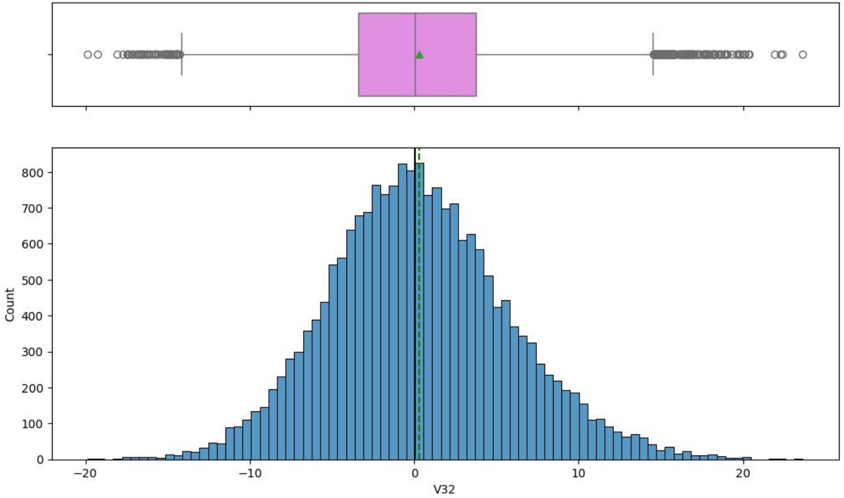




[*Link to Appendix slide on data background check*]

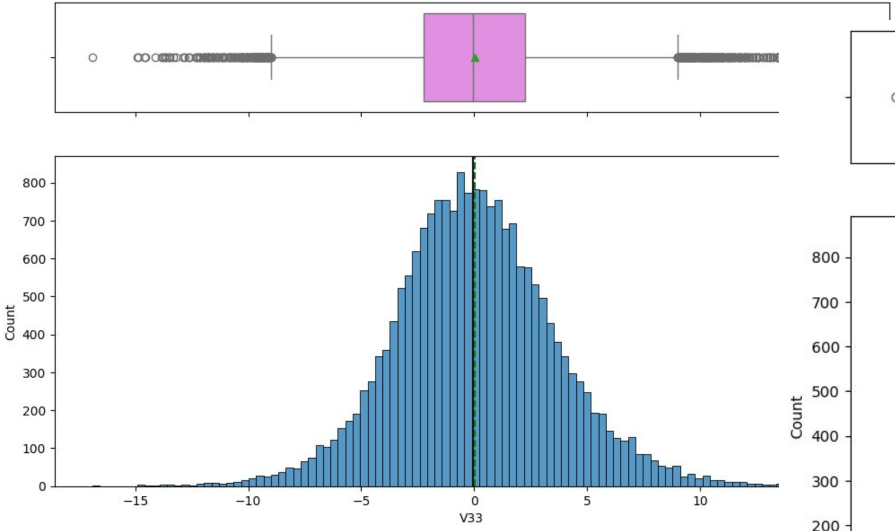
EDA Results

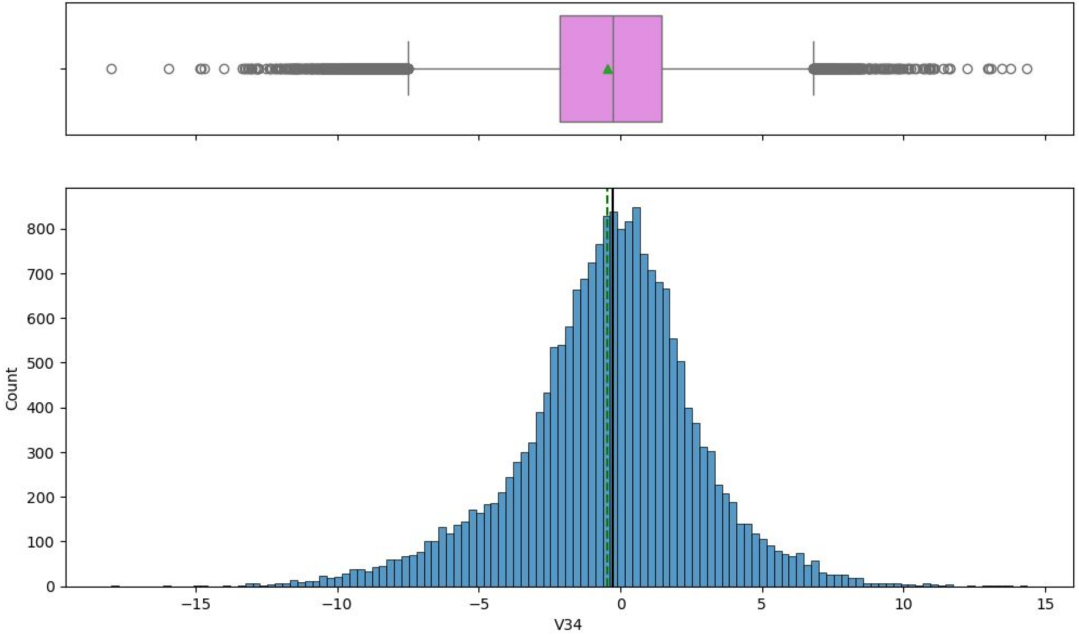




[Link to Appendix slide on data background check](#)

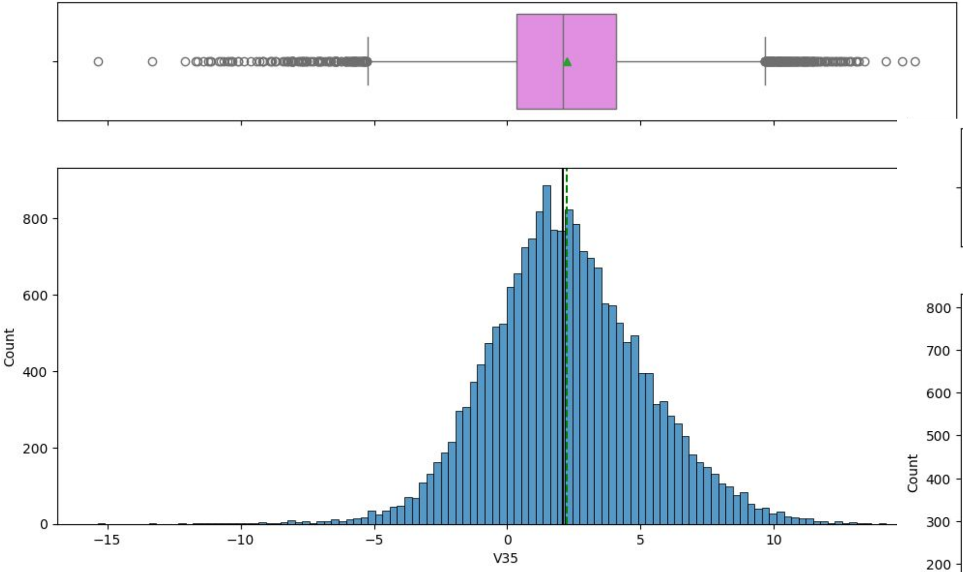
EDA Results

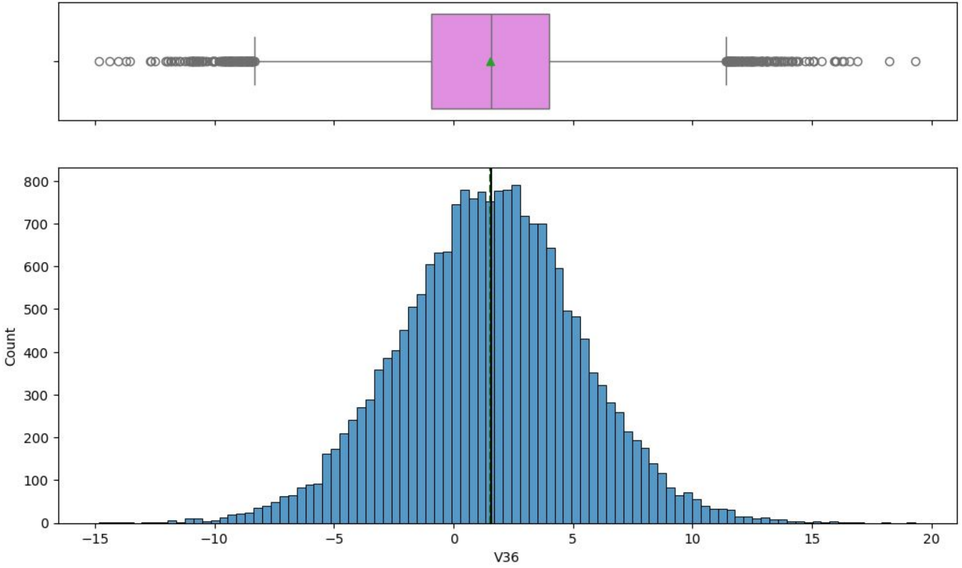




[Link to Appendix slide on data background check]()

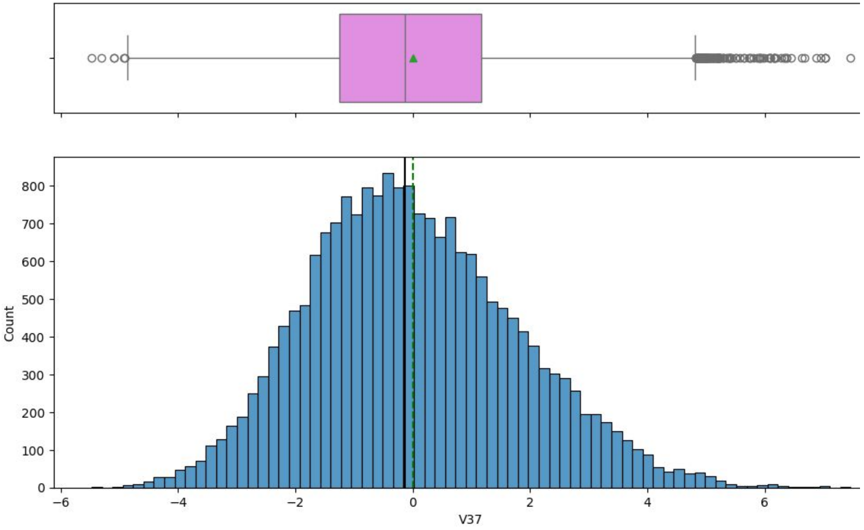
EDA Results

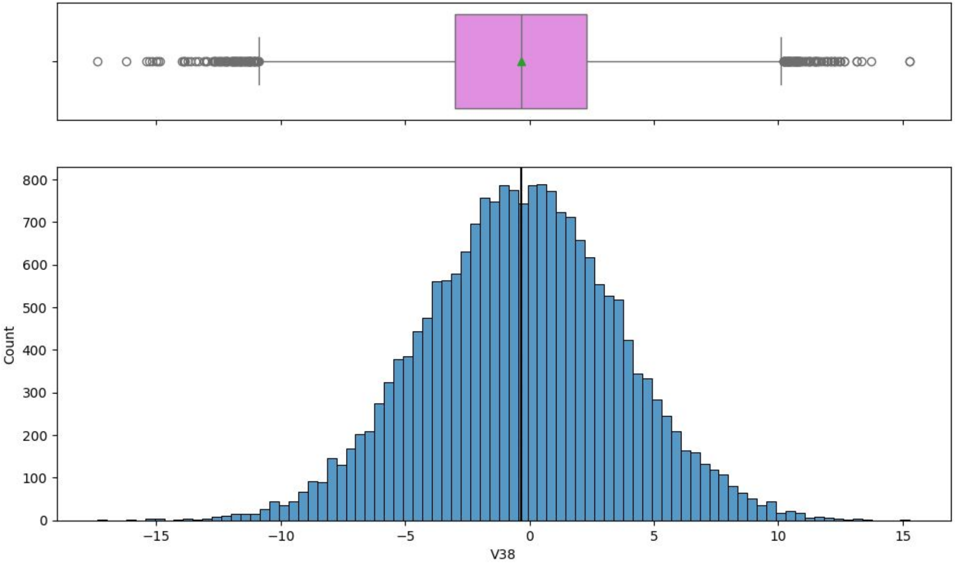




[Link to Appendix slide on data background check]

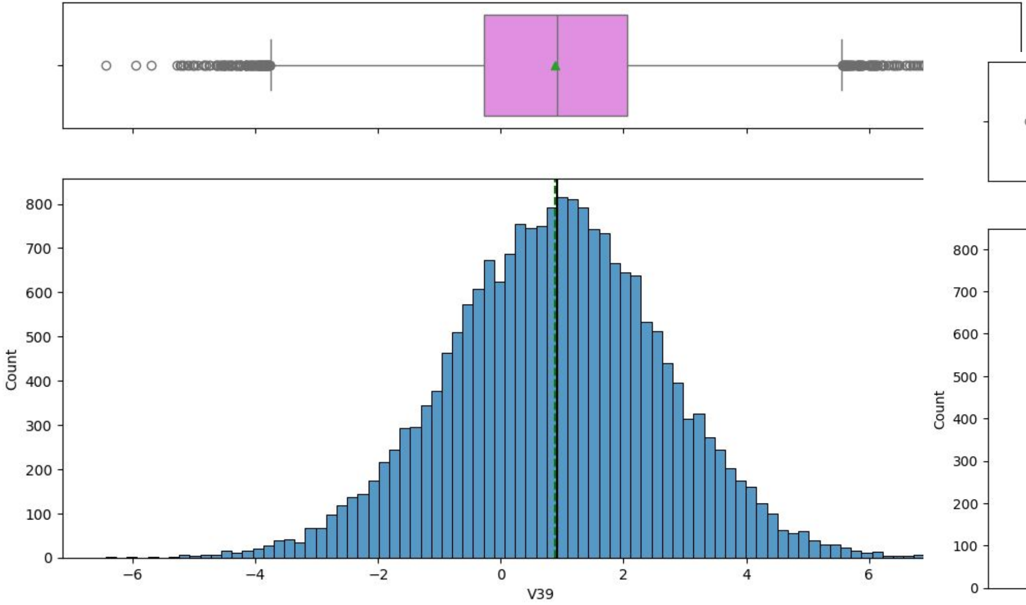
EDA Results

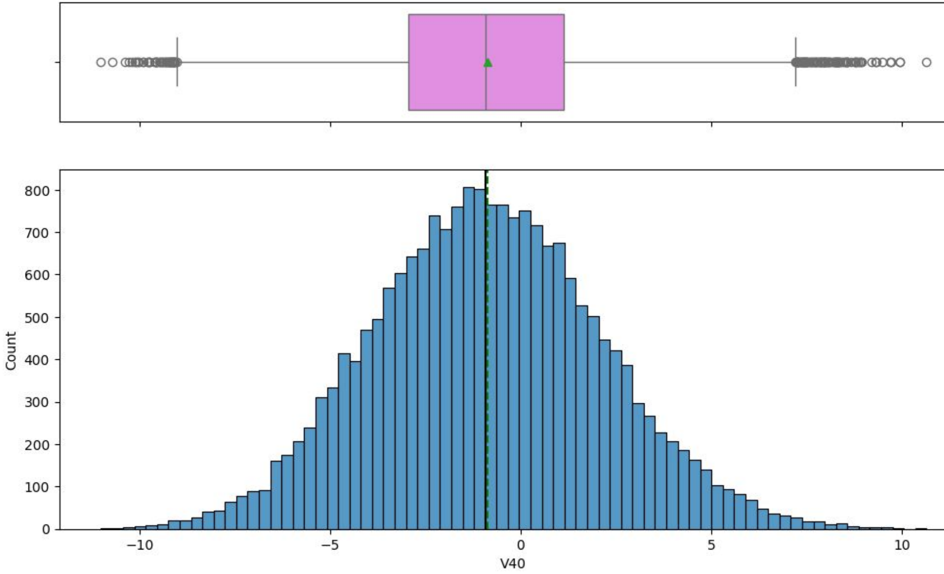




[Link to Appendix slide on data background check]

EDA Results





[Link to Appendix slide on data background check]()