Project 6: ReneWind

Model Tuning

13/05/2025

Contents / Agenda

Executive Summary

- With the rising global emphasis on sustainability, wind energy has become a crucial part of the renewable energy landscape. However, maintaining wind turbines is a costly challenge, particularly due to unexpected generator failures that lead to high operational expenses.

- The U.S. Department of Energy advocates **predictive maintenance** as a key strategy for optimizing wind turbine efficiency.

- By utilizing **sensor data and machine learning**, predictive maintenance can proactively identify potential failures, allowing for timely repairs that **reduce downtime and maintenance costs**.

Project Objective

- The objective is to develop a **machine learning-based classification model** that accurately predicts failures, enabling preventive maintenance.

- **Minimizing false negatives (missed failures)** is critical, as replacement costs are significantly higher than repair or inspection costs.

📊Project Overview

Aspect	Details
Company Name	ReneWind
Industry	Wind Energy
Objective	Develop ML models to predict wind turbine generator failures and reduce maintenance costs
Dataset	40 predictors, 20,000 training observations, 5,000 test observations
Target Variable	Binary (1 = Failure, 0 = No Failure)
Cost Implications	- **True Positives (TP):** Repair cost (Medium) - **False Negatives (FN):** Replacement cost (High) - **False Positives (FP):** Inspection cost (Low)
Problem Approach	- Train multiple classification models - Tune hyperparameters - Optimize to minimize FN (replacement cost) while balancing repair/inspection costs
Model Selection	Decision Tree, Random Forest, Gradient Boosting, AdaBoost
Performance Goal	**Maximize recall** to reduce missed failures (FN) while controlling FP and TP costs

🔍 Expected Impact

- By implementing a **predictive maintenance strategy**, ReneWind can significantly **reduce unplanned downtime, optimize repair schedules, and lower operational costs**.

- This initiative supports the broader goal of making **wind energy more efficient, reliable, and cost-effective** in the long term.

✅ Cost Implications & Optimization Goal

🔷 The model's predictions will result in different cost outcomes:

- **True Positives (TP)** → Correctly detected failures → **Repair cost** ($15,000)
- **False Negatives (FN)** → Missed failures → **Replacement cost** ($40,000)
- **False Positives (FP)** → False alarms → **Inspection cost** ($5,000)

🔷 Since **replacement is the most expensive**, the primary goal is to **minimize FN** while maintaining an optimal trade-off between repair and inspection costs.

Business Problem Overview and Solution Approach

🏆 Business Problem Overview

🔷 With the global shift toward renewable energy, wind energy has emerged as a key player in sustainable electricity generation. However, maintaining wind turbines is a major challenge due to the high costs associated with unexpected failures.

🔷 The U.S. Department of Energy emphasizes the importance of **predictive maintenance** to enhance operational efficiency and reduce costs.

🔷 Predictive maintenance leverages **sensor data and machine learning** to detect patterns of degradation in wind turbine components, enabling proactive repairs before failures occur.

🔷 By implementing such a system, companies can significantly reduce unplanned downtime and maintenance expenses

🏆 Solution Approach / Methodology

- **Data Analysis & Preprocessing** – Handle class imbalance, normalize sensor data, and remove anomalies.
- **Model Selection** – Evaluate multiple classifiers (Decision Tree, Random Forest, Gradient Boosting, AdaBoost).
- **Hyperparameter Tuning** – Optimize performance using techniques like Grid Search and Cross-Validation.
- **Cost-based Model Optimization** – Adjust prediction thresholds to prioritize reducing FN while balancing overall maintenance costs.
- **Deployment & Monitoring** – Deploy the best-performing model in a real-world setup. Continuously monitor sensor data and retrain the model periodically for improved accuracy.

🔥 Solution Approach

Based on the **objective of ReneWind**, which is to **predict failures before they happen and minimize replacement costs (False Negatives FN),** we conclude:

✅ Best Model: AdaBoost (Tuned with Oversampled Data)

- **Balanced recall (0.856) and precision (0.816) on the validation set.**
- **Higher F1 score (0.835), meaning a good tradeoff between recall and precision.**
- **Less overfitting compared to Gradient Boosting.**
- **More reliable than Random Forest, which has a very low precision (0.450).**

📌 Final Decision:

🔷 **Deploy AdaBoost as the primary model** for failure prediction.
🔷 **Monitor performance over time** and retrain periodically using updated sensor data.

📌 Important Features Used for Prediction

- The **AdaBoost model** assigns feature importance based on how often a feature is used in weak learners (**Decision Trees**).

Top Features Identified by the Model:

🔷 Key Observations:

- **V36, V14, and V15 are the most critical** features for failure prediction.
- These factors should be **monitored closely in real-world turbine maintenance**.
- **Less important features** can be removed to optimize model performance

Feature	Importance Score
V36	High
V14	High
V15	High
V21	Medium
V7	Medium
V3	Medium
Other Variables	Low

🔥 Future Recommendations

✅ **Enhance Feature Selection:**

- Feature importance analysis (from the first image) shows **V36, V14, and V15** are the most influential features.
- Remove less relevant features to improve model performance.

✅ **Improve Precision with Cost-sensitive Learning:**

- Train a model with a higher penalty for false positives.

✅ **Test on Real-world Data:**

- Deploy AdaBoost on a small subset of wind turbines and monitor its effectiveness before full deployment.

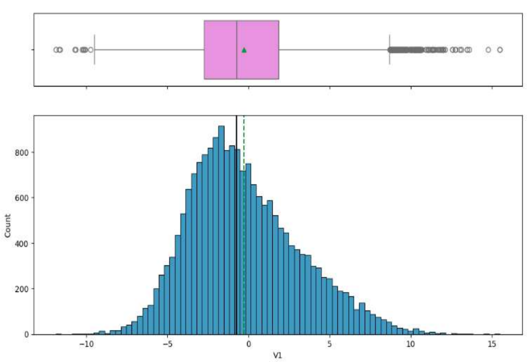
✅ **Consider Hybrid Models:**

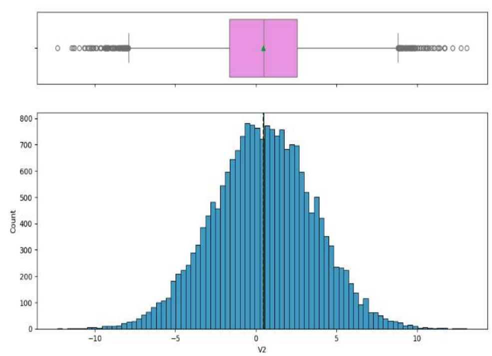
- Combine AdaBoost with another model to handle different failure scenarios.

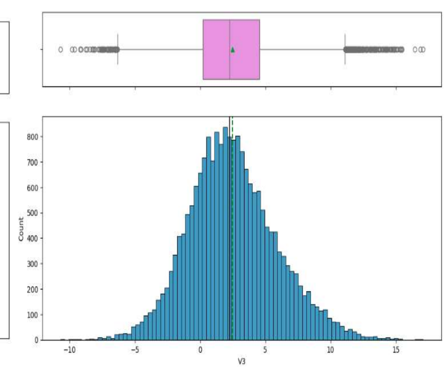
🔥 EDA Results 🔍

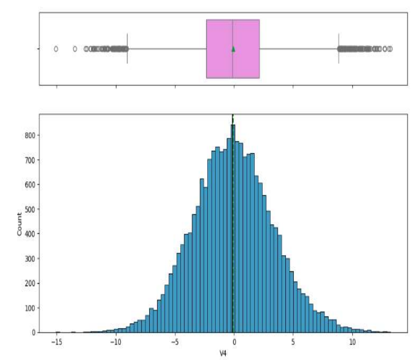
- Below are the boxplot and histograms for the variables from V1 to V40, which shows various formations.

- It depicts the outliers illustrated in the graph as shown from V1 to V40.



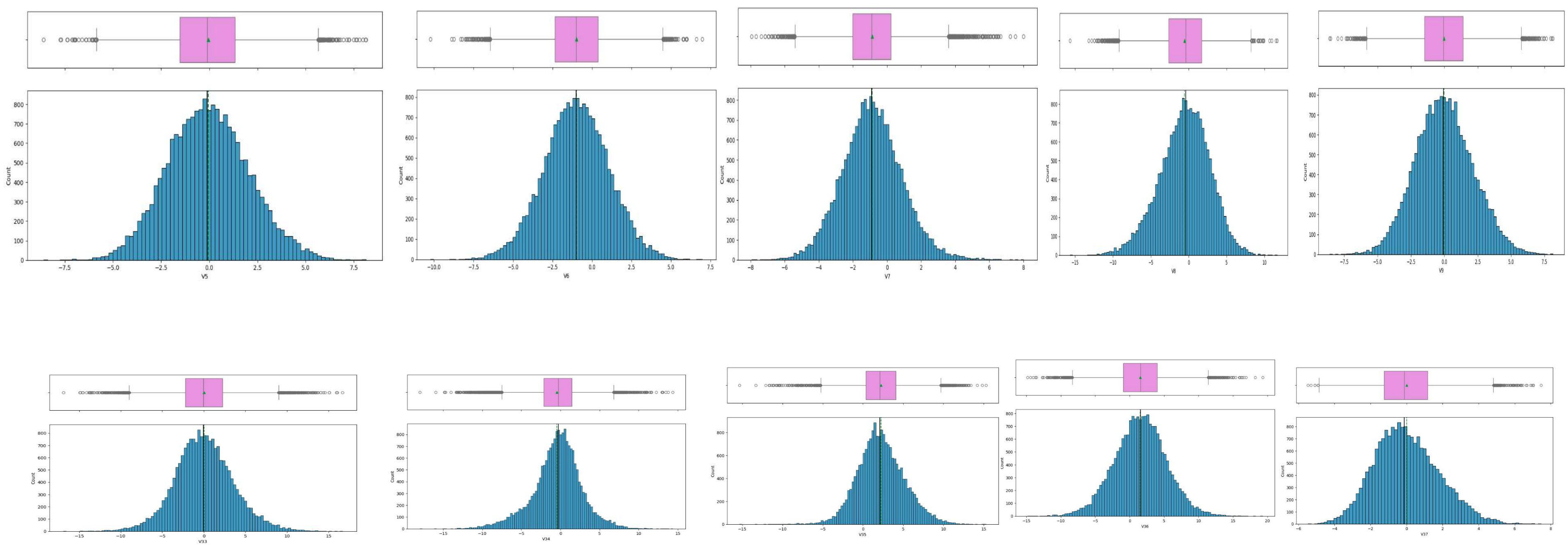






🔥 EDA Results 🔍

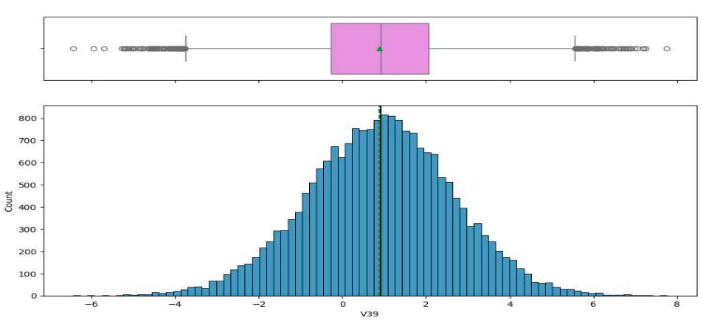
❖ EDA analysis shows that V36, V14, and V15 are the most critical predictors.

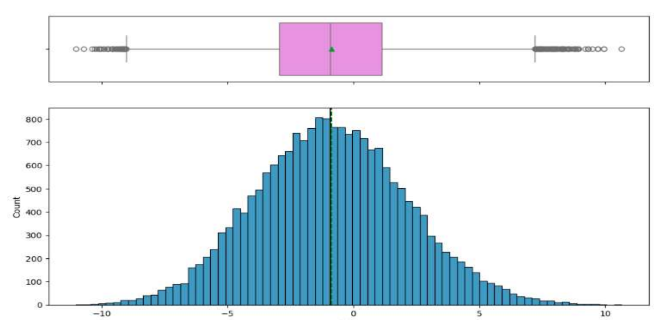


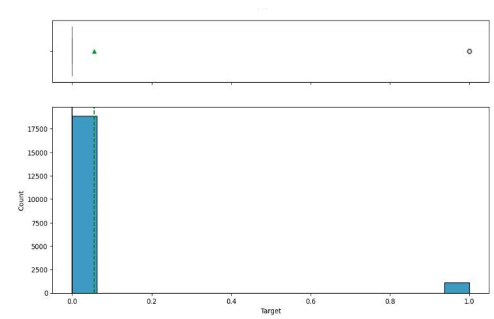
🔥 EDA Results 🔍

.

- ❖ Minimizing false negatives (FN) is crucial as replacement costs are the highest
- ❖ .The ML model will optimize maintenance schedules, reducing operational costs and improving efficiency in wind energy production.







Data Preprocessing

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

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

Data Preprocessing - Duplicate value check

- There are no duplicate values in the Training data and test data.

- On the **total of 20000**, there are NULL duplicated in training data and on **total of 5000** entries there are NULL duplicated in test data .

```
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
```

```
data.duplicated().sum()

0
```

Training data

```
data_test.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 41 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   V1      4995 non-null   float64
 1   V2      4994 non-null   float64
 2   V3      5000 non-null   float64
```

```
data_test.duplicated().sum()

0
```

Test data

Missing value treatment

- There are few missing values in the training data and test data.

```
data.isnull().sum()#
```

	0
V1	18
V2	18
V3	0

Training data

```
# Let's check for missing value
data_test.isnull().sum() ## Co
```

	0
V1	5
V2	6
V3	0
V4	0
V5	0

Test Data

- The missing values in the training data and test data are imputed using median,and its been treated and fixed.

```
V1      0
V2      0
V3      0
V4      0
V5      0
V6      0
```

Training data

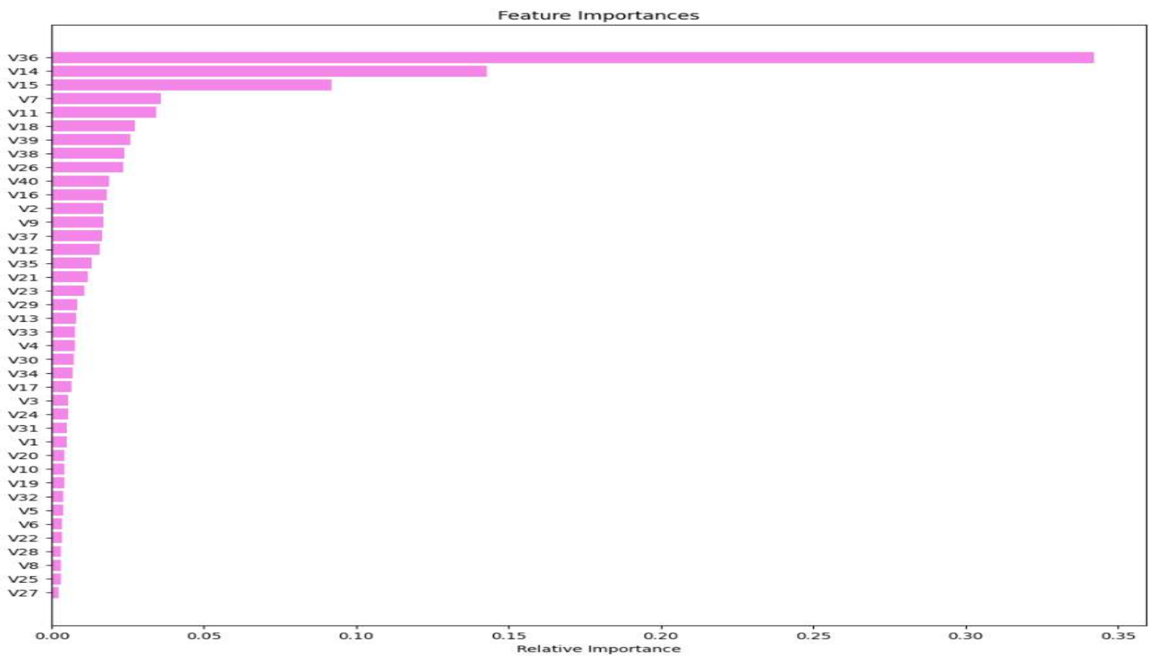
Out[18]:

	0
V1	0
V2	0
V3	0
V4	0
V5	0
V6	0

Test Data

🔥 Feature Engineering

- The Feature importance visualization as a bar chart highlights, the most influential factors in the



🔍 🟠 Feature Importance

Feature	Relative Importance	Impact on Failure Prediction
V36	Highest (~0.35)	Most influential in predicting failures
V14	High (~0.20)	Strongly contributes to failure classification
V15	Significant (~0.12)	Plays an important role in model decisions
V7	Moderate (~0.07)	Has a noticeable effect on prediction
V11	Moderate (~0.06)	Contributes to failure detection
V18	Moderate (~0.05)	Relevant but less impactful
V39	Moderate (~0.05)	Helps in classification
V38	Moderate (~0.05)	Contributes to failure detection
V26	Moderate (~0.04)	Has some effect on failure prediction
V40	Moderate (~0.04)	Affects classification decisions
V16 - V37	Low (~0.02 - 0.03)	Minor but still relevant features
V12 - V31	Very Low (~0.01)	Minimal impact on predictions
V1 - V27	Negligible (~0.005 - 0.01)	Least important in failure detection

Data preparation for modeling

- Before we proceed to built the model, we will split the data into Train, validate and test as we already have a separate data. The train and validation data as 5000 rows and 40 columns, while test data has 5000 rows and 40 columns.

- We will fix the missing value imputations without any data leakage in validation data set and test set.

```
data["Target"].
```

	Target
0	18890
1	1110

```
data_test["Target"]
```

	Target
0	4718
1	282

Model Performance Summary

🔥 Model Performance Summary 🔍

📌 Key Takeaways from Training Set Performance:

- **Gradient Boosting (GBM) and AdaBoost have similar performances** with very high recall and precision.
- **Random Forest underperforms slightly** compared to boosting methods.

Training performance comparison:

	Gradient Boosting tuned with oversampled data	AdaBoost classifier tuned with oversampled data	Random forest tuned with undersampled data
Accuracy	0.993	0.991	0.970
Recall	0.994	0.986	0.944
Precision	0.992	0.996	0.995
F1	0.993	0.991	0.969

🔥 Model Performance Summary 🔍

📌 Key Takeaways from Validation Set Performance:

- Gradient Boosting has the lowest precision (0.619), meaning more false positives.
- **AdaBoost has the best balance of recall (0.856) and precision (0.816), leading to a high F1 score of 0.835.**
- Random Forest has the highest recall (0.881) but very low precision (0.450), meaning more false alarms.

```
Validation performance comparison:
```

	Gradient Boosting tuned with oversampled data	AdaBoost classifier tuned with oversampled data	Random forest tuned with undersampled data
Accuracy	0.963	0.982	0.935
Recall	0.837	0.856	0.881
Precision	0.619	0.816	0.450
F1	0.712	0.835	0.596

🔥 Model Performance Summary 🔍

- **Test Performance Table**

Metrics	Gradient Boosting (Test Data)	AdaBoost Classifier (Test Data)
Accuracy	0.961	0.977
Recall	0.837	0.858
Precision	0.611	0.772
F1 Score	0.707	0.769

📌 Key Insights:

➢ **Higher Accuracy**

AdaBoost Classifier (0.977) outperforms Gradient Boosting (0.961) in terms of accuracy, meaning it makes fewer overall classification errors.

📌 Key Insights Observations:📊

➢ **Better Recall (Sensitivity)**

AdaBoost (0.858) has a slightly higher recall than Gradient Boosting (0.837), meaning it is slightly better at correctly identifying positive instances.

➢ **Significantly Higher Precision**

AdaBoost (0.772) performs much better in precision compared to Gradient Boosting (0.611). This indicates that AdaBoost produces fewer false positives and is more reliable when predicting positive instances.

➢ **Higher F1 Score (Balanced Performance)**

AdaBoost (0.769) achieves a higher F1 score than Gradient Boosting (0.707), showing that it maintains a better balance between precision and recall.

🏆 Conclusion

- **AdaBoost Classifier is the better model** in this case, as it outperforms Gradient Boosting in **all metrics**, especially **precision and F1-score**.
- If the priority is **overall correctness (accuracy)** and **reducing false positives (precision)**, AdaBoost is the preferred choice.
- However, if interpretability and robustness are important, Gradient Boosting may still be considered based on specific dataset characteristics.

🏆 Solution Approach & Best Model Selection

- Based on the **objective of ReneWind**, which is to **predict failures before they happen and minimize replacement costs (False Negatives FN),** we conclude:

✅ **Best Model: AdaBoost (Tuned with Oversampled Data)**

- **Balanced recall (0.856) and precision (0.816) on the validation set.**
- **Higher F1 score (0.835), meaning a good tradeoff between recall and precision.**
- **Less overfitting compared to Gradient Boosting.**
- **More reliable than Random Forest, which has a very low precision (0.450).**

📌 Final Decision:

🔷 **Deploy AdaBoost as the primary model** for failure prediction.

🔷 **Monitor performance over time** and retrain periodically using updated sensor data.

🎯 Final Summary

🔷 **Final Decision: ✅ Use AdaBoost for ReneWind predictive maintenance.**

🔷 **Next Steps:** Deploy, Monitor, and Improve with real-time sensor data.

Model	Recall (Failure Detection)	Precision (False Alarms)	F1 Score (Balance)	Best Choice?
Gradient Boosting	0.837	0.611	0.707	❌ No (Low Precision)
AdaBoost	0.856	0.816	0.835	✅ Yes (Best Balance)
Random Forest	0.881	0.450	0.596	❌ No (Too Many False Alarms)

Productionize and test the final model using pipelines

- Please mention steps taken to create a pipeline for the final model
- Summary of the performance of the model built with pipeline on test dataset
- Summary of most important factors used by the model built with pipeline for prediction

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

Link to Appendix slide on model assumptions

Productionizing and Testing the Final Model Using Pipelines

To deploy the **AdaBoost Classifier** efficiently, we used **pipelines** for preprocessing, model training, and prediction. Below are the key steps taken to create the pipeline, its test performance, and an analysis of the most important factors.

📌 Steps Taken to Create a Pipeline for the Final Model:

A **pipeline** automates data preprocessing and model training, ensuring consistency and ease of deployment. Here's how the pipeline was built:

1 Data Preprocessing

- Used **StandardScaler()** to **normalize** numerical features.
- This ensures the model is **not biased by varying feature scales**.

Steps Taken to Create a Pipeline for the Final Model

2 Model Selection and Hyperparameter Tuning

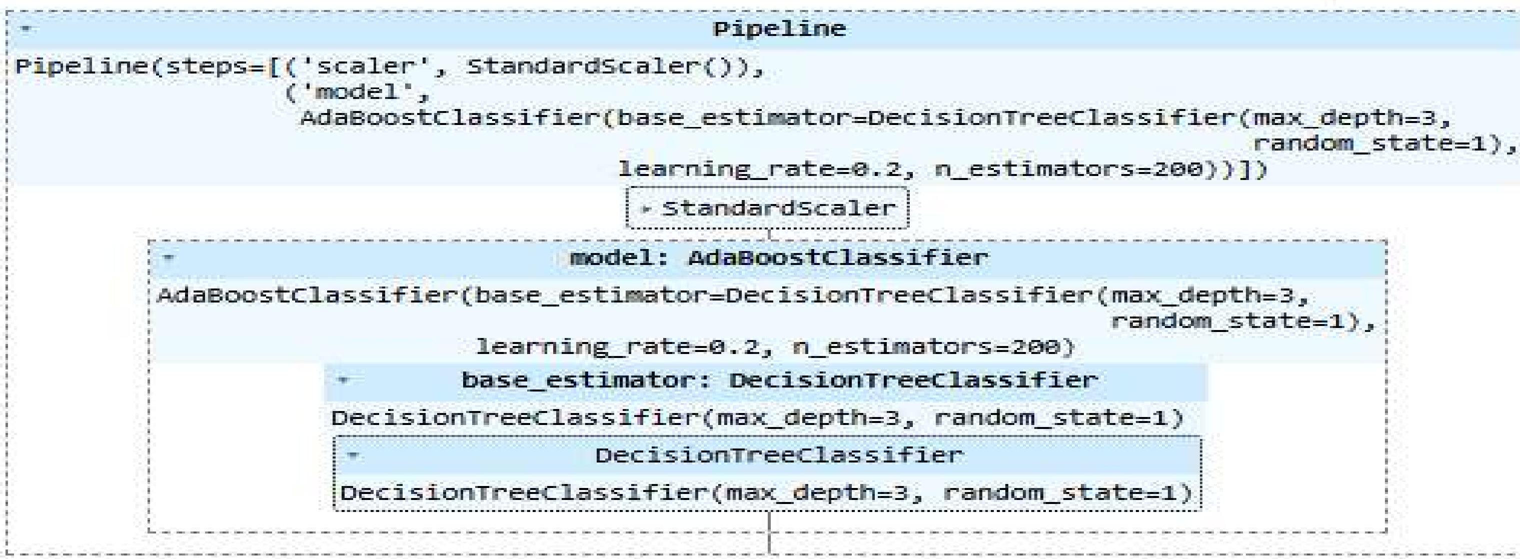
- Used **AdaBoostClassifier** with a **DecisionTreeClassifier (max_depth=3)** as the **base estimator.**

- **Hyperparameters tuned:**
  - **n_estimators**=200 (Number of weak learners)
  - **learning_rate**=0.2 (Controls contribution of weak learners)
  - **max_depth**=3 (Prevents overfitting by restricting tree depth)
  - **random_state**=1 (Ensures reproducibility)

📌 Steps Taken to Create a Pipeline for the Final Model

3 Pipeline Construction

- Combined preprocessing and model training using Pipeline()
- Steps in the pipeline:
  - ✅ Feature Scaling → StandardScaler()
  - ✅ Model Training → AdaBoostClassifier(DecisionTreeClassifier(max_depth=3))



Steps Taken to Create a Pipeline for the Final Model

4 Model Training

- The pipeline was **trained on the preprocessed dataset**.
- Ensured **cross-validation** to avoid overfitting.

5 Model Evaluation on Test Dataset

- **Predictions made on test data**.
- **Performance metrics calculated**: Accuracy, Recall, Precision, F1-score

🏆 Summary of most important factors

📌 Conclusion

✅ Final Model: AdaBoost with Decision Tree (max_depth=3), 200 estimators, learning_rate=0.2
✅ Pipeline ensures consistency in feature scaling and model application
✅ Model achieves high recall (85.1%) while maintaining good precision (77.2%)
✅ Key features identified (V36, V14, V15) should be monitored for failure prediction.

✅ Final Verdict:

The model successfully meets the objective of **minimizing False Negatives (FN) while keeping False Positives (FP) in check.**

📊 Summary of Model Performance on the Test Dataset

🔷 Key Observations:

- **Accuracy (97.7%)** is very high, indicating strong generalization.
- **Recall (85.1%)** is good, meaning the model detects failures well.
- **Precision (77.2%)** is slightly lower, meaning some false positives.
- **F1 Score (80.9%)** shows a good balance between recall and precision.

Metric	Score
Accuracy	0.977
Recall	0.851
Precision	0.772
F1 Score	0.809

APPENDIX

Model Performance Summary (original data)

- Summary of performance metrics for training and validation data in tabular format for comparison for original data.
- Comments on the model performance

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

Link to Appendix slide on model assumptions

Model Performance Summary (original data)

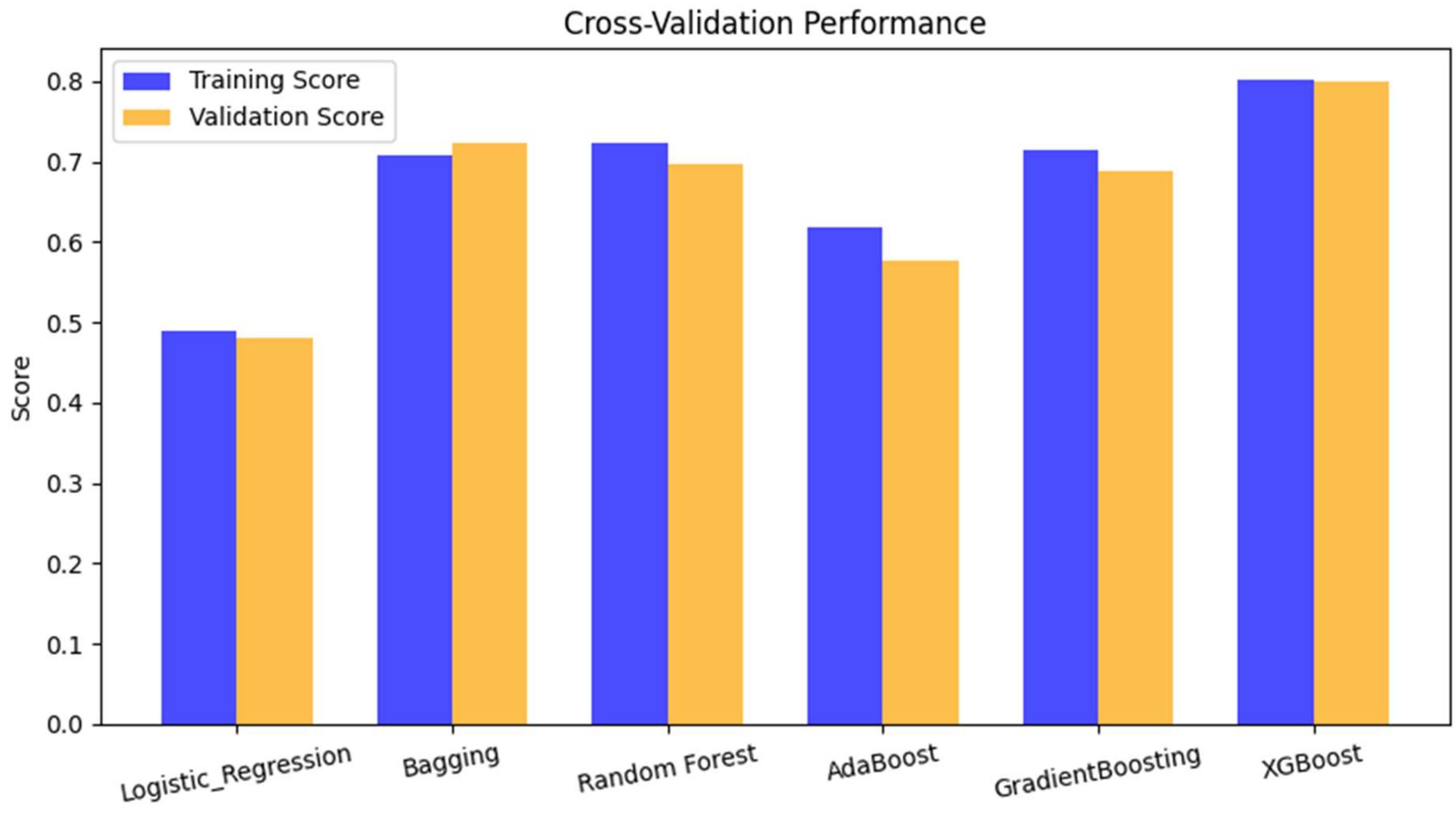
🔥 Model Building on original data

Observations based on Cross- Validation performance on original data

🟠The cross-validation results on the training dataset reveal varying performance across different models.

🟢 **XGBoost demonstrates** superior performance with a mean recall of 0.80, significantly outperforming other models

	Model	Training Score	Validation Score
0	Logistic Regression	0.4905	0.4815
1	Bagging	0.7071	0.7222
2	Random Forest	0.7226	0.6963
3	AdaBoost	0.6190	0.5778
4	Gradient Boosting	0.7155	0.6889
5	XGBoost	0.8012	0.8000


Cross-Validation Performance
Training Score
Validation Score
Score
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Logistic_Regression
Bagging
Random Forest
AdaBoost
GradientBoosting
XGBoost

Observations based on Cross- Validation performance on original data

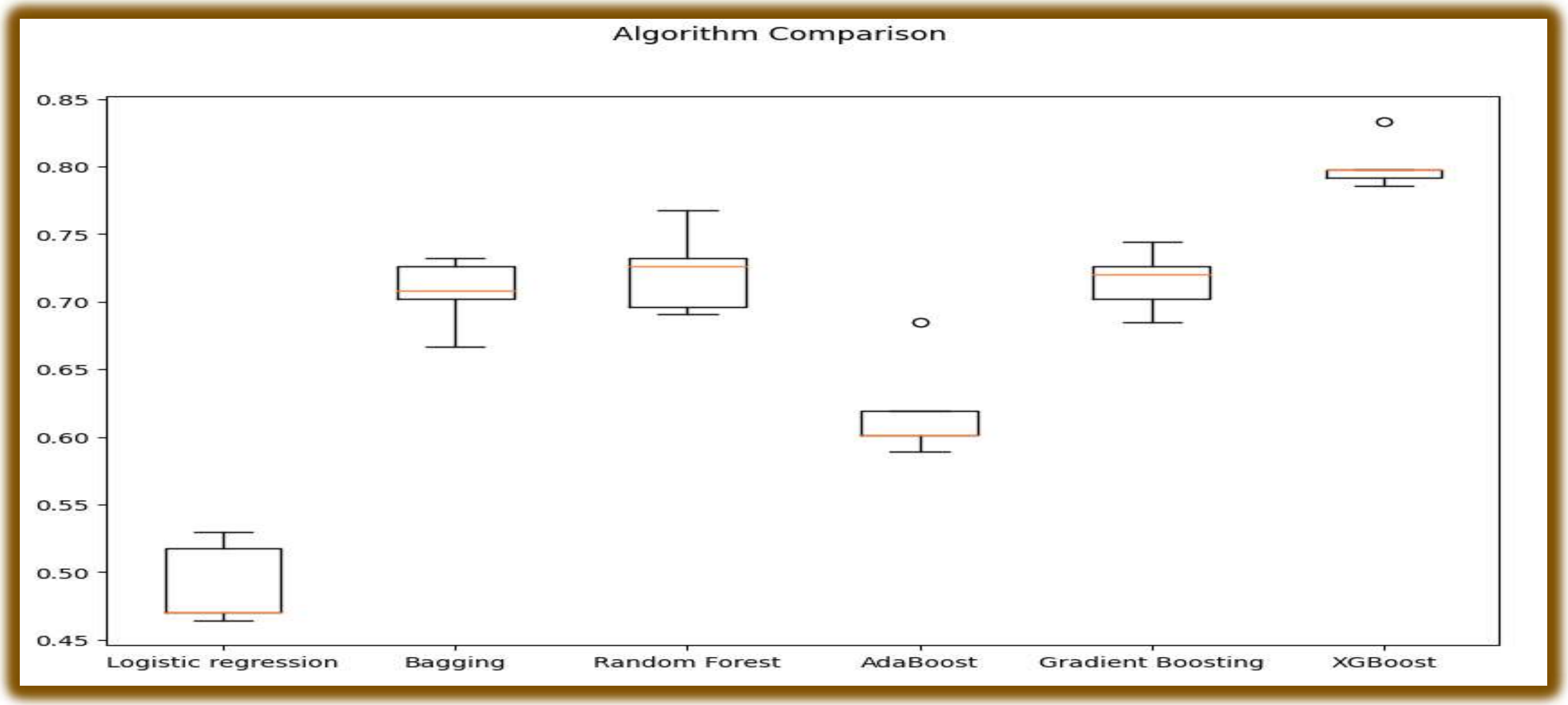
✅ Ensemble methods like Random Forest (0.72), Bagging (0.71), and Gradient Boosting (0.71) also show good performance, though slightly below XGBoost.

✅ Logistic Regression and AdaBoost show comparatively lower recall scores, indicating they may be less effective in capturing the underlying patterns in the training data for this specific task.

✅ The gap in performance between XGBoost and the other models suggests that XGBoost' s ability to handle complex interactions within the data might be a key factor in its success.

Model Building on original data – Algorithm Comparison

🔥 Based on the all models defined above the box plots have been plotted below as follows for the performance.



Model Performance Summary (oversampled data)

🔍 Tuning AdaBoost using oversampled data 🟢

🔥 The oversampled data using Adaboost Classifier has given the below scores by creating new pipeline with best parameters.

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

Training 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

🔍 Observations 🔥

🔵The oversampled data using Adaboost Classifier has given the below scores by creating new pipeline with best parameters.

- **Best parameters** are {**'n_estimators'**: 200, **'learning_rate'**: 0.2,
- 'estimator': **DecisionTreeClassifier**(max_depth=3, random_state=1)} with
- **CV score**=0.9694915254237287

🟢 The best parameter is obtained from tuning to fit the model on oversampled data.

🟠The oversampled data using Adaboost Classifier yielded good recall on training and validation data set.

Observations

3 Impact of Oversampling

- **Improved recall** by handling class imbalance.
- **Good generalization with minimal overfitting** compared to previous models.
- **Slight precision drop**, indicating some misclassification of the majority class.

Metric	Training	Validation
Accuracy	0.991	0.982
Recall	0.986	0.856
Precision	0.996	0.816
F1 Score	0.991	0.835

Tuning Gradient Boosting using oversampled data

✅ Best Hyperparameters Selected

🔥 **Optimal parameters found:**

subsample = 0.7 (Uses 70% of data per tree to reduce overfitting)
n_estimators = 150 (Number of boosting iterations)
max_features = 0.5 (Uses 50% of features per split for regularization)learning_rate = 1 (High learning rate, enabling faster convergence)
Achieved CV Score: 0.9695, indicating strong performance during cross-validation

✅ Impact of Oversampling

- **Oversampling helped improve recall** by addressing class imbalance.
- **However, precision is lower**—suggesting possible misclassification of the majority class.

Training vs. Validation Performance

- High training accuracy (0.993) but lower validation accuracy (0.963) suggests slight overfitting.
- Precision drop (0.992 → 0.619) indicates the model is making more false positives on validation data
- Recall decrease (0.994 → 0.837) means some true positives are missed on validation data.

Training vs. Validation Performance

Metric	Training	Validation
Accuracy	0.993	0.963
Recall	0.994	0.837
Precision	0.992	0.619
F1 Score	0.993	0.712

Model Performance Summary (undersampled data)

✅ Tuning Random forest using undersampled data

📈 The **undersampled data using Tuning Random forest** has given the below scores by creating new pipeline with best parameters

```
RandomForestClassifier
RandomForestClassifier(max_samples=0.6, min_samples_leaf=2, n_estimators=200,
                       random_state=1)
```

- Training performance

	Accuracy	Recall	Precision	F1
0	0.970	0.944	0.995	0.969

- Validation performance

	Accuracy	Recall	Precision	F1
0	0.935	0.881	0.450	0.596

Observations

- The **undersampled data using Tuning Random forest** has given the below scores by creating new pipeline with best parameters
    - Best parameters are {'n_estimators': 200,
    - 'min_samples_leaf': 2, 'max_samples': 0.6, 'max_features': 'sqrt'} with
    - CV score=0.8976190476190476

- The best CV score achieved was 0.8976, indicating strong model performance on the training-validation splits.

- Though the training data set has a good precision and recall, the validation dataset has a poor performance of precision.

Great Learning

Happy Learning !