Technical Report

CS5062

Machine Learning

Assignment 2

Author:
Doroteya Stoyanova

November 7, 2024

Contents

1 Battery Status Prediction of an Electric Vehicle(EV)

1.1 Introduction

1.2 Task 1.1:Classification Methods Overview

- **Naive Bayes** is a probabilistic classifier based on Bayes' theorem, assuming that each feature contributes independently to the probability of a class.**Gaussian Naive Bayes** is used when the data is normally distributed. It is assumed that the data follows a normal distribution.
- **K-Nearest Neighbors (KNN)** is a non-parametric algorithm that classifies new cases based on the majority class of their kk closest neighbours. It assumes similar data points are near each other, and the optimal value of kk, which affects accuracy, can be tuned through methods like **GridSearchCV**.
- **Ensemble Learning (EL)** combines multiple models strategically to enhance predictive performance beyond the capabilities of individual models. We will explore techniques such as **Bagging**, which reduces variance by averaging predictions from base classifiers trained on random subsets of the data, and **Random Forest**, which further improves accuracy by averaging predictions from numerous decision trees fitted on different samples. **AdaBoost** focuses on transforming weak classifiers into strong ones by iteratively giving more weight to misclassified instances, while
- **Support Vector Machine (SVM)** is an algorithm that plots data points in an n-dimensional space, where each feature represents a coordinate. It classifies data by finding the optimal hyperplane that best separates the classes.

1.3 Task 1.2: Data EDA and Preprocessing

1.3.1 Data Import and Initial Analysis

First, we import all the necessary libraries and load our dataset:

```
import pandas as pd

train_data = pd.read_csv('dataset/Train.csv')
test_data = pd.read_csv('dataset/Test.csv')


print("Training Data:")
print(train_data.head())

print("\nTesting Data:")
print(test_data.head())
```

1.3.2 Dataset Description

The dataset provided is divided into training and testing subsets and consists of recorded trip data, including the following features: specific trips, vehicle speed measured in km per hour and the battery status of the electric vehicle, indicating if the battery is active (1) or inactive (0).

1.3.3 Data Analysis

Initial analysis using pandas' descriptive statistics revealed that the data is imbalanced.

Table 1: Training and Testing Dataset Statistics

Dataset	Status	Count	Percentage
Training	0	78,773	73.56%
	1	28,312	26.44%
Total		107,085	100%
Testing	0	44,687	74.27%
	1	15,485	25.73%
Total		60,172	100%

Our data is highly skewed, with fewer than 50% active battery entries, therefore we had to think of strategies to address this. Because the data is imbalanced, two approaches could be taken: undersample and reduce the size of the majority classes to match the minority class. Alternatively, utilise synthetic data to oversample the minority class. We decided to attempt the latter.

1.3.4 Data Preprocessing

Based on our analysis, we performed the following preprocessing steps:

1. Missing values:

```
# Check for missing values for both train and test
print("Missing values in the dataset:\n",
   train_data.isnull().sum())

print("Missing values in the dataset:\n",
   test_data.isnull().sum())
```

Due to the imbalanced data, we employed Synthetic Minority Oversampling Technique (SMOTE) to balance the data. The dataset was split into features and target, then into training and test sets. SMOTE was applied only to the training data, preserving the original distribution in the validation set. Feature scaling was performed using MinMaxScaler, fitted on the balanced training data and applied to all sets. This approach maintains the integrity of the test set as unbiased representations of real-world data, preventing data leakage. The preprocessing concludes with an overview of dataset shapes and class distributions at each stage.

```
    X_train = train_data.drop('Battery_Status', axis=1)
    y_train = train_data['Battery_Status']

    # Prepare the test data
    X_test = test_data.drop('Battery_Status', axis=1)
    y_test = test_data['Battery_Status']
```

Dataset Overview			
Scaled Training Dataset			
Feature	Min	Max	Total Samples
Trip	0.00	1.00	107,085
VehicleSpeed_km_h	0.00	1.00	
Resampled Training Dataset (After SMOTE)			
Status	Count	Percentage	Total Samples
0	78,773	50.00%	157,546
1	78,773	50.00%	
Test Dataset			
Status	Count	Percentage	Total Samples
0	44,687	74.27%	60,172
1	15,485	25.73%	

Table 2: Summary of Scaled Training Dataset, Resampled Training Dataset (After SMOTE), and Test Dataset

```
    # Scale the training and test data using MinMaxScaler
    scaler = MinMaxScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    smote = SMOTE(random_state=42)
    X_train_resampled, y_train_resampled =
   smote.fit_resample(X_train_scaled, y_train)
```

1.3.5 Methods Implementation

Our implementations leverage scikit-learn for model development, with training conducted on an M1 chip processor. For hyperparameter optimization, we employ Grid Search for the K-NN and Randomized Search CV for all Ensemble methods, enabling us to fine-tune them effectively for enhanced performance. After training, we print the test results and visualise the ROC curve, the **precision-recall curve**, highlighting the balance between precision and recall, particularly focusing on the area under the curve (AP) for the minority class and lastly the confusion matrix and the F1 score.

1.3.6 Naive Bayes

To showcase the usefulness of Naive Bayes, we utilise the Gaussian variation which assumes each class follows a normal distribution. It assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

```
    from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train_resampled, y_train_resampled)
y_pred_test = gnb.predict(X_test_scaled)
y_pred_proba_test = gnb.predict_proba(X_test_scaled)[:, 1]

test_accuracy = accuracy_score(y_test, y_pred_test)
```

```
conf_matrix = confusion_matrix(y_test, y_pred_test)
class_report = classification_report(y_test, y_pred_test)

print("Gaussian Naive Bayes Results:")
print("============================")
print(f"Test Accuracy: {test_accuracy:.4f}")
print("\nClassification Report (Test Set):")
print(class_report)

print(classification_report(y_test,  y_pred_test))

plt.figure(figsize=(8, 6))
conf_matrix = confusion_matrix(y_test,  y_pred_test)
sns.heatmap(conf_matrix/np.sum(conf_matrix), annot=True,
            fmt='.2%', cmap='Blues')
plt.title(f'Confusion Matrix - Gaussian Naive Bayes - Test Set')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# ROC Curve for Test Set
fpr_test, tpr_test, treshold = roc_curve(y_test,
   y_pred_proba_test)
roc_auc_test = auc(fpr_test, tpr_test)

plt.figure(figsize=(8, 6))
plt.plot(fpr_test, tpr_test, color='purple', lw=2, label=f'ROC
   curve (AUC = {roc_auc_test:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Gaussian Naive Bayes (Test Set)')
plt.legend(loc="lower right")
plt.show()

# Precision-Recall Curve
plt.figure(figsize=(8, 6))
precision, recall, _ = precision_recall_curve(y_test,
   y_pred_proba_test)
average_precision = average_precision_score(y_test,
   y_pred_proba_test)
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2,
   color='b')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title(f'Precision-Recall Curve (Test Set):
   AP={average_precision:.2f}')
plt.show()
```

1.3.7 KNN

Another method we attempted was K-Nearest Neighbors (KNN) classifier with the Grid Search Cross-Validation to find the optimal number of neighbors (k), followed by the aforementioned model evaluation on the test set. We approached the selection of the optimal k value by defining a range of values that intentionally does not start from 1, as a k value of 1 can lead to potential overfitting by being overly sensitive to noise in the data. Instead, we began with slightly higher values to strike a balance between capturing neighbourhood structure and maintaining generalizability. This approach reduces the risk of overfitting and helps the model better capture the underlying patterns within the data.

```
k_values = list(range(3, 100))
param_grid = {'n_neighbors': k_values}

knn = KNeighborsClassifier()
cv = StratifiedKFold(n_splits=5)

grid_search = GridSearchCV(estimator=knn, param_grid=param_grid,
                           scoring='accuracy', cv=cv, verbose=1,
   n_jobs=-1)
grid_search.fit(X_train_resampled, y_train_resampled)


best_k = grid_search.best_params_['n_neighbors']
best_knn = grid_search.best_estimator_

print(f"\nBest k value: {best_k}")
print(f"Best accuracy from Grid Search:
   {grid_search.best_score_:.4f}")


y_pred_knn_test = best_knn.predict(X_test_scaled)
y_pred_proba_knn_test = best_knn.predict_proba(X_test_scaled)[:,
   1]


test_accuracy = accuracy_score(y_test, y_pred_knn_test)
print(f"Test Accuracy: {test_accuracy:.4f}")


print("\nClassification Report (Test Set):")
print(classification_report(y_test, y_pred_knn_test))


plt.figure(figsize=(8, 6))
conf_matrix = confusion_matrix(y_test, y_pred_knn_test)
sns.heatmap(conf_matrix / np.sum(conf_matrix), annot=True,
   fmt='.2%', cmap='Blues')
plt.title(f'Confusion Matrix - KNN (k={best_k}) - Test Set')
plt.ylabel('True Label')
```

```
plt.xlabel('Predicted Label')
plt.show()


fpr, tpr, treshold = roc_curve(y_test, y_pred_proba_knn_test)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
   (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'ROC Curve - KNN (k={best_k}) - Test Set')
plt.legend(loc="lower right")
plt.show()

plt.figure(figsize=(8, 6))
precision, recall, _ = precision_recall_curve(y_test,
   y_pred_proba_test)
average_precision = average_precision_score(y_test,
   y_pred_proba_test)
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2,
   color='b')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title(f'Precision-Recall Curve (Test Set):
   AP={average_precision:.2f}')
plt.show()

print("\nFinal Results for Best KNN Model:")
print("================================")
print(f"Best k value: {best_k}")
print(f"Test Accuracy: {test_accuracy:.4f}")
```

1.3.8 Ensemble Methods

In the Task 1.1, we specified that we will implement Bagging, Random Forest and AdaBoosting.We decided to use RandomizedSearchCV instead of GridSearchCV. Unlike GridSearchCV, which exhaustively tests every parameter combination, RandomizedSearchCV randomly samples a defined number of combinations, providing a diverse exploration of the parameter space without an extensive computational cost. This approach allows us to effectively test various configurations, such as the number of estimators, maximum samples, and maximum features. Once the best parameters are identified, the model is trained on the resampled training data and its performance is evaluated on the test set.

- Bagging Method The grid parameters we selected for the Bagging Method are in

Table 3.

Parameter	Value	Explanation
estimator	BaggingClassifier(randomstate)	The base estimator for bagging. Random state ensures reproducibility.
param_distributions	n_estimators: [10, 50, 100, 200, 300]	Number of base estimators. Higher values generally increase performance but also computational cost.
	max_samples: [0.1, 0.5, 0.7, 0.9]	Fraction of samples to draw for training each base estimator. Affects diversity of the ensemble.
	max_features: [0.1, 0.5, 0.7, 0.9]	Fraction of features to draw for training each base estimator. Influences feature diversity.
scoring	'balanced_accuracy'	Metric used for evaluation. Balanced accuracy is useful for imbalanced datasets.
n_iter	10	Number of parameter settings sampled. Balances exploration and computational time.
cv	5	Number of cross-validation splits. Helps in robust performance estimation.
verbose	1	Controls the verbosity of the search process.
n_jobs	-1	Number of jobs to run in parallel. -1 means using all processors.

Table 3: RandomizedSearchCV Configuration for Bagging Classifier with Explanations

```
    param_grid = {
    'n_estimators': [10, 50, 100, 200, 300],
    'max_samples': [0.1, 0.5, 0.7, 0.9],
    'max_features': [0.1, 0.5, 0.7, 0.9]
}


bagging = BaggingClassifier(random_state=42)

grid_search = RandomizedSearchCV(
    estimator=bagging,
    param_distributions=param_grid,
    scoring='balanced_accuracy',
    n_iter=10,
    cv=5,
    verbose=1,
    n_jobs=-1
```

```
)


grid_search.fit(X_train_resampled, y_train_resampled)


best_params = grid_search.best_params_
best_bagging = grid_search.best_estimator_
print("\nBest parameters:", best_params)
print(f"Best accuracy from RandomGrid Search:
   {grid_search.best_score_:.4f}")


y_pred_bagging_test = best_bagging.predict(X_test_scaled)
y_pred_proba_bagging_test =
   best_bagging.predict_proba(X_test_scaled)[:, 1]
```

- Random Forest Classifier The table below summarizes the hyperparameters we considered and their corresponding options.The scoring criterion, set to balanced_accuracy, accounts for our class imbalances, ensuring fair performance evaluation (Table 4).:

Hyperparameter	**Options**	**Explanation**
`n_estimators`	[10, 50, 100, 200, 300]	Specifies the number of trees in the forest. Higher values can improve accuracy but increase runtime.
`max_features`	[`'auto'`, `'sqrt'`]	Controls the number of features considered at each split. `'auto'` uses all features, while `'sqrt'` uses a subset.
`max_depth`	[10, 20, 30, 40, 50]	Sets the maximum depth of each tree. Deeper trees can capture more patterns but risk overfitting.
`min_samples_split`	[2, 5, 10]	Minimum number of samples required to split a node. Higher values reduce tree complexity.
`min_samples_leaf`	[1, 2, 4]	Minimum number of samples required for each leaf node. Higher values can reduce overfitting in high-dimensional data.
`bootstrap`	[`True`, `False`]	Determines whether bootstrap samples are used when building trees.

Table 4: Parameter grid for `RandomForestClassifier` hyperparameter tuning.

```
param_grid_rf = {
    'n_estimators': [10, 50, 100, 200, 300],
    'max_features': ['auto', 'sqrt'],
    'max_depth': [ 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
```

```
}


random_forest = RandomForestClassifier(random_state=42)


grid_search_rf = RandomizedSearchCV(
    estimator=random_forest,
    param_distributions=param_grid_rf,
    scoring='balanced_accuracy',
    n_iter=10,
    cv=5,
    verbose=1,
    n_jobs=-1
)


grid_search_rf.fit(X_train_resampled, y_train_resampled)

best_params_rf = grid_search_rf.best_params_
best_rf = grid_search_rf.best_estimator_
print("\nBest parameters for Random Forest:", best_params_rf)
print(f"Best accuracy from Randomized Search:
   {grid_search_rf.best_score_:.4f}")

t
y_pred_rf_test = best_rf.predict(X_test_scaled)
y_pred_proba_rf_test =
   best_rf.predict_proba(X_test_scaled)[:, 1]


test_accuracy_rf = accuracy_score(y_test, y_pred_rf_test)
print(f"Test Accuracy: {test_accuracy_rf:.4f}")

print("\nClassification Report (Test Set):")
print(classification_report(y_test, y_pred_rf_test))
```

- Ada boost The main difference in our approach to building an AdaBoost ensemble is defining a simple yet effective base estimator, such as `DecisionTreeClassifier`. This decision tree serves as the "weak learner" in the AdaBoost model, with parameters chosen to manage complexity and enhance generalizability. Specifically, we set a maximum tree depth, limiting each tree to three levels to prevent overfitting while still capturing essential patterns in the data. The min_samples_split=5 parameter requires that a node must contain at least five samples to split, further refining the model's focus on meaningful patterns without becoming overly specific(Tables 5 and 6.

Component	Parameter	Description
Base Estimator	max_depth=3	Limits tree depth to prevent overfitting
	min_samples_split=5	Ensures meaningful splits
	random_state=42	Ensures reproducibility
AdaBoostClassifier	base_estimator	Uses the defined DecisionTreeClassifier

Table 5: AdaBoost Ensemble Configuration

Table 6: AdaBoost Hyperparameter Grid and RandomizedSearchCV Setup

Parameter	Values
`n_estimators`	[10, 50, 100, 200]
`learning_rate`	[0.01, 0.1, 0.5, 1.0, 1.5]
`cv`	5
`n_iter`	10
`scoring`	balanced_accuracy
`base_estimator`	DecisionTree (max_depth=3, min_samples_split=5)
`random_state`	42

1.3.9 SVM

We used LinearSVC to make the code work fast and efficiently. As in the Ensemble Methods, we used RandomSearch, and used the following parameters to achieve our results:

Component	Parameter	Values/Description
LinearSVC	C	[0.1, 1, 10, 100, 1000]
	loss	['hinge', 'squared_hinge']
	max_iter	[1000, 2000]
RandomizedSearchCV	estimator	LinearSVC(random_state=42)
	scoring	'balanced_accuracy'
	n_iter	20
	cv	5
	n_jobs	-1 (use all processors)

Table 7: Linear SVM and RandomizedSearchCV Configuration

```
base_svm = LinearSVC(random_state=42)

param_grid_svm = {
    'C': [0.1, 1, 10, 100, 1000],
    'loss': ['hinge', 'squared_hinge'],
    'max_iter': [1000, 2000],
}
```

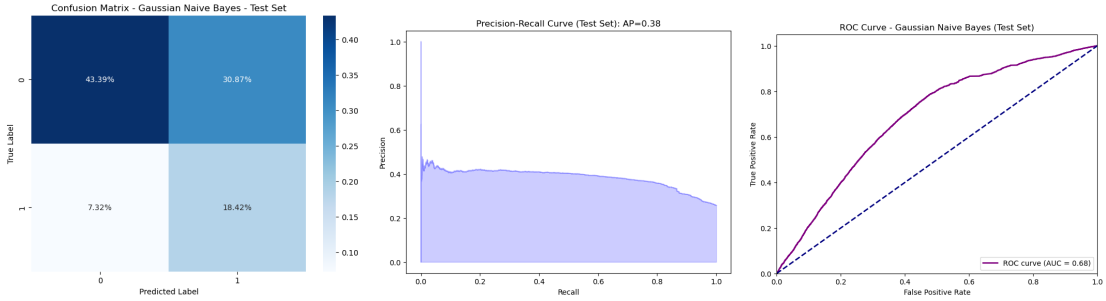
```
grid_search_svm = RandomizedSearchCV(
    estimator=base_svm,
    param_distributions=param_grid_svm,
    scoring='balanced_accuracy',
    n_iter=20,
    cv=5,
    verbose=1,
    n_jobs=-1,
    random_state=42
)

grid_search_svm.fit(X_train_resampled, y_train_resampled)

best_params_svm = grid_search_svm.best_params_
best_svm = grid_search_svm.best_estimator_

print("\nBest parameters for Linear SVM:", best_params_svm)
print(f"Best cross-validated balanced accuracy:
   {grid_search_svm.best_score_:.4f}")


y_pred_svm_test = best_svm.predict(X_test_scaled)
y_pred_proba_svm_test = best_svm.decision_function(X_test_scaled)


test_accuracy_svm = accuracy_score(y_test, y_pred_svm_test)
print(f"Test Accuracy: {test_accuracy_svm:.4f}")


print("\nClassification Report (Test Set):")
print(classification_report(y_test, y_pred_svm_test))
```

1.4 Task 2: Model Performance Evaluation

1.4.1 Gaussian Naive Bayes

Table 8: Gaussian Naive Bayes Classification Results (Test Accuracy: 0.6181)

Class/Metric	**Precision**	**Recall**	**F1-score**	**Support**
Class 0	0.86	0.58	0.69	44,687
Class 1	0.37	0.72	0.49	15,485
Accuracy		0.62		60,172
Macro Avg	0.61	0.65	0.59	60,172
Weighted Avg	0.73	0.62	0.64	60,172



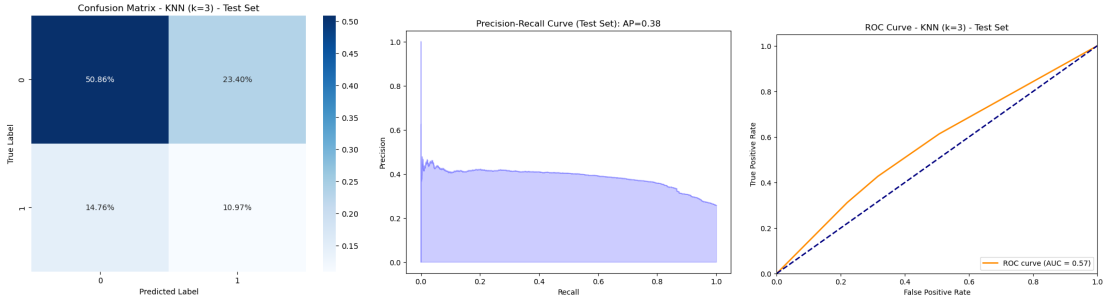
(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 1: Performance Metrics for Gaussian Naive Bayes

1.4.2 K-Nearest Neighbors (KNN)

Table 9: K-Nearest Neighbors Classification Results (k=3, Test Accuracy: 0.6183)

Class/Metric	Precision	Recall	F1-score	Support
Class 0	0.78	0.68	0.73	44,687
Class 1	0.32	0.43	0.37	15,485
Accuracy		0.62		60,172
Macro Avg	0.55	0.56	0.55	60,172
Weighted Avg	0.66	0.62	0.63	60,172



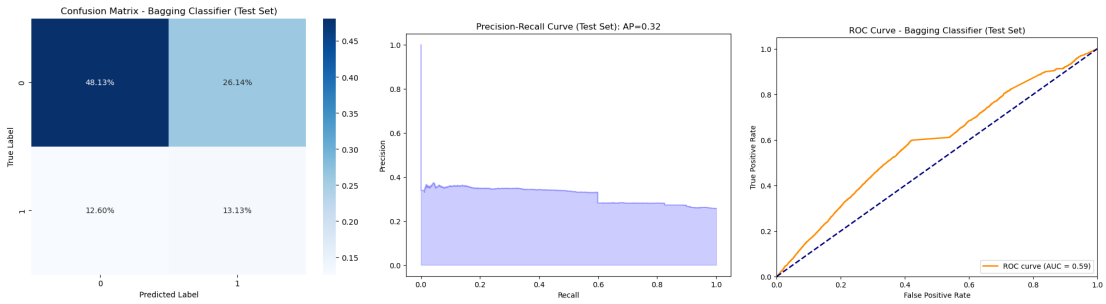
(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 2: Performance Metrics for KNN

1.4.3 Bagging Classifier

Table 10: Bagging Classifier Results (Test Accuracy: 0.6126)

Class/Metric	Precision	Recall	F1-score	Support
Class 0	0.79	0.65	0.71	44,687
Class 1	0.33	0.51	0.40	15,485
Accuracy		0.61		60,172
Macro Avg	0.56	0.58	0.56	60,172
Weighted Avg	0.67	0.61	0.63	60,172



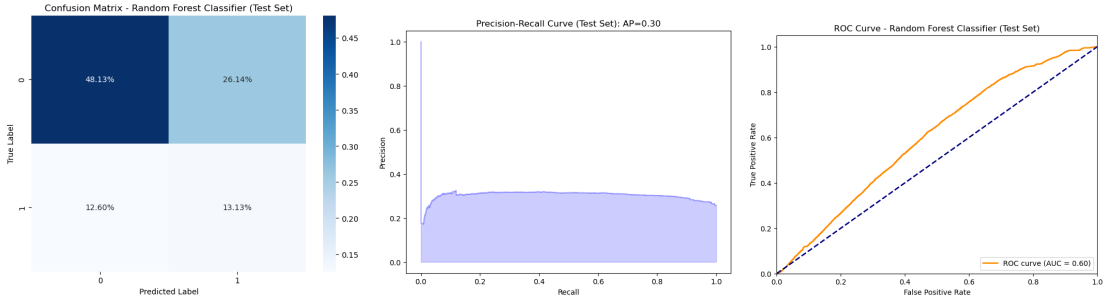
(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 3: Performance Metrics for Bagging Classifier

1.4.4 Random Forest Classifier

Table 11: Random Forest Classifier Results (Test Accuracy: 0.57)

Class/Metric	Precision	Recall	F1-score	Support
Class 0	0.79	0.56	0.66	44,687
Class 1	0.31	0.57	0.41	15,485
Accuracy		0.57		60,172
Macro Avg	0.55	0.57	0.53	60,172
Weighted Avg	0.67	0.57	0.59	60,172



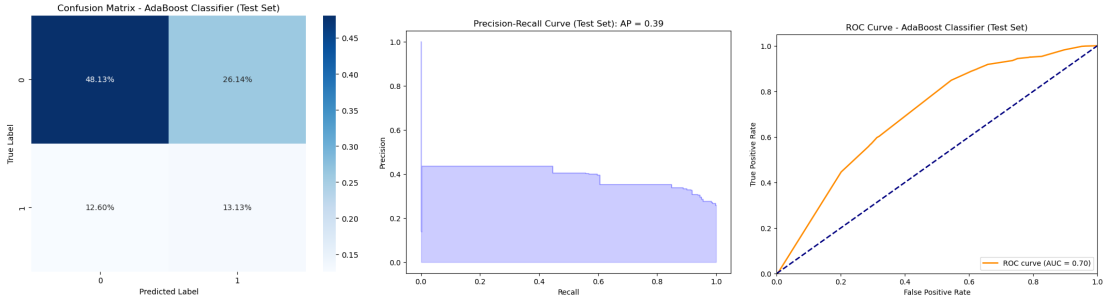
(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 4: Performance Metrics for Random Forest Classifier

1.4.5 Ada boost

Table 12: AdaBoost Classifier Results (Test Accuracy: 0.66)

Class/Metric	Precision	Recall	F1-score	Support
Class 0	0.83	0.68	0.75	44,687
Class 1	0.39	0.61	0.48	15,485
Accuracy		0.66		60,172
Macro Avg	0.61	0.64	0.61	60,172
Weighted Avg	0.72	0.66	0.68	60,172



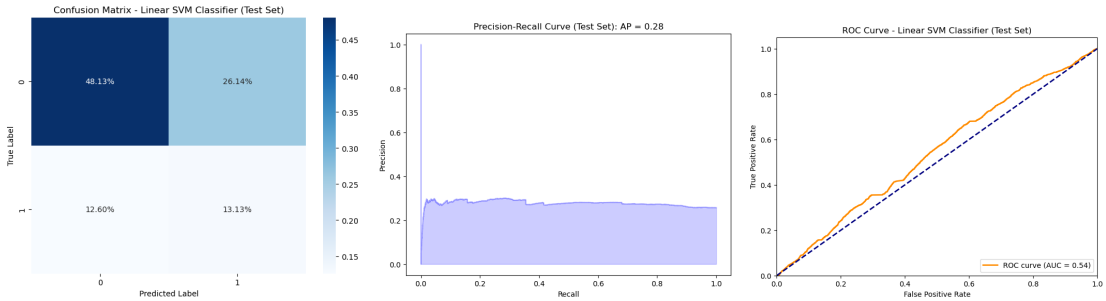
(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 5: Performance Metrics for Ada Booster

1.4.6 SVM(Linear SVC)

Table 13: SVM Classifier Results (Test Accuracy: 0.74)

Class/Metric	Precision	Recall	F1-score	Support
Class 0	0.74	1.00	0.85	44,687
Class 1	0.00	0.00	0.00	15,485
Accuracy		0.74		60,172
Macro Avg	0.37	0.50	0.43	60,172
Weighted Avg	0.55	0.74	0.63	60,172



(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 6: Performance Metrics for Linear SVM

1.5 Task 3:Analysis of Results

In analyzing the performance of the six traditional machine learning models for EV battery status prediction in Table 14, significant variations in effectiveness emerged across different evaluation metrics. The Linear SVM achieved the highest numerical accuracy at 0.74, but this figure masks a critical flaw - the model completely failed to predict the minority class (Class 1), essentially defaulting to majority class predictions. This highlights the importance of looking beyond simple accuracy metrics in imbalanced classification tasks.

AdaBoost emerged as the most practically effective model, demonstrating a test accuracy of **0.66** with notably balanced performance across both classes, however, the

Table 14: Model Performance Comparison

Model	Accuracy	Macro Avg	F1 Class 0	F1 Class 1
SVM (Linear)	0.74	0.43	0.85	0.00
AdaBoost	0.66	0.61	0.75	0.48
KNN (k=3)	0.62	0.55	0.73	0.37
Gaussian NB	0.62	0.59	0.69	0.49
Bagging	0.61	0.56	0.71	0.40
Random Forest	0.57	0.53	0.66	0.41

cross-validation revealed a potential overfitting. Its success can be attributed to its adaptive learning approach, achieving F1-scores of 0.75 and 0.48 for Classes 0 and 1 respectively. The model's ability to maintain good precision (0.83 for Class 0, 0.39 for Class 1) and recall (0.68 for Class 0, 0.61 for Class 1) across both classes demonstrates its good handling of the class imbalance problem.

The KNN and Gaussian Naive Bayes models showed similar overall accuracy (0.62), but with different strengths. KNN demonstrated a moderate balance between classes, while Gaussian Naive Bayes showed particularly strong recall (0.72) for the minority class, though at the cost of lower precision. The Bagging Classifier performed similarly with an accuracy of 0.61, showing consistent but unremarkable results across metrics. Surprisingly, the Random Forest model showed the lowest accuracy (0.57) despite its ensemble nature, suggesting potential issues with parameter optimization. One thing to note is that KNN also appears to be overfitting.

1.5.1 Selected Model:Adaboost

Taking a deeper look at AdaBoost's application, its success can be attributed to its effective use of vehicle speed as a predictive feature. The model builds an ensemble of simple decision trees (max _depth=3) that capture different aspects of the speed-battery relationship. Higher speeds typically correlate with discharge patterns due to increased power consumption, while lower speeds might indicate charging opportunities through regenerative braking. The boosting process adaptively focuses on difficult-to-classify instances, particularly beneficial for minority class predictions.

Additional data collection, particularly for minority class scenarios, could enhance model performance. The analysis ultimately demonstrates that in real-world applications like battery status prediction, the true value of a model lies not in raw accuracy but in its ability to handle class imbalances and provide reliable predictions across all scenarios.

2 Battery Status of EV utilizing RNN domain

2.1 Task 4:RNN Development

2.1.1 RNN models

- The Simple RNN processes sequential data by maintaining a hidden state that updates with each time step. It combines the current input and the previous hidden state to produce an output. However, it struggles with long-term dependencies due

to the vanishing gradient problem, where the gradients used in backpropagation diminish over time, making it hard to learn from distant inputs.

- Long Short-Term Memory (LSTM) networks enhance Simple RNNs by introducing memory cells and three gates: input, forget, and output gates. These gates allow LSTMs to selectively remember or forget information, enabling them to manage long-term dependencies more effectively and mitigate the vanishing gradient problem.
- Gated Recurrent Units (GRUs) simplify LSTMs by combining the forget and input gates into a single update gate and merging the cell state with the hidden state. GRUs are computationally more efficient while still capturing essential temporal dependencies.
- Bi-directional RNNs (BRNNs) consist of two RNNs: one processes the sequence from start to end, while the other processes it backward. This dual approach captures context from both past and future states, however more computationally expensive than GRU or the Simple RNN.

2.1.2 4.a Data Preprocessing

We import the data as we did in the previous task.

```
train_data = pd.read_csv('dataset/Train.csv')
test_data = pd.read_csv('dataset/Test.csv')


print("Training Data:")
print(train_data.head())

print("\nTesting Data:")
print(test_data.head())
```

2.1.3 4.b Data Preprocessing for RNNs

First of all,the RNNs usually need 3D shape data and sequences. RNNs, and are designed to process sequential data. In this case, the goal is to predict the battery status based on historical vehicle speed, which involves capturing the temporal patterns in the data. Traditional machine learning models such as Decision Trees or Linear Regression expect each sample to be independent, whereas RNNs require sequences of data to understand the temporal relationship between them.

In our case, vehicle speed and battery status are both time-dependent. For example, the vehicle speed at a given time step could have a relationship with the speed at previous time steps, which could influence the prediction of battery status. By converting the data into sequences, we ensure that the model can learn these temporal dependencies effectively.

The create_sequences function uses a sliding window approach to generate the sequences. This is a common technique used to prepare time series data for RNNs. For each unique trip (identified by Trip), the function generates a sequence of data for the specified sequence_length (20 time steps in this case). The sequence is essentially a fixed-length

chunk of the time series, which allows the RNN to look at the previous sequence_length time steps when making a prediction.

The reason this sliding window technique is used is that RNNs are trained to remember patterns in previous time steps. For example, if a vehicle's speed increases steadily for a period, it might impact the battery's future state. By feeding in sequences, the model can learn these long-range dependencies between vehicle speed and battery status. Each sequence is accompanied by a target value, which is the battery status at the time step following the sequence (i.e., predicting the battery status after observing the past sequence_length time steps).

RNNs require data in a 3D format: [samples, time steps, features], where:

- **Samples**: Individual sequences,
- **Time steps**: The number of time steps in each sequence (20 in this case),
- **Features**: The values at each time step (e.g., vehicle speed).

For example, if we have 100 sequences of 20-time steps with 1 feature, the shape would be $(100, 20, 1)$, allowing the RNN to process the sequences and learn temporal dependencies.

Class imbalance can negatively impact model performance, especially when certain classes, such as "low battery", are underrepresented. To address this, **BorderlineSMOTE** is applied to oversample the minority class by generating synthetic data points. Since SMOTE operates on 2D data, the sequences are initially flattened from a 3D shape (samples, time steps, features) into 2D (samples, features). After applying SMOTE, the data is reshaped back into 3D to preserve the temporal structure essential for the RNN.

```
    import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from imblearn.over_sampling import BorderlineSMOTE



X_train = train_data.drop('Battery_Status', axis=1)
y_train = train_data['Battery_Status']

X_test = test_data.drop('Battery_Status', axis=1)
y_test = test_data['Battery_Status']


scaler = MinMaxScaler()
sequence_length = 20


X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

X_train_df = pd.DataFrame(X_train_scaled,
   columns=X_train.columns)
X_train_df['Trip'] = train_data['Trip'].values
X_train_df['Battery_Status'] = y_train.values
```

```
X_test_df = pd.DataFrame(X_test_scaled, columns=X_test.columns)
X_test_df['Trip'] = test_data['Trip'].values
X_test_df['Battery_Status'] = y_test.values

# Create sequences
def create_sequences(df, sequence_length):
    X_seq = []
    y_seq = []
    for trip_id in df['Trip'].unique():
        trip_mask = df['Trip'] == trip_id
        trip_speed = df[trip_mask]['VehicleSpeed_km_h_'].values
        trip_battery = df[trip_mask]['Battery_Status'].values

        for i in range(sequence_length, len(trip_speed)):
            X_seq.append(trip_speed[i-sequence_length:i])
            y_seq.append(trip_battery[i])

    return np.array(X_seq), np.array(y_seq)

# Sequences for the train and test
X_train_seq, y_train_seq = create_sequences(X_train_df,
   sequence_length)
X_test_seq, y_test_seq = create_sequences(X_test_df,
   sequence_length)

# Reshape X sequences to be [samples, time steps, features]
X_train_seq = X_train_seq.reshape((X_train_seq.shape[0],
   X_train_seq.shape[1], 1))
if X_test_seq.size > 0:
    X_test_seq = X_test_seq.reshape((X_test_seq.shape[0],
   X_test_seq.shape[1], 1))

# Apply BorderlineSMOTE to the training sequences (flatten the
   3D array to 2D)
borderline_smote = BorderlineSMOTE(random_state=42,
   kind='borderline-1')

# flatten for SMOTE
X_train_seq_flat = X_train_seq.reshape(X_train_seq.shape[0], -1)

# apply SMOTE on the flattened data
X_train_seq_balanced_flat, y_train_seq_balanced =
   borderline_smote.fit_resample(X_train_seq_flat, y_train_seq)

# reshape to 3d
X_train_seq_balanced = X_train_seq_balanced_flat.reshape(
    X_train_seq_balanced_flat.shape[0], sequence_length, 1
)
```

Sequence Shapes	
X_train	(153182, 20, 1)
y_train	(153182,)
X_test	(58872, 20, 1)
y_test	(58872,)
Class Distributions	
Training set (after balancing)	
Class 0	50.0%
Class 1	50.0%
Test set	
Class 0	74.18%
Class 1	25.82%

Table 15: Sequence shapes and class distributions for the dataset

2.1.4 Task 4.c-d Implementation

Our implementation encompassed all previously described RNN models. Upon thorough data analysis, we identified overfitting as a critical challenge, particularly when dealing with highly imbalanced datasets. To address this issue, we strategically incorporated dropout layers between each RNN layer. This approach proved effective in mitigating overfitting and significantly enhancing the recall of the minority class (label "1"), which was notably underrepresented in the test dataset.

We introduced dropout layers with a rate of 0.2 following each RNN layer. This technique improved the model's generalization capabilities by randomly deactivating neurons during the training process, thereby reducing the model's reliance on any specific set of features.

We selected the Adam algorithm for optimisation due to its superior efficiency in adapting the learning rate and its ability to navigate around local minima, offering notable advantages over traditional Stochastic Gradient Descent (SGD).

The architecture of our model was carefully designed to balance complexity and performance. All layers of the models, including the initial layer, utilize the **Rectified Linear Unit (ReLU)** activation function. Following the RNN layers, we introduced a dense layer comprising 128 units with ReLU activation, serving as a fully connected layer to further process the extracted features.

The model culminates in an output layer consisting of a single unit with sigmoid activation, optimized for binary classification tasks. We compiled the model using the aforementioned Adam optimizer and binary cross-entropy as the loss function (See Figure 7).

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional)	(None, 20, 100)	20,800
dropout_12 (Dropout)	(None, 20, 100)	0
bidirectional_1 (Bidirectional)	(None, 20, 100)	60,400
dropout_13 (Dropout)	(None, 20, 100)	0
bidirectional_2 (Bidirectional)	(None, 100)	60,400
dropout_14 (Dropout)	(None, 100)	0
dense_5 (Dense)	(None, 128)	12,928
dense_6 (Dense)	(None, 1)	129

(a) Bidirectional Architecture

Layer (type)	Output Shape	Param #
gru_4 (GRU)	(None, 20, 50)	7,950
dropout_9 (Dropout)	(None, 20, 50)	0
gru_5 (GRU)	(None, 20, 50)	15,300
dropout_10 (Dropout)	(None, 20, 50)	0
gru_6 (GRU)	(None, 20, 50)	15,300
dropout_11 (Dropout)	(None, 20, 50)	0
gru_7 (GRU)	(None, 128)	69,120
dense_4 (Dense)	(None, 1)	129

(b) GRU arch

Layer (type)	Output Shape	Param #
lstm_3 (LSTM)	(None, 20, 50)	10,400
dropout_6 (Dropout)	(None, 20, 50)	0
lstm_4 (LSTM)	(None, 20, 50)	20,200
dropout_7 (Dropout)	(None, 20, 50)	0
lstm_5 (LSTM)	(None, 50)	20,200
dropout_8 (Dropout)	(None, 50)	0
dense_4 (Dense)	(None, 128)	6,528
dense_5 (Dense)	(None, 1)	129

(c) LSTM

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 20, 50)	2,600
dropout (Dropout)	(None, 20, 50)	0
simple_rnn_1 (SimpleRNN)	(None, 20, 50)	5,050
dropout_1 (Dropout)	(None, 20, 50)	0
simple_rnn_2 (SimpleRNN)	(None, 50)	5,050
dropout_2 (Dropout)	(None, 50)	0
dense (Dense)	(None, 128)	6,528
dense_1 (Dense)	(None, 1)	129

(d) Simple RNN

Figure 7: Architectures for our RNN models

2.2 Task 5:Results

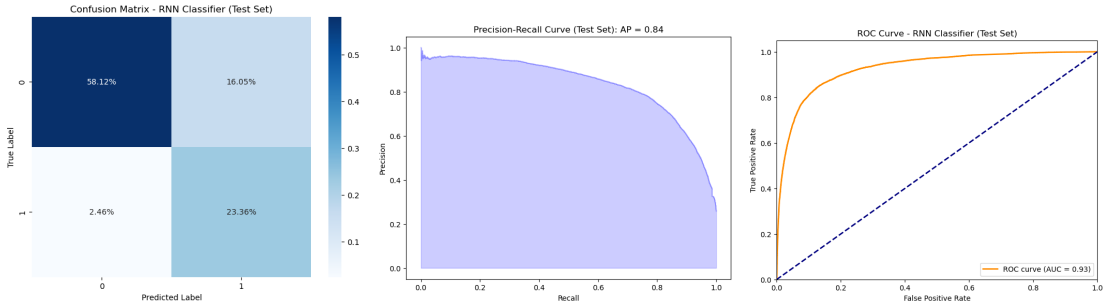
2.2.1 Simple RNN

Table 16: Performance Metrics for Simple RNN

Metric	Value
Test Loss	0.3942
Test Accuracy	0.8149

Table 17: Classification Report for Simple RNN (Test Set)

Class	Precision	Recall	F1-score	Support
0	0.96	0.78	0.86	43,669
1	0.59	0.90	0.72	15,203
Accuracy			0.81	58,872
Macro Avg	0.78	0.84	0.79	58,872
Weighted Avg	0.86	0.81	0.82	58,872



(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 8: Performance Metrics for Simple RNN

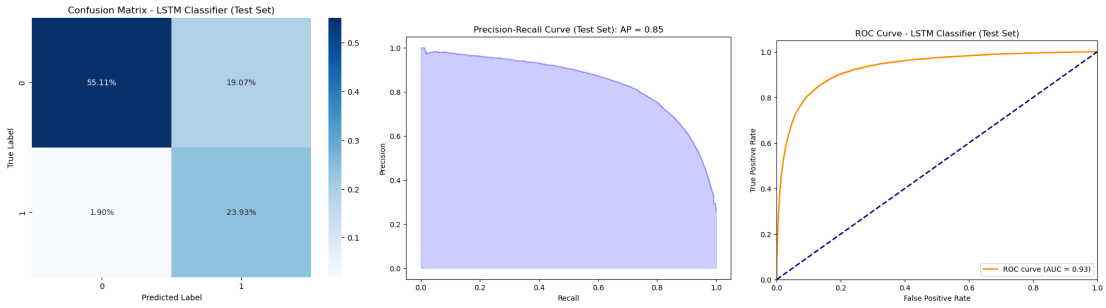
Table 18: Performance Metrics for LSTM

Metric	Value
Test Loss	0.4267
Test Accuracy	0.7903

2.2.2 LSTM RNN

Table 19: Classification Report for LSTM (Test Set)

Class	Precision	Recall	F1-score	Support
0	0.97	0.74	0.84	43,669
1	0.56	0.93	0.70	15,203
Accuracy			0.79	58,872
Macro Avg	0.76	0.83	0.77	58,872
Weighted Avg	0.86	0.79	0.80	58,872



(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 9: Performance Metrics for LSTM RNN

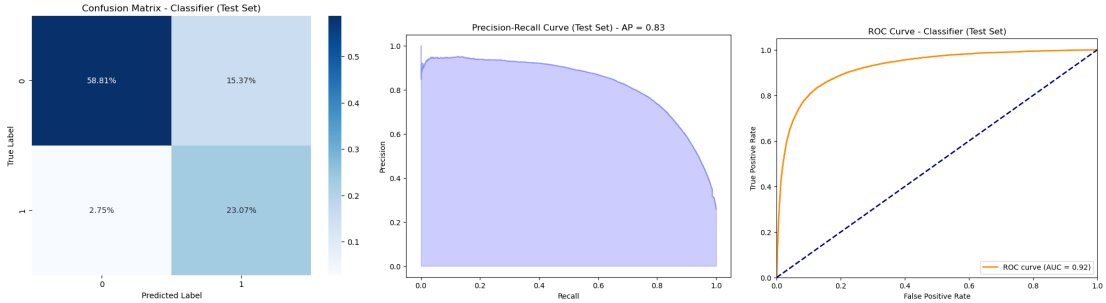
2.2.3 GRU RNN

Table 20: Performance Metrics for GRU

Metric	Value
Test Loss	0.3947
Test Accuracy	0.8188

Table 21: Classification Report for GRU (Test Set)

Class	Precision	Recall	F1-score	Support
0	0.96	0.79	0.87	43,669
1	0.60	0.89	0.72	15,203
Accuracy			0.82	58,872
Macro Avg	0.78	0.84	0.79	58,872
Weighted Avg	0.86	0.82	0.83	58,872



(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 10: Performance Metrics for GRU RNN

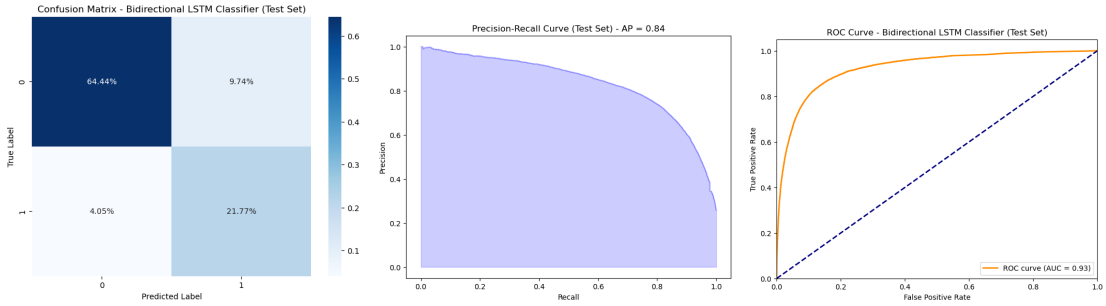
2.2.4 Bidirectional RNN

Table 22: Performance Metrics for Bidirectional RNN

Metric	Value
Test Loss	0.3362
Test Accuracy	0.8621

Table 23: Classification Report for Bidirectional RNN (Test Set)

Class	Precision	Recall	F1-score	Support
0	0.94	0.87	0.90	43,669
1	0.69	0.84	0.76	15,203
Accuracy			0.86	58,872
Macro Avg	0.82	0.86	0.83	58,872
Weighted Avg	0.88	0.86	0.87	58,872



(a) Confusion Matrix (b) Precision-Recall Curve (c) ROC Curve

Figure 11: Performance Metrics for LSTM RNN

2.3 Task 6: Commentary on results

In evaluating the performance of the four RNN models on the minority class (battery status = 1), we observe varying capabilities in handling imbalanced data and capturing temporal dependencies. The Simple RNN achieved an accuracy of 0.8149, demonstrating

high recall (0.90) but lower precision (0.59) for the minority class. This performance suggests that while the Simple RNN effectively captures short-term trends, resulting in high recall, its limitations in retaining information over long sequences lead to some false positives, as indicated by the lower precision.

The LSTM model, despite its complexity, showed the lowest overall accuracy (0.7903) among the four models. It exhibited strong recall (0.93) for the minority class but struggled with precision, indicating potential overfitting. This suggests that the LSTM may have focused too closely on minority-specific patterns, affecting its ability to generalize effectively. The GRU demonstrated slightly better performance than both the Simple RNN and LSTM, achieving an accuracy of 0.8188 and a balanced recall of 0.89 for the minority class. The GRU's simpler architecture appears to provide an effective balance, reducing computational complexity while still managing temporal dependencies adequately.

The Bidirectional RNN stands out as the highest-performing model, with an accuracy of 0.8621 and the strongest handling of class imbalance. It exhibited high and balanced precision and recall (both 0.84) on the minority class. This superior performance can be attributed to its ability to process sequences in both forward and backward directions, allowing it to capture more comprehensive context and temporal dependencies. The Bidirectional RNN's effectiveness in managing the class imbalance is evident in its balanced precision and recall for the minority class.

All models showed some imbalance in predictions, with generally higher precision for class 0 and higher recall for class 1, indicating a likely class imbalance in the dataset. However, the Bidirectional RNN managed this imbalance most effectively. These results highlight the trade-offs between model complexity and performance in handling imbalanced data and temporal dependencies. While the more complex Bidirectional RNN yielded the best results when implemented correctly, simpler models like GRU and Simple RNN also delivered competitive performance, suggesting that the task may not require handling very long-term dependencies. Given that the Bi-directional model required longer to train, we chose the GRU for the comparison task.

2.4 Task 7.a: GRU vs AdaBoost

AdaBoost and GRU exhibit distinct performance levels in predicting battery status, with the GRU significantly outperforming AdaBoost. AdaBoost achieved an accuracy of 0.6589 (65.89%), while the GRU model delivered a much higher accuracy of 0.8149 (81.49%). This difference suggests that the GRU is more adept at learning and classifying both classes effectively.

Both models faced challenges in dealing with the imbalanced dataset, but the GRU showed a stronger ability to handle this imbalance. The precision and recall values reveal that AdaBoost struggles with the minority class (Class 1), as evidenced by its precision of 0.39 and recall of 0.61 for Class 1. In contrast, the GRU exhibited better results with a precision of 0.59 and recall of 0.90 for the minority class, showcasing its capability to better identify positive cases while maintaining high overall accuracy.

In terms of F1 scores, which combine precision and recall, the GRU outperformed AdaBoost in both classes. For Class 0 (the majority class), the GRU achieved an F1-score of 0.86, compared to AdaBoost's 0.75. For Class 1, the GRU's F1 score of 0.72 is much higher than AdaBoost's 0.48. These improvements in the F1-score for both classes reflect the GRU's superior balance between precision and recall, particularly in identifying and classifying the minority class.

The GRU's advantage is primarily attributed to its architecture, which is designed to capture temporal dependencies in sequential data. In battery status prediction, where past information is crucial in forecasting future states, the GRU's ability to remember or forget information over time is crucial. This allows it to model the complex patterns inherent in the sequential nature of the data. Conversely, AdaBoost, being an ensemble learning method, focuses on combining weak classifiers but does not explicitly model temporal dependencies, which may explain its suboptimal performance in this task. A key distinction between the two models is the type of data preprocessing used. The GRU model utilized BorderSMOTE, a variation of SMOTE that generates synthetic samples near the decision boundary, whereas AdaBoost was trained using the standard SMOTE. BorderSMOTE's ability to generate more challenging synthetic samples helps the model focus on harder-to-classify instances, likely contributing to the GRU's better performance, especially the recall and precision for the minority class.

2.5 Task 7.b: Extending LSMT

To make the most of LSTM models for predicting battery status, we can look at different ways to improve how well they work.

One good idea is to use attention methods together. Attention helps the model pay more attention to the important parts of the input. This makes it better at understanding long-term connections in the data and noticing key moments or important pieces of information. Attention-based LSTMs can do better than regular LSTMs by giving different importance to different features. This is especially useful for tasks like ours, where understanding the full context is important for making accurate predictions.

Another useful addition is using Bidirectional LSTMs (BiLSTMs), which read the input sequence both forward and backward. This method helps the model understand the data better because it can use information both backwards and forwards to make correct predictions. BiLSTMs can be very helpful for us because the meaning of the battery status at any point in time can depend on the information around it.

We can also look at mixed models, like ConvLSTM, which combine the best features of convolutional layers and LSTM units. Convolutional layers are great at finding small patterns and details, while LSTMs are good at remembering information over a long time in ordered data. By combining these two helpful parts, the ConvLSTM model can better grasp the complicated patterns in our battery status data. This may help it perform better in tasks like predicting battery status.

Also, we need to think about ways to fix the uneven distribution of classes in our data. Techniques like BorderSMOTE(which we attempted for this assignment), which creates synthetic data, or changing class weights during training can help the model learn the different classes better. This helps the model predict the minority class more accurately, such as telling if there's a problem with the battery. Making sure the model has enough examples to learn from for both groups can improve how well it predicts.

Besides these main methods, we can also look at combining LSTM models with other types of ensemble models, like decision trees or random forests. By using different modelling methods together, we can make better and more accurate predictions about the status of our batteries.