

BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering  
ENCS5141 - INTELLIGENT SYSTEMS LAB

## Case Study #2: Ensemble Methods: A Comprehensive Comparative study

**Prepared by:** Ahmad Abbas

**Instructor:** Aziz Qaroush

**Assistant:** Eng. Mazen Amria

**Date:** December 25, 2023

# Abstract

This report aims to dive into ensemble learning, specially Bagging and Boosting methods. It focuses on Random Forest and XGBoost models with evaluating them across various scenarios, including the Fashion MNIST, Student Dropout, and Street View House Numbering datasets, in order to investigate their performance and hyperparameter tuning. The report examines each algorithm's capability in handling large, imbalanced, and noisy datasets, providing insights into their strengths, limitations, and optimization strategies.

# Contents

<b>1</b>	<b>Literature Review</b>	<b>1</b>
1.1	Ensemble Methods . . . . .	1
1.1.1	Sequential Ensemble Methods (Boosting) . . . . .	2
1.2	Parallel Ensemble Methods . . . . .	4
1.3	Bagging vs Boosting . . . . .	6
1.3.1	Usage of Decision Trees . . . . .	6
1.3.2	Performance and Speed: . . . . .	6
1.3.3	Regularization . . . . .	6
1.3.4	Learning Rate Optimization . . . . .	6
1.4	Evaluation Metrics . . . . .	7
1.4.1	Accuracy . . . . .	7
1.4.2	F1-Score . . . . .	7
1.4.3	ROC AUC . . . . .	7
1.5	Scenarios and Datasets . . . . .	8
1.5.1	Fashion MNIST . . . . .	8
1.5.2	Predict students' dropout and academic success . . . . .	8
1.5.3	The Street View House Numbers (SVHN) Dataset . . . . .	8
<b>2</b>	<b>Scenarios Analysis</b>	<b>9</b>
2.1	First Scenario: Fashion MNIST Dataset . . . . .	9
2.2	Second Scenario: Student Dropout Dataset . . . . .	10
2.3	Third Scenario: SVHN Dataset . . . . .	11
2.4	Overall Comparative Analysis . . . . .	12
2.4.1	Discussion . . . . .	12
<b>3</b>	<b>Conclusion</b>	<b>13</b>

# 1 Literature Review

In this report, we will talk about about ensemble learning in general, and introduce one method from the two types Bagging and Boosting. Moreover, we will talk about their specification including hyper-parameters, advantages, limitations, and discuss the main differences including the performance.

## 1.1 Ensemble Methods

Ensemble methods are machine learning techniques that aim ti increase the performance by combining several base models instead of using one model. This combination of models increase the accuracy of the results, which is the main reason of the popularity of ensemble methods [1].

Ensemble methods can be divided into three main categories: sequential, parallel, and stacking ensemble techniques. For this report, we will use decision trees [2] to describe ensemble methods' concept and application. However, Ensemble Methods do not only pertain to Decision Trees.

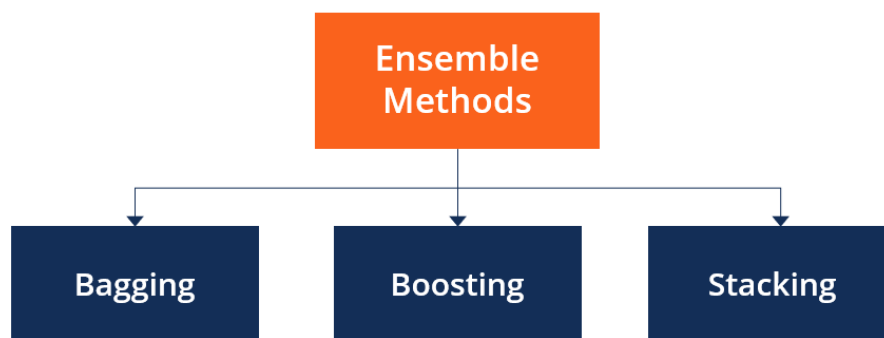


Figure 1.1: Ensemble Methods Categories [3]

For each member of the ensemble, a bootstrap sample of the training dataset is created, and a decision tree model is trained on each sample [4]. There are different ways of combining models results such as:

1. **Voting:** Multiple models vote on the prediction. In classification, the majority vote wins (hard voting), while in regression, the average prediction is used (soft voting).
2. **Stacking:** A model learns how to best combine the predictions from several models.
3. **Bagging:** Each model is trained on a random subset of the data, and their predictions are averaged (regression) or voted on (classification).
4. **Boosting:** Models are added sequentially to correct the errors of prior models, with final predictions made from the weighted sum of all models.

### 1.1.1 Sequential Ensemble Methods (Boosting)

For sequential techniques, they generate base models as a sequence, e.g., Adaptive Boosting, this generation improves the reliance between the base learners. Thus, the performance improved by assigning higher weights learners who had previously been misrepresented [3]. This approach first discussed in a 1990 paper by Schapire [5].

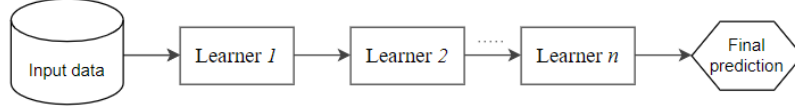


Figure 1.2: Block diagram of sequential ensemble learning [6].

The core idea of boosting revolves around sequentially applying a base learning algorithm to modified versions of the input data. Specifically, boosting methods train a weak learner using the input data, then generate predictions. Subsequently, they identify incorrectly classified training samples. These misclassified instances are then used to form an adjusted training set, which is employed to train the next weak learner. This process is repeated, with each successive learner focusing on the samples that were previously misclassified, thereby continually refining the model's accuracy [6].

---

**Algorithm 1** Generic Boosting Algorithm [5]

---

- 1: Initialize the training dataset  $D$
  - 2: Initialize weights  $w_i$  for each instance  $x_i$  in  $D$
  - 3: **for**  $t = 1$  to  $T$  **do**
  - 4:     Train weak learner  $L_t$  using  $D$  with weights  $w_i$
  - 5:     Calculate error  $\epsilon_t$  of learner  $L_t$  on  $D$
  - 6:     Calculate learner weight  $\alpha_t = f(\epsilon_t)$
  - 7:     **for** each instance  $(x_i, y_i)$  in  $D$  **do**
  - 8:         Update weight  $w_i \leftarrow g(w_i, \alpha_t, L_t(x_i), y_i)$
  - 9:     **end for**
  - 10:    Normalize weights  $w_i$
  - 11: **end for**
  - 12: **return** Final model  $M$ , a weighted combination of  $L_1, L_2, \dots, L_T$
- 

There are many algorithms used in boosting such as: AdaBoost, Gradient Boosting, XGBoost, and LightGBM [7]. For this report, we will talk about XGBoost and its specifications.

XGBoost algorithm, which is a decision tree based, a very reliable and scalable algorithm for classification that uses gradient boosting structure [8]. It was firstly introduced in 2016 in "XGBoost: A scalable tree boosting system" [9]. Surely, it came with improvements to gradient boosting. One of them is using regularization term that prevents overfitting in its loss function.

XGBoost have similar hyperparameters to gradient boost such as [10]:

1. **learning\_rate**: Updates use step size shrinkage to avoid overfitting. The weights of newly added features can be obtained immediately after each boosting step.
2. **max\_depth**: Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. XGBoost aggressively consumes memory when training a deep tree.
3. **subsample**: Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration.

The XGBoost algorithm has many advantages, including a low need for feature engineering (i.e., data normalization and feature scaling), as the algorithm can handle these scenarios well. XGBoost also has the ability to handle missing values, which is a beneficial feature for datasets from the real world.

XGBoost’s number of hyperparameters, despite its flexibility, is one of its drawbacks and can make tuning process difficult [6]. Moreover, because it is limited to convex loss functions, it might not be appropriate for all applications [11]. In addition, being a greedy algorithm that takes a long time and doesn’t require multi-threaded optimization [12].

---

**Algorithm 2** XGBoost Algorithm [13]

---

**Require:**

- $I$ : a training set  $\{(x_i, y_i)\}_{i=1}^n$  with feature dimension  $m$
  - $L(y, \hat{y})$ : a differentiable loss function
  - $K$ : number of boosting iterations
  - $\eta$ : the learning rate
  - $\lambda$ : regularization coefficient
  - $\gamma$ : minimum loss reduction required for a split
- 1: Initialize the model with a constant value:
  - 2:  $F_0(x) = \arg \min_{\gamma} \sum_{i=1}^n L(y_i, \gamma)$
  - 3: **for**  $k = 1$  to  $K$  **do**
  - 4:   /\* Compute the sum of gradients and Hessians of all the samples \*/
  - 5:    $G \leftarrow \sum_{i \in I} \partial L(y_i, F_{k-1}(x_i)) / \partial F_{k-1}(x_i)$
  - 6:    $H \leftarrow \sum_{i \in I} \partial^2 L(y_i, F_{k-1}(x_i)) / \partial^2 F_{k-1}(x_i)$
  - 7:    $T \leftarrow \text{BUILD-TREE}(I, G, H)$
  - 8:   Let the terminal regions of  $T$  be  $R_j$  for  $j = 1, \dots, J_k$
  - 9:   **for**  $j = 1, \dots, J_k$  **do**
  - 10:      $w_j = -\frac{G_j}{H_j + \lambda}$
  - 11:   **end for**
  - 12:   where  $G_j$  and  $H_j$  are the sum of gradients and Hessians at leaf node  $j$ .
  - 13:   Update the model:
  - 14:    $F_k(x) = F_{k-1}(x) + \eta \sum_{j=1}^{J_k} w_j \mathbb{I}\{x \in R_j\}$
  - 15: **end for**
  - 16: **return** the final ensemble  $F_K(x)$
-

## 1.2 Parallel Ensemble Methods

Parallel methods work by training a number of basic classifiers separately and then putting their classifications together through a combining process. A well-known parallel method is bagging, which has been further developed into the Random Forest algorithm [14]. These parallel ensemble algorithms create base learners at the same time in order to ensure there's differences among the different models in the group [15].

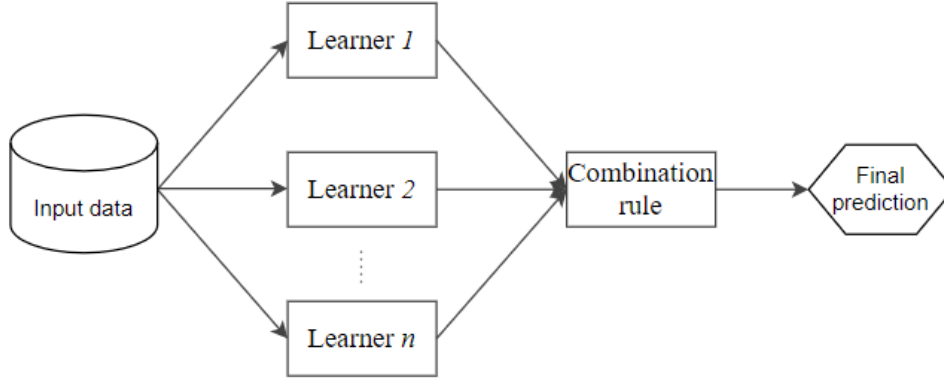


Figure 1.3: Block diagram of parallel ensemble learning [6].

Bagging mainly aims to make the predictions of the base models, especially models that are sensitive to any change in the training data, and make them more consistent. When the models in the group vary a lot in their predictions, it works with better performance. Decision trees when bagged give better results. On the other hand, methods like k-nearest neighbors and naïve Bayes are quite stable, they don't change much with slight data variation so bagging doesn't improve their performance by much [16].

---

### Algorithm 3 Bagging Algorithm [17]

---

**Require:** Data set  $TR = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$

**Require:** Base learning algorithm  $L$

**Require:** Number of learning rounds  $K$

```

1: for  $k = 1, \dots, K$  do
2:    $TR_k \leftarrow \text{BOOTSTRAP}(TR)$ 
3:    $h_k \leftarrow L(TR_k)$ 
4: end for
  
```

**Ensure:** Final ensemble classifier  $H(X)$

```

5: function  $H(X)$ 
6:   return  $\arg \max_{y \in Y} \sum_{k=1}^K I(h_k(X) = y)$ 
7: end function
  
```

---

There are many algorithms used in bagging, we will only discuss Random Forest algorithm [18], which decision tree is the main block in this algorithm. This algorithm is widely used due to it's simple implementation with fast and high performance.

The random forest get high accuracy by having each tree vote on the final outcome, which helps stop any single tree from overfitting the data too closely. The algorithm operates in parallel, with each tree in the ensemble acting as a base learner, following the ensemble learning framework shown in Figure 1.3.

What makes random forests special is that they mix things up by using different sets of information for each tree. This method is like making sure every tree gets a unique set of questions to answer. It's different from other methods that let every tree use all the information available. By doing things this way, the trees end up being quite different from one another, which is good because it means they can make better group decisions.

In general, it can be noticed that random forests usually do a better job than if you just used the basic method of letting each tree see all the data. They're also pretty good at dealing with missing pieces of information and can handle lots of different data at once. Despite its strengths, including its ability to handle missing data and perform well with high-dimensional datasets [19], the random forest algorithm does face challenges. As the number of trees increases, while accuracy may improve, the model's training and prediction speed can decrease, highlighting a trade-off between performance and efficiency.

---

**Algorithm 4** Random Forest Algorithm

---

**Require:** Training data set  $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$

**Require:** Number of attributes  $p$

**Require:** Number of trees  $T$  in the forest

**Require:** Number of class labels  $C$

```

1: for  $t = 1, \dots, T$  do
2:   Generate a bootstrap sample  $S_t$  from the input data  $S$ 
3:   Fit a base learner  $h_t$  using  $S_t$ 
4:   for each node  $n$  in learner  $h_t$  do
5:     Randomly select  $k$  attributes, where  $k \approx \sqrt{p}$ 
6:     Compute the best split features using the selected attributes
7:     Split the node using the optimal split features
8:   end for
9:   Repeat the process until stopping criteria are met
10: end for
11: Combine the outputs of all trees to form the forest prediction
12: For a given test sample  $x$ , the final predicted class label is:
13:  $H_T(x) = \arg \max_{j \in \{1, \dots, C\}} \left( \sum_{t=1}^T I(h_t(x) = j) \right)$ 
```

---



## 1.3 Bagging vs Boosting

In this section, we will compare the two algorithms we discussed, XGBoost and Random Forest. We will introduce the differences across many aspects.

### 1.3.1 Usage of Decision Trees

- **Random Forest:** It constructs a forest of many decision trees while training, each of them trained on different subset of data. With this approach it reduces the variance and increases the bias.
- **XGBoost:** It constructs the trees in sequence, such that each tree is enhanced and corrected by the next tree on the sequence. It reduces both bias and variance [20].

### 1.3.2 Performance and Speed:

- **Random Forest:** In general, Random forest is consumes more memory due to its process of building trees independently in parallel so we have multiple trees. These parallel trees are the main reason of Random forest speed, mainly with training large datasets [21].
- **XGBoost:** It requires more memory-eefcient mainly due to its sequential tree-building process, which no need to save more than the current tree and the previous one with intermediate data in each step. This approach takes longer to complete but gives more accurate and frequently better-performing model, particularly in complex datasets [9] .

### 1.3.3 Regularization

- **Random Forest:** Despite being better in reducing the risk of overfitting than single decision tree, due to its ability on building multiple trees with randomized data, it does not have a specific way to reduce overfitting. It mainly depends on randomness and the aggregation of results.
- **XGBoost:** it uses two key strategies, L1 and L2 regularization, to prevent overfitting. L1 helps in choosing the most important features by reducing the less important ones to zero, while L2 keeps any single feature from becoming too influential by shrinking all feature coefficients. This approach ensures XGBoost models are both effective and not overly complex, particularly useful in handling large and complex datasets.[22].

### 1.3.4 Learning Rate Optimization

XGBoost learning rate is a critical part of its hyperparameters for enhancing the model. The common approach is to experiment different values for learning rate, generally between 0.01 and 0.2 [23]. This process may be done with multiple tools as GridSearchCV [24] or RandomizedSearchCV [25] for tuning the parameters by running multiple learning rate values and compare their performance. When tuning XGBoost's parameters, it's also important to consider the relation between the learning rate and the number of trees (n\_estimators). A smaller learning rate usually requires more trees to model all the

subtleties in the data effectively. Thus, the learning rate and the number of trees should be tuned together to find the best combination for your specific dataset and task [26].

In the other hand, Random Forest is not like XGBoost in the concept of learning rate optimization. Because it builds trees independently and does not use the concept of a learning rate in the same way. It focuses more in parameters like the number of trees, max depth of the trees, and features to consider at each split with the same way in XGBoost [27].

## **1.4 Evaluation Metrics**

### **1.4.1 Accuracy**

The accuracy metric provides an overall assessment of the model's ability to correctly predict the output on the complete dataset. Each individual in the dataset contributes equally to the accuracy score, as every unit holds the same weight [28].

### **1.4.2 F1-Score**

The F1-score can be seen as a weighted average of Precision and Recall, with the best F1-score being 1 and the worst being 0. Precision refers to the number of correct positive results divided by the number of all positive results, while Recall refers to the number of correct positive results divided by the number of positive results that should have been returned. Precision and Recall hold equal importance in determining the F1-score and the harmonic mean helps find the optimal balance between these two metrics [28].

### **1.4.3 ROC AUC**

The Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC) are main metrics in evaluating the performance of classification models. The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at different threshold settings, providing a comprehensive view of the model's ability to distinguish between the two classes. The TPR (or sensitivity) measures how well the model identifies positives, while the FPR indicates the proportion of negatives that are incorrectly identified as positives. The AUC, a single number summary of the ROC curve, quantifies the overall ability of the model to avoid false classification. An AUC of 1 represents perfect classification, while an AUC of 0.5 suggests no discriminative ability, more like random guessing. The ROC AUC is particularly useful because it is independent of the threshold chosen and provides an aggregated measure of performance across all possible classification thresholds [29].

## 1.5 Scenarios and Datasets

### 1.5.1 Fashion MNIST

The Fashion-MNIST (F-MNIST) dataset consists of 60,000 samples in the training set and 10,000 samples in the testing set, featuring 10 categories of 28x28 grayscale images of fashion products. As shown in Figure 1.4, the class labels, names, and some sample images of F-MNIST are displayed [30]. This dataset will represent the scenario of having a large dataset.

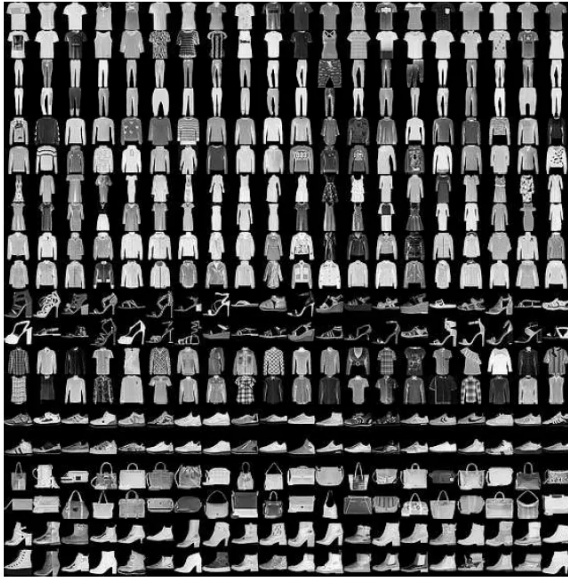
Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Figure 1.4: Fashion-MNIST Dataset Images with Labels and Description

### 1.5.2 Predict students' dropout and academic success

A dataset created from a college that combines information from different sources. It includes details about students in various majors gathered when they first enrolled. This information covers their educational background, personal demographics, and socio-economic status, along with their grades for the first two semesters. The dataset is used to make models that can predict whether a student might drop out or do well academically. The challenge is that this prediction task is divided into three groups, and there's a notable imbalance, with one group having a lot more data points than the others [31]. This dataset represent the imbalanced datasets.

### 1.5.3 The Street View House Numbers (SVHN) Dataset

SVHN is a real-world image dataset that requires little preparation or formatting for the development of machine learning and object recognition algorithms. It shares some flavor similarities with MNIST, but it comes from a much harder, unsolved real world problem (recognizing digits and numbers in natural scene images) and incorporates an order of magnitude more labeled data (over 600,000 digit images). The home numbers in Google Street View photos are used to calculate SVHN [32]. This dataset will be the scenario of noisy and large data.

## 2 Scenarios Analysis

In this section, we will talk about three different scenarios to conduct a comprehensive comparative study between Random Forest and XGBoost. We will use the three dataset that discussed in literature review. In our models, we did a random search for best hyperparameters. The Table 1 shows the hyperparameters for the models:

Table 1: Hyper-parameters for Random Forest and XGBoost

Model	Hyperparameter	Type	Tested
Random Forest	n_estimators	integer	{10, 50, 100}
	criterion	nominal	{gini, entropy}
	max_depth	integer	{10, 50, 100}
	max_features	nominal	{sqrt, log2}
XGBoost	n_estimators	integer	{10, 50, 100}
	learning_rate	float	{0.01, 0.1, 0.2}
	max_depth	integer	{5, 50, 100}

### 2.1 First Scenario: Fashion MNIST Dataset

The first scenario focuses on the Fashion MNIST dataset, particularly focusing on the performance of models on large datasets. After hyperparameter tuning, the selected parameters for Random Foerst were: n\_estimators = 100, criterion = 'entropy', max\_depth = 100, and max\_features = 'sqrt'. For XGBoost, they were: n\_estimators = 100, learning\_rate = 0.1, max\_depth = 50.

As shown in Figure 2.1, both models have similar ROC Curves. Moreover, as shown in Table 2, we can see that XGBoost outperformed Random Forest but not that much. This shows their ability to handle large datasets in terms of performance.

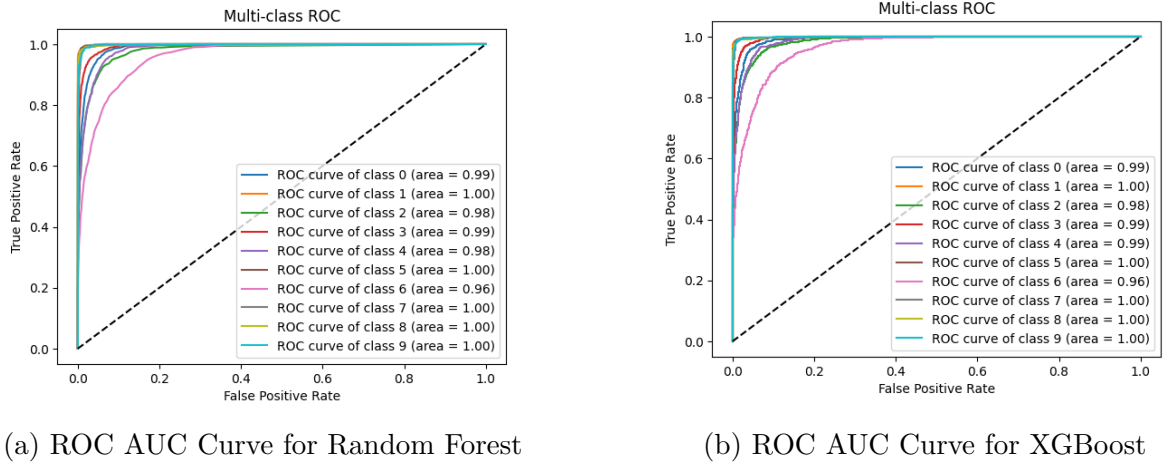


Figure 2.1: ROC AUC Curves for Models on the Fashion MNIST Dataset

Table 2: Performance Metrics Comparison

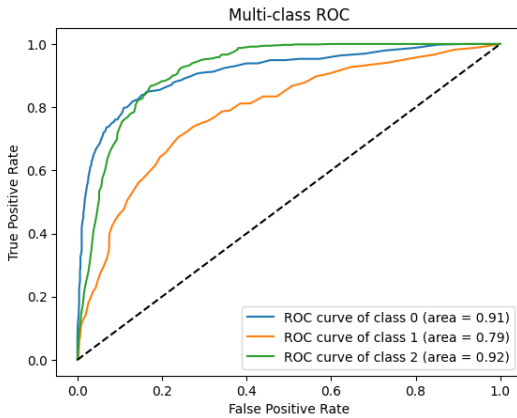
Metric	Precision	Recall	F1-Score	Accuracy	Macro Avg	Micro Avg
Random Forest	0.87463	0.87570	0.87419	0.87570	0.87463	0.87570
XGBoost	0.88697	0.88780	0.88664	0.88780	0.88697	0.88780

In terms of computational efficiency, the differences were clear between both models in Fashion MNIST. Random Forest, optimized for speed, completed training in 83 seconds. This was significantly shorter time than XGBoost, which took 442 seconds. But with higher memory consumption of 166.7 MiB. However, XGBoost consumed 36.4 Mib. This trade-off between time and memory was clearly shown in this large dataset with small difference in the accuracy.

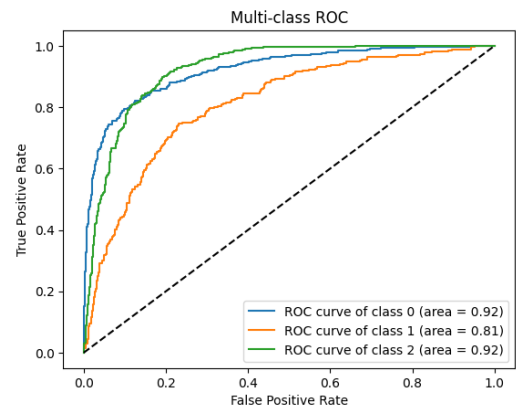
## 2.2 Second Scenario: Student Dropout Dataset

The Second scenario focuses on Student Dropout Dataset, particularly focusing on the performance of models on imbalanced datasets. We have imported this library using ucimlrepo library[33]. After hyperparameter tuning, the selected parameters for Random Foerst were: `n_estimators = 100`, `criterion = 'entropy'`, `max_depth = 100`, and `max_features = 'log2'`. For XGBoost, they were: `n_estimators = 100`, `learning_rate = 0.1`, `max_depth = 5`.

As shown in Figure 2.2, both models have similar ROC Curves with better ROC AUC for XGBoost. Moreover, as shown in Table 3, the results shows that XGBoost slightly outperforms Random Forest in terms of precision, recall, F1-score, and accuracy. This suggests that XGBoost's handling of imbalanced data, likely due to its built-in regularization and focus on correcting previous errors, is more effective than Random Forest's approach.



(a) ROC AUC Curve for Random Forest



(b) ROC AUC Curve for XGBoost

Figure 2.2: ROC AUC Curves for Models on Student Dropout Dataset

Table 3: Performance Metrics Comparison

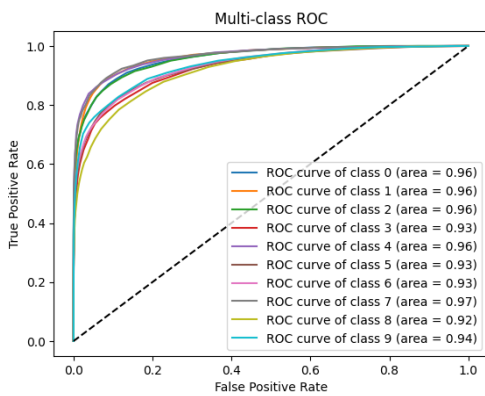
Metric	Precision	Recall	F1-Score	Accuracy	Macro Avg	Micro Avg
Random Forest	0.70133	0.65124	0.65421	0.75593	0.70133	0.75593
XGBoost	0.71448	0.67683	0.68425	0.76746	0.71448	0.76746

In terms of computational efficiency, the differences were small between both models because the dataset was small and not large enough. Random Forest had smaller execution time (1.28 seconds) than XGBoost (4.91 seconds). But with more memory consumption, 7.5 MiB for Random Forest and 0.8 MiB for XGBoost.

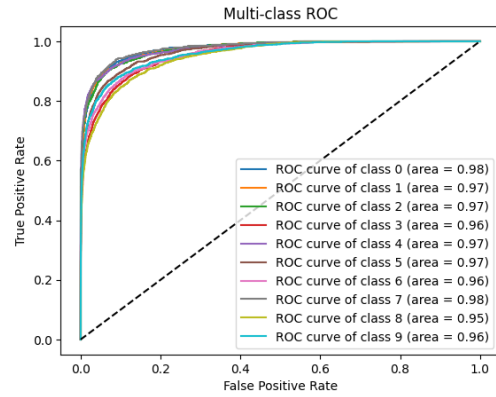
### 2.3 Third Scenario: SVHN Dataset

The last scenario contains two issues, large and noisy datasets. After hyperparameter tuning, the selected parameters for Random Foerst were: `n_estimators = 100`, `criterion = 'entropy'`, `max_depth = 50`, and `max_features = 'logo2'`. For XGBoost, they were: `n_estimators = 100`, `learning_rate = 0.1`, `max_depth = 50`.

As shown in Figure 2.3, we can see difference in ROC Curves with better ROC AUC for XGBoost for most of all classes. Moreover, as shown in Table 4, XGBoost outperforms Random Forest in all metrics showing how much it is robust ti noisy imbalanced data more that Random Forest.



(a) ROC AUC Curve for Random Forest



(b) ROC AUC Curve for XGBoost

Figure 2.3: ROC AUC Curves for Models on SVHN

Table 4: Performance Metrics Comparison

Metric	Precision	Recall	F1-Score	Accuracy	Macro Avg	Micro Avg
Random Forest	0.74490	0.74328	0.74269	0.74367	0.74524	0.74367
XGBoost	0.77001	0.76962	0.76928	0.76994	0.77027	0.76994

Regarding computational efficiency, the difference in performance between the two models was notably shown, due to the large noisy dataset. The Random Forest model achieved a quicker execution time, completing its task in 416 seconds, compared to the 1066 seconds required by XGBoost. However, this speed came at the cost of higher memory usage, with Random Forest consuming 336.6 MiB, more than the 308 MiB used by XGBoost.

## 2.4 Overall Comparative Analysis

This subsection presents an overall comparison of the Random Forest and XGBoost models across the three scenarios. The comparison highlights key performance metrics and computational efficiencies, offering a top view of each model’s strengths and weaknesses in different scenarios.

Table 5: Overall Performance Metrics Comparison

Scenario	Model	Precision	Recall	F1-Score	Accuracy	Macro Avg	Micro Avg
Fashion MNIST	Random Forest	0.87463	0.87570	0.87419	0.87570	0.87463	0.87570
	XGBoost	0.88697	0.88780	0.88664	0.88780	0.88697	0.88780
Student Dropout	Random Forest	0.70133	0.65124	0.65421	0.75593	0.70133	0.75593
	XGBoost	0.71448	0.67683	0.68425	0.76746	0.71448	0.76746
SVHN	Random Forest	0.74490	0.74328	0.74269	0.74367	0.74524	0.74367
	XGBoost	0.77001	0.76962	0.76928	0.76994	0.77027	0.76994

Table 6: Overall Computational Efficiency Comparison

Scenario	Model	Execution Time (s)	Memory Usage (MiB)
Fashion MNIST	Random Forest	83	166.7
	XGBoost	442	36.4
Student Dropout	Random Forest	1.28	7.5
	XGBoost	4.91	0.8
SVHN	Random Forest	416	336.6
	XGBoost	1066	308

### 2.4.1 Discussion

Overall, the comparison shows that under different scenarios, Random Forest and XGBoost performed with different efficiency and performance characteristics. Though it required more time to execute and used less memory than Random Forest, XGBoost outperformed Random Forest in the Fashion MNIST and SVHN situations. The performance difference was more subtle in the Student Dropout scenario, indicating the ability of both models to handle imbalanced datasets. These findings highlight how crucial it is to take into account model correctness as well as computing capacity in real-world applications.

### 3 Conclusion

This report was a comparative study across three different scenarios, including Fashion MNIST, Student Dropout, and SVHN datasets, showing different aspects of Random Forest and XGBoost. While both algorithms showed competence in handling various datasets, XGBoost generally outperformed Random Forest, especially in terms of handling imbalanced data due to its regularization techniques and error correction mechanisms, which effectively reduces overfitting and enhances model accuracy. However, trade-offs between time and memory efficiency were observed, highlighting the importance of selecting the appropriate model based on specific dataset characteristics and computational resource availability making Random Forest a preferable choice in scenarios with such scenarios. The report concludes that the choice of algorithm should be guided by specific dataset characteristics and resource availability.



## References

- [1] E. Lutins, “Ensemble methods in machine learning: What are they and why use them?” *Medium*, Jun. 2018. [Online]. Available: <https://towardsdatascience.com/ensemble-methods-in-machine-learning-what-are-they-and-why-use-them-68ec3f9fef5f>
- [2] J. R. Quinlan, “Induction of decision trees,” *Machine learning*, vol. 1, pp. 81–106, 1986.
- [3] C. Team, “Ensemble methods,” Nov. 2023. [Online]. Available: <https://corporatefinanceinstitute.com/resources/data-science/ensemble-methods/>
- [4] Z. Cha and Y. Ma, *Ensemble Machine Learning: Methods and applications*. Springer International Publishing, Feb. 2012. [Online]. Available: <https://ci.nii.ac.jp/ncid/BB11157570>
- [5] R. E. Schapire, “The strength of weak learnability,” *Machine learning*, vol. 5, pp. 197–227, 1990.
- [6] I. D. Mienye and Y. Sun, “A survey of ensemble learning: Concepts, algorithms, applications, and prospects,” *IEEE Access*, vol. 10, pp. 99 129–99 149, 2022.
- [7] G. Kunapuli, *Ensemble Methods for Machine Learning*. Manning Publications Co., 2023. [Online]. Available: <http://gen.lib.rus.ec/book/index.php?md5=01989EC2FEE17833A3124D5D112140A2>
- [8] J. H. Friedman, “Greedy function approximation: a gradient boosting machine,” *Annals of statistics*, pp. 1189–1232, 2001.
- [9] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.
- [10] [Online]. Available: <https://xgboost.readthedocs.io/en/stable/parameter.html>
- [11] Y. Guang, “Generalized xgboost method,” *arXiv preprint arXiv:2109.07473*, 2021.
- [12] M. Ma, G. Zhao, B. He, Q. Li, H. Dong, S. Wang, and Z. Wang, “Xgboost-based method for flash flood risk assessment,” *Journal of Hydrology*, vol. 598, p. 126382, 2021.
- [13] R. Yehoshua, “Xgboost: The definitive guide (part 1) - towards data science,” *Medium*, Oct. 2023. [Online]. Available: <https://towardsdatascience.com/xgboost-the-definitive-guide-part-1-cc24d2dcd87a>
- [14] Y. Pandya, “Ensemble methods/ techniques in machine learning, bagging, boosting, random forest, gbdt, xg boost, stacking, light gbm, catboost — analytics vidhya,” *Medium*, Dec. 2021. [Online]. Available: <https://medium.com/analytics-vidhya/>

- [15] H. Liu, A. Gegov, and M. Cocea, *Ensemble Learning Approaches*. Cham: Springer International Publishing, 2016, pp. 63–73. [Online]. Available: [https://doi.org/10.1007/978-3-319-23696-4\\_6](https://doi.org/10.1007/978-3-319-23696-4_6)
- [16] N. C. Oza and K. Tumer, “Classifier ensembles: Select real-world applications,” *Information fusion*, vol. 9, no. 1, pp. 4–20, 2008.
- [17] G. Zararsiz, H. Y. Akyildiz, D. GÖKSÜLÜK, S. Korkmaz, and A. ÖZTÜRK, “Statistical learning approaches in diagnosing patients with nontraumatic acute abdomen,” *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 24, no. 5, pp. 3685–3697, 2016.
- [18] T. K. Ho, “Random decision forests,” in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1. IEEE, 1995, pp. 278–282.
- [19] S. Hong and H. S. Lynn, “Accuracy of random-forest-based imputation of missing data in the presence of non-normality, non-linearity, and interaction,” *BMC medical research methodology*, vol. 20, no. 1, pp. 1–12, 2020.
- [20] A. Kumar, “Random forest vs xgboost: Which one to use? examples,” Dec. 2023. [Online]. Available: <https://vitalflux.com/random-forest-vs-xgboost-which-one-to-use/>
- [21] A. Liaw, M. Wiener *et al.*, “Classification and regression by randomforest,” *R news*, vol. 2, no. 3, pp. 18–22, 2002.
- [22] A. Um, “L1, l2 regularization in xgboost regression - albert um - medium,” *Medium*, Mar. 2022. [Online]. Available: <https://albertum.medium.com/l1-l2-regularization-in-xgboost-regression-7b2db08a59e0>
- [23] J. Brownlee, “Tune learning rate for gradient boosting with xgboost in python,” Aug. 2020. [Online]. Available: <https://machinelearningmastery.com/tune-learning-rate-for-gradient-boosting-with-xgboost-in-python/>
- [24] [Online]. Available: [https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.GridSearchCV.html#sklearn.model\\_selection.GridSearchCV](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV)
- [25] [Online]. Available: [https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.RandomizedSearchCV.html#sklearn.model\\_selection.RandomizedSearchCV](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html#sklearn.model_selection.RandomizedSearchCV)
- [26] A. Jain, “Mastering xgboost parameter tuning: A complete guide with python codes,” Oct. 2023. [Online]. Available: <https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/>
- [27] W. contributors, “Random forest,” Dec. 2023. [Online]. Available: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

[//en.wikipedia.org/wiki/Random\\_forest](https://en.wikipedia.org/wiki/Random_forest)

- [28] M. Grandini, E. Bagli, and G. Visani, “Metrics for multi-class classification: an overview,” *arXiv preprint arXiv:2008.05756*, 2020.
- [29] [Online]. Available: <https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc>
- [30] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms,” *arXiv preprint arXiv:1708.07747*, 2017.
- [31] V. Realinho, M. Vieira Martins, J. Machado, and L. Baptista, “Predict students’ dropout and academic success,” UCI Machine Learning Repository, 2021, DOI: <https://doi.org/10.24432/C5MC89>.
- [32] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, “Reading digits in natural images with unsupervised feature learning,” 2011.
- [33] D. Newman, S. Hettich, C. Blake, and C. Merz, “Uci repository of machine learning databases,” 1998. [Online]. Available: <http://www.ics.uci.edu/~mlearn/MLRepository.html>

## List of Figures

1.1	Ensemble Methods Categories [3]	1
1.2	Block diagram of sequential ensemble learning [6].	2
1.3	Block diagram of parallel ensemble learning [6].	4
1.4	Fashion-MNIST Dataset Images with Labels and Description	8
2.1	ROC AUC Curves for Models on the Fashion MNIST Dataset	9
2.2	ROC AUC Curves for Models on Student Dropout Dataset	10
2.3	ROC AUC Curves for Models on SVHN	11

## List of Tables

1	Hyper-parameters for Random Forest and XGBoost . . . . .	9
2	Performance Metrics Comparison . . . . .	10
3	Performance Metrics Comparison . . . . .	11
4	Performance Metrics Comparison . . . . .	11
5	Overall Performance Metrics Comparison . . . . .	12
6	Overall Computational Efficiency Comparison . . . . .	12