Ensemble Methods Project Report

Mohammed Ali EL ADLOUNI

October 21, 2024

1 Introduction

In this report, we explore the use of various ensemble methods on a set of four datasets for classification tasks. The objective is to evaluate and compare the performance of different ensemble approaches such as bagging, boosting, and stacking and variations of them. The datasets pose specific challenges due to the varying nature of the data distributions and imbalances.

2 Methodology

2.1 Performance Metrics

In this section, we describe the four performance metrics that were used to evaluate the classifiers: recall, precision, F1 score, and balanced accuracy. For the hyperparameter tuning and comparison between classifiers, we focused on optimizing the F1 score.

2.1.1 Recall

Recall, also known as sensitivity or true positive rate, measures the proportion of actual positives that are correctly identified. It is defined as:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Where:

- TP = True Positives (correctly predicted positive cases)
- FN = False Negatives (actual positives that were predicted as negative)

Advantage: Recall is especially useful when the cost of missing a positive instance is high, making it an essential metric in imbalanced datasets.

2.1.2 Precision

Precision measures the proportion of predicted positive instances that are actually positive. It is defined as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Where:

- TP = True Positives
- FP = False Positives (actual negatives that were predicted as positive)

Advantage: Precision is important when the cost of a false positive is high, helping reduce unnecessary interventions or misclassifications.

2.1.3 F1 Score

The F1 score is the harmonic mean of precision and recall, balancing the two metrics. It provides a single score that accounts for both false positives and false negatives:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Advantage: The F1 score is useful when we seek a balance between precision and recall. It is particularly effective when dealing with imbalanced datasets, as it emphasizes minimizing both false positives and false negatives.

2.1.4 Balanced Accuracy

Balanced accuracy is the average of recall (sensitivity) for the positive class and recall for the negative class. It provides a better measure of performance on imbalanced datasets:

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right)$$

Where:

- TN = True Negatives (correctly predicted negative cases)
- FP = False Positives

Advantage: Balanced accuracy helps address the bias towards the majority class in imbalanced datasets, ensuring that performance on both the positive and negative classes is taken into account.

2.1.5 Optimization Strategy

For the purpose of this project, we optimized the classifiers based on the F1 score. This metric was used for hyperparameter tuning and for comparing the performance of different models. The F1 score was chosen because it provides a balance between precision and recall, which is critical when working with imbalanced datasets.

2.2 Algorithms

In this section, we describe the algorithms used in the project, starting from simple linear models to more advanced ensemble methods. We will explain how each algorithm works, provide pseudocode, and describe how it addresses issues like overfitting, variance, and bias.

2.2.1 Penalized Logistic Regression

Penalized Logistic Regression is a linear model for binary classification, where regularization is applied to prevent overfitting. It estimates the probability that a given instance belongs to a class using the logistic function:

$$h_\beta(x_i) = \frac{1}{1 + e^{-\beta^T x_i}}$$

To prevent overfitting, regularization is introduced in the objective function, where the penalty is applied on the magnitude of the coefficients:

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \left[-\sum_{i=1}^{n} \left(y_i \log(h_\beta(x_i)) + (1 - y_i) \log(1 - h_\beta(x_i))\right) + \lambda \|\beta\|_p \right]$$

Where $p = 1$ for Lasso (L1 regularization) and $p = 2$ for Ridge (L2 regularization). The regularization parameter λ controls the trade-off between fitting the training data well and keeping the model simple.

2.2.2 Simple Decision Tree as a Baseline

The Simple Decision Tree is used as the baseline model for comparison. A Decision Tree recursively partitions the input space into subsets based on feature splits. Each split is chosen to maximize the information gain or minimize the Gini index. The model learns these partitions based on the training data and makes predictions based on the majority class in the leaves.

Although Decision Trees are intuitive and interpretable, they are prone to overfitting, especially when the tree grows too deep. This makes it a useful baseline to demonstrate how more advanced models (Bagging and Boosting) improve upon the performance of simple models.

2.2.3 Bagging (Bootstrap Aggregating)

Bagging is an ensemble method designed to reduce the variance of a single model (in this case, decision trees). It generates multiple versions of the original model by training each model on a different bootstrapped sample of the data (random sampling with replacement). The predictions from these individual models are aggregated (by majority voting for classification or averaging for regression) to form a final prediction.

Bagging is especially useful when individual models are prone to overfitting, as it smooths out the errors from individual models by averaging them.

Pseudocode for Bagging:

```
Input: Training set {(x_i, y_i)}, number of base models T
Output: Aggregated model

1. For t = 1 to T do:
   a. Create a bootstrap sample S_t from the training set
   b. Train a base model h_t on S_t
2. For each test instance, aggregate predictions from all models:
   a. Classification: Use majority voting
   b. Regression: Use averaging
3. Return the final aggregated model
```

2.2.4 Random Forest

Random Forest is an improvement on Bagging, where two types of randomization are applied. First, each decision tree is trained on a different bootstrapped sample of the data, as in Bagging. Second, at each split in the decision tree, only a random subset of features is considered. This "double sampling" (sampling data and features) reduces the correlation between trees, improving the robustness and accuracy of the model.

Random Forest addresses the high variance problem of decision trees while also preventing overfitting through feature randomness.

Pseudocode for Random Forest:

```
Input: Training set {(x_i, y_i)}, number of trees T, number of features F
Output: Random Forest model

1. For t = 1 to T do:
   a. Create a bootstrap sample S_t from the training data
   b. Train a decision tree on S_t, at each split only F randomly chosen features used
2. Aggregate predictions across all T trees using majority voting
3. Return the Random Forest model
```

2.2.5 Gradient Boosting

Gradient Boosting is a sequential ensemble method where models are trained one after another, each one correcting the errors made by the previous models. The objective function is minimized using

gradient descent. Unlike Bagging, which builds models independently, Gradient Boosting builds models iteratively, with each model trying to improve upon the residuals of the previous models.

It is especially useful for reducing bias and can capture complex patterns in data.

Pseudocode for Gradient Boosting:

```
Input: Training set {(x_i, y_i)}, number of iterations M, learning rate
Output: Boosted model F(x)

1. Initialize F(x) with a constant prediction
2. For m = 1 to M do:
   a. Compute residuals r_i = y_i - F(x_i)
   b. Train a base model h_m on the residuals
   c. Update F(x): F(x) = F(x) +  * h_m(x)
3. Return the final model F(x)
```

2.2.6 XGBoost

XGBoost is an optimized version of Gradient Boosting that incorporates several enhancements to improve speed and performance:

- **Regularization**: XGBoost adds L1 and L2 regularization to the objective function, which helps control overfitting.
- **Parallelization**: XGBoost builds trees in parallel, making it faster than traditional Gradient Boosting.
- **Handling Missing Data**: XGBoost can efficiently handle missing data by learning optimal splitting directions.

The optimization techniques in XGBoost make it faster and more robust than traditional Gradient Boosting, making it a highly effective model for large-scale and complex datasets.

2.2.7 Stacking

Stacking is a meta-learning technique where multiple base models are trained, and their predictions are combined using a meta-model. The base models provide diverse predictions (e.g., using Logistic Regression, Random Forest, etc.), and the meta-model (typically another Logistic Regression model) learns how to best combine these predictions.

Stacking can capture diverse learning patterns from different models, combining them to produce a more accurate final prediction.

Pseudocode for Stacking:

```
Input: Training set {(x_i, y_i)}, base models {h_1, h_2, ..., h_B}, meta-model g
Output: Stacked model

1. Train each base model h_b on the training data
2. For each test instance, collect predictions from all base models
3. Train the meta-model g on the predictions of the base models
4. For each test instance, predict using the meta-model based on base models' predictions
5. Return the final stacked model
```

3 Experimental Protocol

In this section, we describe the datasets, the experiments conducted, and the hyperparameters used for each model. The primary goal of this section is to enable the reader to reproduce the results presented in the next sections. We describe the datasets in detail, including their class imbalance ratios, followed by the models, their respective hyperparameters, and the optimization process. Finally, we explain the evaluation methodology, including the use of cross-validation, and discuss the parallelization strategy employed for efficient computation.

3.1 Datasets

We used four datasets, each representing a binary classification problem with varying degrees of class imbalance. The datasets were split into training and test sets, with the test set used for final evaluation. The relevant details of the datasets, such as their sample sizes and class 1 ratios, are summarized in Table 1.

Dataset	Training Set Size	Test Set Size	Feature Count	Class 1 Ratio (%)
Dataset 0	41,058	13,686	51	10.0%
Dataset 1	41,058	13,686	51	9.95%
Dataset 2	41,058	13,686	51	9.83%
Dataset 3	41,058	13,686	51	9.81%

Table 1: Summary of datasets used in the experiments. The "Class 1 Ratio" indicates the percentage of instances belonging to the minority class.

The datasets do not differ in terms of the number of features, training and testing sets size. For the class imbalance, the difference is very small. We can make the hypothesis that these 4 datasets come from the same data distribution (Fraud Detection). We will go through various techniques, such as oversampling, undersampling, and cost-sensitive learning and we will test each one of these techniques on one dataset.

3.2 Models and Hyperparameters

We evaluated several machine learning models, including both individual models and ensemble techniques. For each model, we tuned relevant hyperparameters using grid search cross-validation. The hyperparameters and their corresponding ranges are shown in Table 2.

Model	Hyperparameters
Penalized Logistic Regression	C: [20, 30, 40, 50], max_iter: [10,000]
Simple Decision Tree	max_depth: [None, 10, 20]
Bagging	n_estimators: [10, 50, 100]
Random Forest	n_estimators: [50, 100, 200], max_depth: [None, 10, 20]
Gradient Boosting	learning_rate: [0.01, 0.1, 0.2], n_estimators: [50, 100], max_depth: [3]
XGBoost	learning_rate: [0.01, 0.1], n_estimators: [50, 100], max_depth: [3]
Stacking	No hyperparameter tuning for stacking in this experiment

Table 2: Hyperparameters used for each model during grid search cross-validation.

3.2.1 Penalized Logistic Regression

This model applies L2 regularization to prevent overfitting. The regularization strength, controlled by the hyperparameter C, was tuned over a range of values. Higher values of C reduce the regularization effect, while lower values enforce stronger regularization.

3.2.2 Simple Decision Tree

We used a decision tree as a baseline model for comparing other ensemble methods. The tree's maximum depth was tuned to control its complexity. Trees with deeper depth tend to overfit the data, while shallow trees may underfit.

3.2.3 Bagging

Bagging reduces variance by training multiple decision trees on different bootstrapped subsets of the data. The number of base estimators (decision trees) was tuned to evaluate its impact on the performance. Higher numbers of estimators generally lead to better stability but increased computational cost.

3.2.4 Random Forest

Random Forest is an advanced version of Bagging that also introduces feature randomness during tree construction. We tuned the number of trees (n_estimators) and the maximum depth of the trees (max_depth). By limiting the depth, we control overfitting, while increasing the number of trees can lead to better generalization.

3.2.5 Gradient Boosting

Gradient Boosting trains models sequentially, with each new model correcting the errors of the previous one. The key hyperparameters are the learning rate, which controls how much each model contributes, and the number of trees (n_estimators). We kept the tree depth fixed at 3 to prevent overfitting.

3.2.6 XGBoost

We use same hyperparameters as in Gradient Boosting.

3.2.7 Stacking

We do not tune any hyperparameter since stacking is already computationally expensive.

3.3 Evaluation Methodology

For each dataset, we performed a train-test split, with 90% of the data used for training and 10% used for hyperparameter tuning. During training, we employed 3-fold cross-validation to tune hyperparameters and evaluate model performance. For each combination of model and dataset, the performance was assessed using four metrics: F1 Score, Precision, Recall, and Balanced Accuracy. These metrics are particularly important for imbalanced datasets, as they provide a more comprehensive evaluation than standard accuracy.

The hyperparameter tuning was performed using grid search, where each possible combination of hyperparameters was evaluated using cross-validation, and the best set of hyperparameters was selected based on the mean F1 Score.

3.4 Parallelization

To efficiently evaluate seven models across four datasets (7*4=28 cases), we parallelized the training and evaluation process using Python's `joblib` library. This allowed us to distribute the workload across multiple CPU cores, drastically reducing the overall computation time. Parallelization was very beneficial in the project. We had a 60 core processors Virtual Machine and we used 59 cores to perform the training/ testing. It cut the time spent on computation by minimum a factor of 28 ! The time that was spent on total to train and test all combinations of models and datasets was less than 3 minutes !

```
# Example code for parallelization using joblib
from joblib import Parallel, delayed
results = Parallel(n_jobs=-1)(delayed(evaluate_model)(model, X_train,
y_train, X_test, y_test)  for model in models for data in datasets)
```

This parallelization strategy ensured that the experiments were completed in a reasonable amount of time, while still performing a thorough evaluation of each model and hyperparameter combination. The detailed results of these experiments are presented in the next section.

4 Results

In this section, we present and analyze the results obtained from the seven models applied to the four datasets. The performance of each model is evaluated using several metrics : including F1 Score, Precision, Recall, and Balanced Accuracy. Additionally, we consider the computational cost of each algorithm in terms of the elapsed time. We also present the sampling method. Each sampling method was used for a different dataset.

The results for each model and dataset are summarized in Table 3, which provides a comparison of the models across different datasets using the performance metrics, sampling techniques and elapsed time.

Sampling	Model	F1 Score	Precision	Recall	Balanced accuracy	Elapsed Time (s)
none	Penalized Logistic Regression	0.653	0.818	0.543	0.765	16.829
none	Simple Decision Tree	0.712	0.793	0.645	0.814	0.734
none	Bagging	0.784	0.880	0.706	0.848	46.843
none	Random Forest	0.760	0.894	0.662	0.827	25.717
none	Boosting	0.701	0.854	0.594	0.792	23.755
none	XGBoost	0.702	0.857	0.594	0.792	1.936
none	Stacking	0.783	0.874	0.709	0.849	15.400
undersampling	Penalized Logistic Regression	0.545	0.407	0.822	0.846	4.272
undersampling	Simple Decision Tree	0.478	0.334	0.845	0.831	0.247
undersampling	Bagging	0.617	0.478	0.869	0.883	6.161
undersampling	Random Forest	0.628	0.491	0.869	0.886	8.588
undersampling	Boosting	0.583	0.443	0.851	0.867	6.048
undersampling	XGBoost	0.585	0.446	0.848	0.867	0.868
undersampling	Stacking	0.636	0.499	0.876	0.890	5.200
oversampling	Penalized Logistic Regression	0.561	0.432	0.802	0.841	43.310
oversampling	Simple Decision Tree	0.678	0.657	0.700	0.829	2.442
oversampling	Bagging	0.760	0.839	0.695	0.840	155.174
oversampling	Random Forest	0.763	0.834	0.703	0.844	62.532
oversampling	Boosting	0.711	0.751	0.675	0.825	81.535
oversampling	XGBoost	0.680	0.667	0.694	0.827	4.040
oversampling	Stacking	0.766	0.833	0.709	0.847	37.246
cost˙sensitive	Penalized Logistic Regression	0.550	0.418	0.804	0.838	23.886
cost˙sensitive	Simple Decision Tree	0.635	0.519	0.819	0.866	0.723
cost˙sensitive	Bagging	0.742	0.897	0.633	0.812	38.116
cost˙sensitive	Random Forest	0.680	0.593	0.796	0.867	14.990
cost˙sensitive	Boosting	0.703	0.874	0.587	0.789	23.047
cost˙sensitive	XGBoost	0.618	0.488	0.845	0.872	1.944
cost˙sensitive	Stacking	0.706	0.594	0.869	0.901	16.930

Table 3: Performance scores comparison of different models and sampling techniques

4.1 Analysis of Results

In this section, we analyze the performance of various machine learning algorithms on imbalanced datasets with different sampling techniques. The metrics considered are F1 Score, Precision, Recall, Balanced Accuracy, and Elapsed Time. These metrics provide a holistic view of model performance, especially for classification tasks with imbalanced data.

4.1.1 Effect of Sampling Techniques on Performance

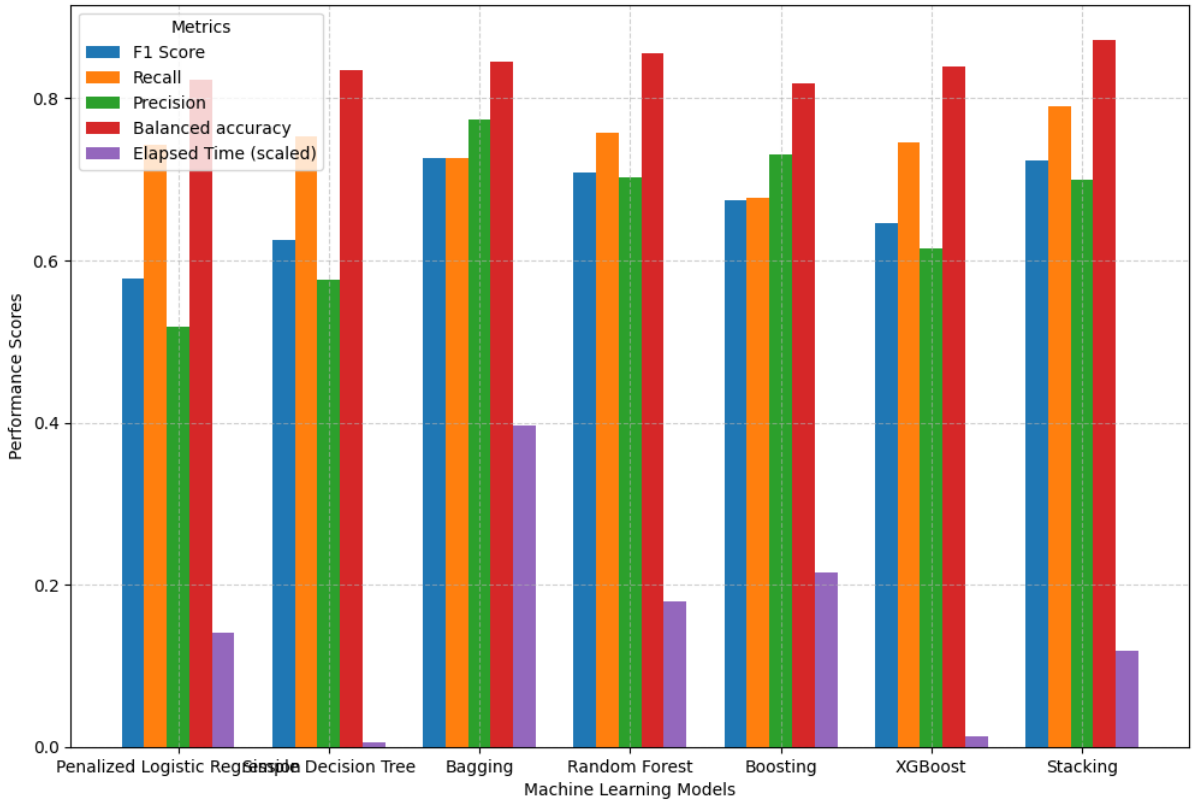

Figure 1: Performance Scores Comparison of Different Machine Learning Models

- **No Sampling:** As seen in Table 3, when no sampling technique is applied, Bagging achieves the highest F1 Score (0.784), followed by Random Forest (0.760) and Stacking (0.783). The precision is also high for these methods, indicating that they are good at correctly classifying the positive class. However, the recall values for some models, like Penalized Logistic Regression and Simple Decision Tree, are relatively low, indicating that these models fail to identify many positive instances.

- **Undersampling:** With undersampling, the recall improves significantly across all models, especially for Random Forest (0.869) and Bagging (0.869). However, this comes at the cost of reduced precision, especially for models like Simple Decision Tree (0.407) and Penalized Logistic Regression (0.407). This trade-off between precision and recall is a typical effect of undersampling, as the model is trained on a smaller, more balanced dataset, which may not represent the minority class well.

- **Oversampling:** The oversampling technique, particularly SMOTE, improves F1 Scores and recall across all models. Random Forest (0.763) and Bagging (0.760) again perform well, showing their robustness to both balanced and imbalanced datasets. The precision remains high, particularly for Bagging (0.839), indicating that oversampling helps in identifying the minority class without losing precision.

- **Cost-Sensitive Learning:** Cost-sensitive methods show a balance between precision and recall, particularly in models like XGBoost (Precision: 0.488, Recall: 0.848). These models perform better in scenarios where the cost of misclassification is high. For example, Stacking achieves a balanced accuracy of 0.869 and an F1 Score of 0.706, indicating that it manages to balance the trade-off between precision and recall effectively.

4.1.2 Comparison of Algorithms

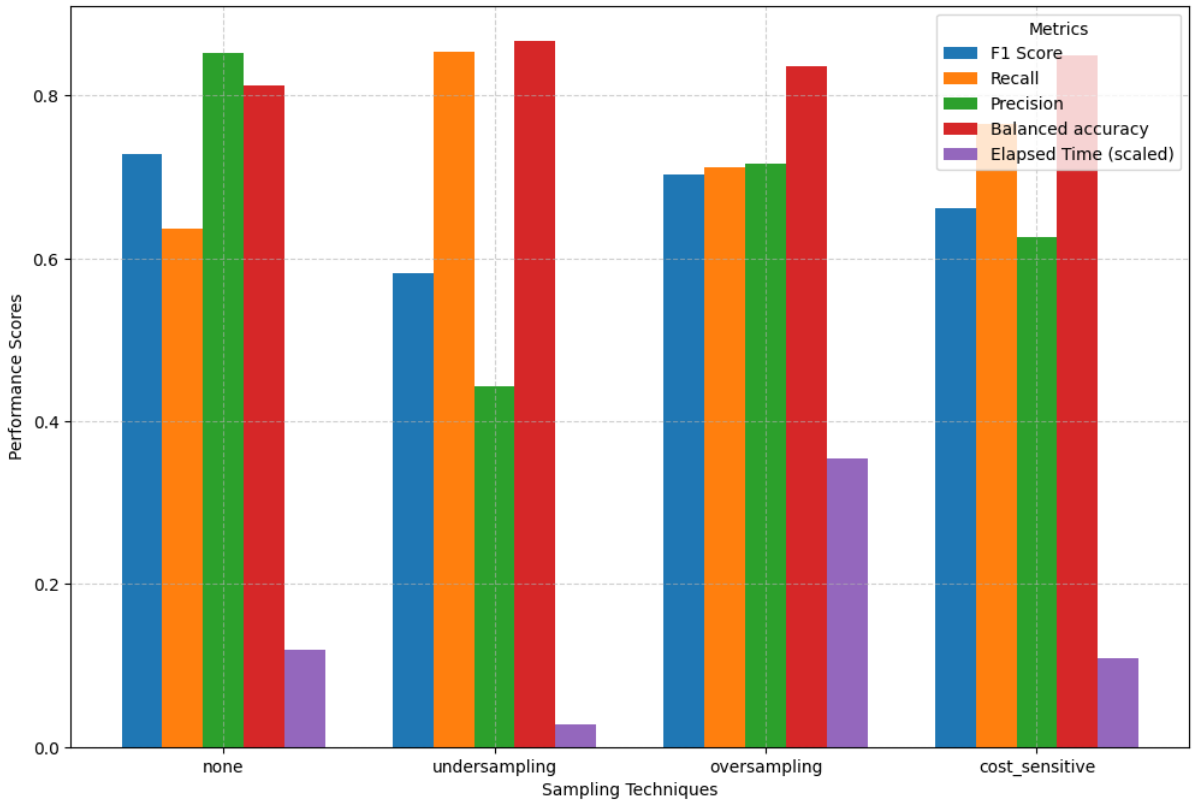

Figure 2: Performance Scores Comparison with Different Sampling Techniques

The performance of the algorithms can be compared using the following metrics:

F1 Score:

- Bagging and Random Forest consistently achieve the highest F1 Scores across most sampling techniques. This highlights their effectiveness in handling imbalanced datasets. Bagging achieves an F1 Score of 0.784 without sampling and 0.760 with oversampling. Random Forest performs similarly, with F1 Scores of 0.760 (no sampling) and 0.763 (oversampling).
- Stacking also performs well, with an F1 Score of 0.783 without sampling and 0.706 with cost-sensitive learning.

Precision:

- Bagging and Stacking show the highest precision across most sampling techniques. Without sampling, Bagging achieves a precision of 0.880 and Stacking achieves 0.874. Oversampling boosts the precision of Bagging to 0.839, making it a reliable method for imbalanced datasets.

Recall:

- Undersampling significantly improves recall for all models, especially Random Forest (0.869) and Bagging (0.869). This indicates that these models are able to capture more of the positive class with undersampling, although this comes at a cost to precision.
- Cost-sensitive learning also improves recall, with Stacking and XGBoost achieving high recall values of 0.869 and 0.848, respectively.

Balanced Accuracy:

- Stacking and Random Forest achieve the highest balanced accuracy across all techniques. For example, without sampling, Stacking achieves a balanced accuracy of 0.849, which improves slightly to 0.901 with cost-sensitive learning.

Elapsed Time:

- Bagging and Random Forest are among the slowest algorithms, with Bagging taking 46.843 seconds and Random Forest taking 25.717 seconds without sampling. XGBoost is the fastest, particularly with no sampling (0.868 seconds) and cost-sensitive learning (1.944 seconds).

4.2 Conclusion

Overall, Bagging, Random Forest, and Stacking are the top-performing models in terms of F1 Score and balanced accuracy. However, the choice of sampling technique has a significant impact on the performance metrics. Oversampling and cost-sensitive learning seem to provide the best trade-off between precision and recall, particularly for Random Forest and XGBoost. For time-sensitive tasks, XGBoost is the most efficient model, providing a good balance between performance and speed.

5 Domain Adaptation

In many real-world applications, collecting labeled data in the target domain can be challenging, time-consuming, and expensive. However, if there is sufficient labeled data available in a source domain, a model trained on this source data can be adapted to generalize well to the target domain. Domain adaptation addresses this by aligning the model's learning from the source domain with the target domain, where labels are not available during training.

In our case, the source domain had labeled data, while the target domain did not. To maximize the utility of the available source data, we attempted domain adaptation using a method called *pseudo-labeling*. This technique involves training a model on the source data and using it to generate "pseudo-labels" for the unlabeled target data. These pseudo-labels are then used to fine-tune the model, helping it adjust to the target domain's characteristics.

5.1 Similarity Between Source and Target

To assess the similarity between the source and target domains, we calculated the Maximum Mean Discrepancy (MMD) score. MMD is a statistical test that measures the difference between two distributions. A lower MMD score indicates higher similarity between the two distributions, while a higher score suggests a greater difference.

In this case, the calculated MMD score between the source and target datasets was:

$$\text{MMD Score} = 0.00238$$

This low MMD score suggests that the source and target domains are fairly similar in terms of their feature distributions, which indicates that domain adaptation has the potential to work effectively in this context.

5.2 Steps Taken for Domain Adaptation

The following steps were taken to perform domain adaptation in our project:

1. **Train on Source Data**: We first trained an XGBoost classifier on the labeled source dataset. This model achieved the following performance on the source test set:
   - **F1 Score**: 0.7327
   - **Precision**: 0.8539
   - **Recall**: 0.6416
   - **Balanced Accuracy**: 0.8150
2. **Generate Pseudo-Labels**: We used the trained model to predict pseudo-labels for the target training data, setting a confidence threshold to filter out low-confidence predictions. In this experiment, we set the confidence threshold to 40%, meaning the low predictions were not retained for fine-tuning.
3. **Fine-Tune the Model**: After generating the pseudo-labels, we combined the pseudo-labeled target data with the source data and fine-tuned the XGBoost model using this combined dataset.

4. **Evaluate Performance on Target Test Set**: Finally, we evaluated the performance of the fine-tuned model on the target test set and compared it with the performance before domain adaptation.

5.3 Results

The results of the domain adaptation process are as follows:

- **Before Domain Adaptation** (trained on source data):
  - **F1 Score**: 0.6330
  - **Precision**: 0.6817
  - **Recall**: 0.5909
  - **Balanced Accuracy**: 0.7859
- **After Domain Adaptation** (with pseudo-labeling):
  - **F1 Score**: 0.6334
  - **Precision**: 0.8131
  - **Recall**: 0.5188
  - **Balanced Accuracy**: 0.7552

5.4 Discussion of Results

The results indicate that after domain adaptation, the precision of the model increased significantly from 0.6817 to 0.8131, meaning the model became more confident in its positive class predictions. However, this improvement came at the cost of recall, which dropped from 0.5909 to 0.5188. This decrease in recall suggests that the model became more conservative in making positive predictions, leading to fewer correctly identified positive examples.

The F1 Score remained almost unchanged (0.6330 before vs. 0.6334 after), indicating that the trade-off between precision and recall balanced out in terms of overall performance. Additionally, the balanced accuracy also slightly decreased after domain adaptation, from 0.7859 to 0.7552, suggesting that the model's ability to handle both classes evenly was slightly affected.

In summary, while domain adaptation improved the precision of the model, it also caused a drop in recall, making the model more selective but less sensitive to positive instances. Further experimentation, such as tuning the confidence threshold or using other adaptation methods, may be needed to improve both precision and recall simultaneously.

6 Conclusion

In this project, we explored various ensemble learning methods to classify an imbalanced dataset and subsequently performed domain adaptation to transfer the model's learning to a new target domain. We evaluated the performance of different models, including Random Forest, Bagging, Boosting, and XGBoost, under different sampling techniques such as oversampling, undersampling, and cost-sensitive learning. The performance was assessed using several metrics, including F1 Score, Precision, Recall, and Balanced Accuracy.

From our experimentation with ensemble methods, we found that models such as Bagging and Stacking achieved strong performance on the source domain. These methods were able to effectively handle the class imbalance problem by either leveraging sampling techniques or adjusting the class weights in cost-sensitive learning. Additionally, the Random Forest and XGBoost models demonstrated robust performance in terms of precision and recall balance.

However, there is room for improvement in the following areas:

- **Hyperparameter Tuning**: Optimizing more diverse parameters such as the number of trees, learning rates, and depth of trees could result in improved performance.
- **Handling Class Imbalance**: Experimenting other advanced sampling methods may help improve recall without sacrificing precision.