Quantifying Landmine Classifier Stability via BC_a Confidence Intervals

Andreas Larsson

December 2025

Abstract

Accurate classification of landmines using passive magnetic sensors is essential for safe disarmament, yet data collection in active operations remains hazardous and costly. This study investigates the minimum sample size (n) required to achieve stable predictive performance using a Random Forest classifier. The second-order accurate Bias Corrected and Accelerated (BC_a) bootstrap methodology is used to evaluate estimator stability across both non-parametric and parametric designs. Method validation via a synthetic consistency check confirmed a 0.94 coverage probability for the BC_a interval. Experimental results on a benchmark landmine dataset indicate that while stability improves with increased data, marginal gains diminish significantly beyond $n = 200$. At this threshold, the non-parametric 95% CI reaches a practical width of 0.0914 with a mean ROC AUC of 0.8220. Based on the experimental results, $n = 200$ is recommended as the optimal balance between classification certainty and operational safety, providing a benchmark for data acquisition in mine-affected regions.

1 Introduction

Landmine detection traditionally relies on active and passive methods. Active sensors are highly accurate but dangerous as they transmit electrical signals that can trigger explosives. Passive detection, which reads magnetic field anomalies, is safer but historically less accurate in classification. While 84% of mine detections are successful, only 57.7% of the mine classifications are correct [1]. Identifying the whereabouts of a landmine is a crucial first step into reducing deaths, but an accurate description of the landmine is an important follow-up to determine which course of action is needed to disarm or controllably detonate it. Research has shown that integrating sensor height and soil type can address this classification gap. Yilmaz et al. [1] achieved a 98.2% detection and 85.8% classification rate in controlled environments using these features. However, gathering such data in active operations is costly and hazardous, leading to sparse datasets.

For a targeted area of operations, it is crucial to determine the minimum sample size (n) required for stable model performance to minimise data collection risks. This study utilizes the dataset from Yilmaz et al. [1] to evaluate the stability of a Random Forest (RF) classifier using the second-order accurate Bias Corrected and Accelerated Confidence Interval (BC_a CI).

To ensure statistical rigour, the study first performs a consistency evaluation on synthetic data to verify that the BC_a interval achieves nominal coverage for the RF-based AUC score. Subsequently, both non-parametric and parametric bootstrap is applied on the landmine data. The aim is to compare and identify a convergence point where additional samples no longer significantly outweigh the costs of data acquisition, providing a recommendation for operational sample sizes.

2 Data

The original dataset, published by Kahraman et al. [2], consist of 338 samples with three features and one target class which are described in table 1. Before making the dataset public, the authors standard scaled all three features where the categorical feature was a numerical label from zero to six. To enhance performance, the soil type feature was one hot encoded so there would not be any distance or order relation interpreted by the model for a single categorical feature. This increased the number of features from three to eight.

Table 1: Description of features and target variables in the Land Mines dataset.

Name	**Type**	**Description**
Voltage	Continuous	Output voltage value from the FLC sensor due to magnetic distortion caused by the mine (measured in Volts).
Height	Continuous	The height of the sensor above the ground (measured in centimetres).
Soil Type	Categorical	The type of soil and moisture condition during measurement. There are 6 distinct types: dry & sandy, dry & humus, dry & limy, humid & sandy, humid & humus, humid & limy.
Mine Type	Target	The class label representing the type of landmine. There are 5 different mine classes (integers 1–5) corresponding to common mine types.

3 Method and result

3.1 Model selection

The original paper used a combined model built on a neural network and heuristic fuzzy k-NN. As no clear configuration details were provided, this experiment uses a Random Forest (RF) classifier instead. The RF is well-suited for this problem because its internal bagging mechanics and high parallelism make it computationally efficient for the thousands of bootstrap iterations required, and it is known to have low variance and work effectively with sparse data.

To determine the optimal parameter settings for the RF, a grid search was run on the training data using 10-fold cross-validation. The results of the grid search are then used as the fixed configuration for all subsequent bootstrap experiments.

3.2 Method validation: Consistency evaluation

Before applying the bootstrap method to the landmine dataset, a simulation study was conducted to verify the consistency and coverage probability of the BC_a confidence interval for the Random Forest model.

A synthetic population of $N = 10,000$ samples was generated with 8 features (4 informative) and 6 classes to mimic the dimensionality of the target problem. To account for the learning curve effects inherent in small sample sizes, the "True Theoretical AUC" was defined as the expected performance of the estimator at $n = 100$. This was calculated by drawing 100 random subsamples of size $n = 100$ from the population, training a model on each, and averaging their unbiased Out-of-Bag (OOB) scores, resulting in a benchmark target of AUC ≈ 0.8341.

The consistency check involved 100 independent simulations. In each simulation, a single sample of $n = 100$ was drawn, and a BC_a interval ($B = 500$) was constructed. The coverage probability was calculated as the proportion of intervals that successfully captured the True Theoretical AUC. The simulation yielded a coverage probability of 0.94, which aligns closely with the nominal confidence level of 0.95. This result confirms that the BC_a intervals are correctly centred and provide a consistent estimate of the uncertainty for the $\text{ROC}_{\text{AUC}_{\text{OvR}}}$ metric in this domain.

3.3 Experiment setting

The main experiment aims to determine the minimum sample size (n) required for stable performance. To provide a rigorous analysis of stability, two distinct bootstrap approaches were employed:

1. **Non-Parametric Bootstrap (Random Design):** Samples are drawn with replacement from the dataset pairs (X, y). This assumes both the features and targets are random variables and accounts for the variability in data sampling.
2. **Parametric Bootstrap (Fixed Design):** The feature set X is held fixed. New target values y^* are generated for each sample based on the class probabilities predicted by the fitted Random Forest model ($\hat{P}(y|x)$). This isolates the variance contributed by the model's probabilistic nature.

The experiment iterates through a set of sample sizes, $ns = [25, 50, 75, 100, 150, 200, 250, n\_\text{samples}]$. For each n:

i A stratified subsample is drawn from the full dataset ($\mathbf{X}_{\text{subsample}}$) to maintain class balance.

ii The original performance estimate ($\hat{\theta}$) is obtained by fitting the RF model on $\mathbf{X}_{\text{subsample}}$ and evaluating it using the unbiased Out-of-Bag (OOB) $\text{ROC}_{\text{AUC}_{\text{OvR}}}$ score.

iii $B = 3000$ bootstrap iterations are run. For the **Parametric** approach, the training set size is maintained at n by resampling indices with replacement, and evaluation is performed on the OOB samples to ensuring strict comparability with the **Non-Parametric** approach.

The $\text{ROC}_{\text{AUC}_{\text{OvR}}}$ metric was chosen as the score because it is robust for multi-class problems (using macro-averaged One-vs-Rest) and measures the model's ability to correctly rank positive classes across all thresholds.

3.4 Choice of confidence interval

To quantify the variation produced by each sample size, a confidence interval (CI) is calculated for each subsample. Given the complex nature of the data (categorical and continuous features) and the small sample sizes, a non-parametric bootstrap method was necessary.

The Bias Corrected and Accelerated Confidence Interval (BC_a CI) was selected because it is a second-order accurate CI. This means its convergence rate to the nominal confidence level (here, 95% or $\alpha = 0.05$) is proportional to $O(1/n)$, which is significantly faster than first-order CIs ($O(1/\sqrt{n})$) like the Percentile method. This accelerated convergence provides tighter, more reliable intervals for the small sample sizes encountered in this problem. The BC_a achieves this by explicitly correcting for the bias ($\hat{w}$) and skewness/acceleration ($\hat{a}$) found in the bootstrap distribution.

3.5 Result

The experiment was executed according to the code in appendix B running versions specified in appendix A. The results, showing the CI span and performance vs. sample size, are plotted in Figure 1.

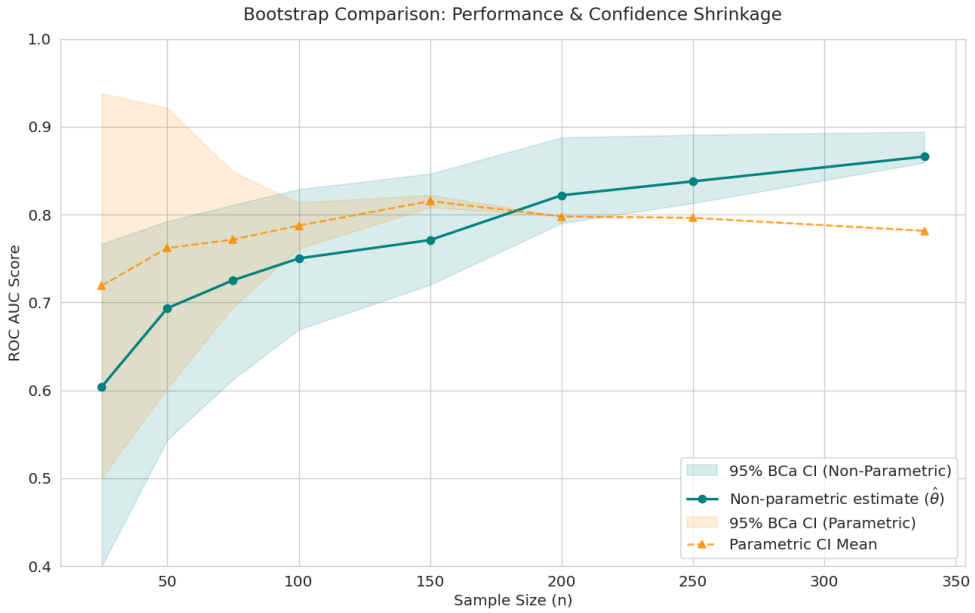

Figure 1: Model Performance and 95% BC_a Confidence Interval Shrinkage as a Function of Sample Size (n).

4 Discussion

From Figure 1, several conclusions regarding estimator stability can be drawn. Firstly, the non-parametric estimator exhibits a shift in relative position within the confidence interval (CI) as n increases. For $n < 100$, the original estimate $\hat{\theta}$ resides in the upper half of the CI, suggesting that small sample sizes are insufficient to fully capture the model's complexity, leading to a bootstrap distribution that is skewed lower. Conversely, as $n > 100$, $\hat{\theta}$ moves to the lower half of the interval. This indicates a degree of estimator optimism, likely due to the RF starting to overfit the specific data signal, which the BC_a interval correctly adjusts for via the $\hat{w}$ factor.

The parametric bootstrap provides a different perspective on model stability. For $n < 100$, the parametric intervals are notably wider than the non-parametric, reflecting the high uncertainty when training an RF model on a sparse dataset to approximate the underlying distribution. At $n \geq 200$, a numerical collapse occurs. The parametric performance remains lower than the non-parametric baseline, and the CI width reduces to zero. This is a consequence of the parametric design, while non-parametric resampling preserves the original (X, y) relationships, the parametric approach relies on labels y^* drawn from the model's own predicted probabilities. Any predictive error in the initial model is kept during this generation phase, introducing stochastic noise. This creates a significant gap between the original estimate and the bootstrap mean, resulting in a large $\hat{w}$ factor that pushes the BC_a percentile indices to the tails of the distribution, effectively collapsing the high and low bounds to the same value.

The contrasting results between the two bootstrap designs highlight a fundamental difference in how they quantify uncertainty. The non-parametric approach, following a 'Random Design' logic, resamples existing data pairs to measure the stability of the estimator relative to the sampling of the population. It proves to be the more robust method for this study, as it preserves the data signal and provides consistent convergence as n increases. In contrast, the parametric approach follows a 'Fixed Design', fixing the features X and measuring the model's sensitivity to label noise. While the non-parametric intervals are recommended for the final determination of the minimum sample size (n), the parametric intervals serve as a valuable diagnostic tool. The divergence and eventual collapse of the parametric band indicate that the RF model has not yet reached the predictive maturity required to serve as a reliable proxy for the true landmine data distribution.

Based on the non-parametric BC_a analysis, $n = 200$ is identified as the optimal sample size for practical model stability, representing the 'breaking point' where marginal gains in certainty no longer justify the hazardous costs of further data acquisition. At $n = 25$, the 95% CI width of 0.3750 confirms significant instability due to extreme data scarcity, and while stability improves as n increases, the interval at $n = 100$ (0.1608) remains too wide to ensure operational safety. Practical convergence is reached at $n = 200$, where the CI width first falls below the 0.10 threshold (0.0914) and the estimator demonstrates a robust mean performance of 0.8220 with a lower bound of 0.7903. This is further strengthened by the observed failure of the parametric approach which collapsed at $n \geq 200$. The experiment showed that the original relationship in the (X, y) pair was needed to generate accurate model predictions on that scale.

5 Conclusion

This study evaluated the stability of a Random Forest classifier for landmine detection using dual bootstrap methodologies to determine the minimum viable sample size for operational deployment. The results demonstrate that while predictive performance improves with increased data, the marginal gains in stability diminish significantly beyond $n = 200$. At this threshold, the non-parametric 95% BC_a confidence interval reaches a practical width of 0.0914 with a mean ROC AUC of 0.8220, ensuring that even the lower bound of performance remains sufficiently high for reliable classification. Furthermore, the comparative analysis revealed that the non-parametric approach is more robust for sparse landmine data, as the parametric bootstrap suffered from generator bias and numerical instability at higher sample sizes. Given the extreme hazards and high costs associated with data acquisition in mine-affected areas, $n = 200$ is recommended as the optimal balance between predictive certainty and investigator safety. This finding provides a data-driven benchmark for field operations, suggesting that resources can be most effectively allocated by focusing on high-quality collection of approximately 33 samples per class rather than pursuing larger, more hazardous datasets with diminishing statistical returns.

References

[1] C. T. Yilmaz, H. T. Kahraman, and S. Söyler, "Passive mine detection and classification method based on hybrid model," *IEEE Access*, vol. 6, pp. 47 870–47 888, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52304953

[2] H. Kahraman, "Land Mines," UCI Machine Learning Repository, 2018, DOI: https://doi.org/10.24432/C54C8Z.

A Software and versions

Table 2: Software and libraries used in the experiment. The environment was based on Python 3.11.14.

Function / Module	Library	Version
pyplot	matplotlib	3.10.7
(core array functionality)	numpy	2.3.5
read_csv, DataFrame	pandas	2.2.3
RandomForestClassifier	scikit-learn	1.7.2
GridSearchCV, train_test_split	scikit-learn	1.7.2
StratifiedShuffleSplit	scikit-learn	1.7.2
roc_auc_score, make_classification	scikit-learn	1.7.2
OneHotEncoder, UndefinedMetricWarning	scikit-learn	1.7.2
classification_report, confusion_matrix	scikit-learn	1.7.2
norm, softmax	scipy	1.16.2
(plotting style)	seaborn	0.13.2

B Code appendix

```
# %%
# Import necessary libraries
import os
import warnings

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from scipy.stats import norm
from scipy.special import softmax

from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.exceptions import UndefinedMetricWarning
from sklearn.metrics import (
    roc_auc_score,
    classification_report,
    confusion_matrix)
from sklearn.model_selection import (
    GridSearchCV,
    StratifiedShuffleSplit,
    train_test_split)
from sklearn.preprocessing import OneHotEncoder

# Set a global seed for reproducability
SEED = 42
np.random.seed(SEED)

# %%
# Since there are some very small sample sizes all probabilities
# predicted by the model must be mapped to all possible classes
# even though all may not be represented in the current sample.
# This ensures consistent estimates and execution.
def get_full_proba(model, X, all_classes):
    """Generates prediction probabilities and pads them to match all_classes."""
    y_proba_subset = model.predict_proba(X)
    known_classes = model.classes_
```

```
    full_y_proba = np.zeros((y_proba_subset.shape[0], len(all_classes)))

    # Map the model's known classes to the correct column index in the full array
    for i, class_label in enumerate(known_classes):
        target_col_index = np.where(all_classes == class_label)[0][0]
        full_y_proba[:, target_col_index] = y_proba_subset[:, i]

    return full_y_proba

# Same reason as above but this time it is the OOB scores from the
# model that are used instead of the results on a test set.
def get_oob_proba(model, y_true, all_classes):
    """
    Generates prediction probabilities from OOB scores and pads them
    to match all_classes. Converts OOB scores to proper probabilities.
    """
    if not hasattr(model, 'oob_decision_function_') or model.oob_decision_function_ is
     None:
        # Return uniform probabilities if OOB unavailable
        return np.full((len(y_true), len(all_classes)), 1/len(all_classes), dtype=
    float)

    oob_scores = model.oob_decision_function_
    known_classes = model.classes_

    # Convert OOB scores to probabilities using softmax
    oob_proba = softmax(oob_scores, axis=1)

    # Pad to match all_classes
    full_oob_proba = np.zeros((oob_scores.shape[0], len(all_classes)))
    for i, class_label in enumerate(known_classes):
        target_col_index = np.where(all_classes == class_label)[0][0]
        full_oob_proba[:, target_col_index] = oob_proba[:, i]

    return full_oob_proba


# Part of the Bias correction confidence interval. The bias correction factor,
# denoted 'w', is calculated for a bootstrap round.
def calculate_bias_correction_factor(bootstrap_estimates, original_estimate, B):
    less_than_mask = bootstrap_estimates < original_estimate
    frac = less_than_mask.sum() / B
    frac = np.clip(frac, 1e-10, 1 - 1e-10)
    return norm.ppf(frac)

# Part of the Bias correction confidence interval. The acceleration factor,
# denoted 'a', is calculated for a bootstrap round. The real estimation would
# use leave one out summations for each sample in n. To reduce data leakage
# this version uses the OOB scores from the random forest model to instead
# of using a separate test set for the evaluation.
def calculate_acceleration_factor(X, y, model_parameters, all_classes, n_jackknife=
    None):
    n = X.shape[0]
    indices = np.arange(n)
    jackknife_estimates = np.zeros(len(indices))

    for i in indices:
        # Delete the current sample from the data
        # the 'leave one out' part
        X_jack = np.delete(X, i, axis=0)
        y_jack = np.delete(y, i, axis=0)

        # If the sample is small and only one class
        # is represented the estimate is 0.5
        if len(np.unique(y_jack)) < 2:
            jackknife_estimates[i] = 0.5
        else:
```

```
            model = RandomForestClassifier(**model_parameters)
            model.fit(X_jack, y_jack.ravel())
            y_proba = get_oob_proba(model, y_jack, all_classes)

            # Suppress warning for single-class OOB edge cases
            with warnings.catch_warnings():
                warnings.simplefilter("ignore", category=UndefinedMetricWarning)
                try:
                    score = roc_auc_score(y_jack, y_proba, multi_class='ovr', average=
    'macro', labels=all_classes)
                except ValueError:
                    score = 0.5
            jackknife_estimates[i] = score

    jack_mean = np.mean(jackknife_estimates)
    numerator = np.sum((jack_mean - jackknife_estimates) ** 3)
    denominator_sq = np.sum((jack_mean - jackknife_estimates) ** 2)

    # Avoid division by zero or a very small number which would
    # only introduce numerical instability. In those cases the
    # acceleration is approximately zero.
    if denominator_sq < 1e-10:
        return 0.0

    denominator = 6 * (denominator_sq ** 1.5)
    return numerator / denominator


# %%
rf_synthetic_params = {
    'n_estimators': 100,
    'max_depth': 10,
    'min_samples_split': 5,
    'min_samples_leaf': 2,
    'oob_score': True,
    'random_state': SEED,
    'n_jobs': -1
}

# Create Synthetic Data
# Make it lrage (N=10,000) to find the "True" Theoretical AUC
X_syn, y_syn = make_classification(n_samples=10000, n_features=8, n_informative=4,
    n_classes=6, random_state=SEED)
# Count the number of classes in the full dataset
all_classes = np.unique(y_syn)

# Define number of samples for subsampling
n_samples = 100

# Calulate the "True" Theoretical AUC by
# repeatedly sampling from the large synthetic dataset.
truths = np.zeros(n_samples)
for i in range(n_samples):
    idx = np.random.choice(range(10000), size=n_samples)

    X_subsample, y_subsample = X_syn[idx], y_syn[idx]

    # Calculate "True" AUC
    rf_true = RandomForestClassifier(**rf_synthetic_params)
    rf_true.fit(X_subsample, y_subsample)

    # Use the OOB probability function to get unbiased scores
    y_probs_oob = get_oob_proba(rf_true, y_subsample, all_classes)

    truths[i] = roc_auc_score(y_subsample, y_probs_oob, multi_class='ovr', average='
    macro', labels=all_classes)

TRUE_AUC = truths.mean()
```

```
print(f"True Theoretical AUC: {TRUE_AUC}")

# Simulation Loop (Coverage Check)
# Take small samples (n=100) and see if the BCa interval captures the True AUC
n_simulations = 100

coverage_count = 0

# Set the confidence level of the CI
alpha = 0.05
# Set number of bootstrap iterations
B = 500

for i in range(n_simulations):
    # Draw a small sample (n=100) from the big population
    idx = np.random.choice(range(10000), size=n_samples)
    X_subsample, y_subsample = X_syn[idx], y_syn[idx]

    # Initiate variable for the bootstrap loop
    bootstrap_roc_aucs = np.zeros(B)
    y_subsample_flat = y_subsample.ravel()

    # Calculate the original estimate (theta hat) for the subsample
    # Use the oob proba for the test score.
    rf_subsample = RandomForestClassifier(**rf_synthetic_params)
    rf_subsample.fit(X_subsample, y_subsample.ravel())
    y_subsample_proba = get_oob_proba(rf_subsample, y_subsample, all_classes)

    # Hide warning which may occur for small n
    with warnings.catch_warnings():
        warnings.simplefilter("ignore", category=UndefinedMetricWarning)
        subsample_roc_auc_estimate = roc_auc_score(y_subsample, y_subsample_proba,
    multi_class='ovr', average='macro', labels=all_classes)


    # Create list of subsample indices
    indices_pool = np.arange(n_samples)
    # Create list of all class indices
    class_indices = {c: np.where(y_subsample_flat == c)[0] for c in np.unique(
    y_subsample_flat)}


    for b in range(B):
        # Generate the bootstrap sample and use the class indices to
        # ensure a proportional sample of classes are drawn from the
        # original subsample. The draws are with replacement.
        bootstrap_indices_list = []
        for c, cls_idxs in class_indices.items():
            if len(cls_idxs) > 0:
                bootstrap_indices_list.append(np.random.choice(cls_idxs, size=len(
    cls_idxs), replace=True))

        bootstrap_idx = np.concatenate(bootstrap_indices_list)

        # The test set (oob samples) are generated based on the indices
        # which were not drawn from the original sample.
        test_idx = np.setdiff1d(indices_pool, bootstrap_idx)

        # If the draws from the test set are not representative then
        # continue to the next round.
        if len(test_idx) == 0 or len(np.unique(y_subsample[test_idx])) < 2:
            bootstrap_roc_aucs[b] = 0.5
            continue

        # Define the sets based on the previous indexing.
        X_boot_train = X_subsample[bootstrap_idx]
        y_boot_train = y_subsample[bootstrap_idx]
        X_boot_test = X_subsample[test_idx]
```

```
            y_boot_test = y_subsample[test_idx]

            # Create a model for the bootstrap iteration and fit the data.
            rf_best_bootstrap = RandomForestClassifier(**rf_synthetic_params)
            rf_best_bootstrap.fit(X_boot_train, y_boot_train.ravel())

            # Evaluate on the test set previously created.
            y_proba = get_full_proba(rf_best_bootstrap, X_boot_test, all_classes)

            # Ignore the terminal print of the warning which may occur for small n.
            with warnings.catch_warnings():
                warnings.simplefilter("ignore", category=UndefinedMetricWarning)
                try:
                    roc_auc = roc_auc_score(y_boot_test, y_proba, multi_class='ovr',
        average='macro', labels=all_classes)
                except ValueError:
                    roc_auc = 0.5

            # If the result is nan then it is replaced with a proper
            # value to ensure the continuation of the experiment.
            if np.isnan(roc_auc):
                roc_auc = 0.5

            # Save the bootstrap iteration result
            bootstrap_roc_aucs[b] = roc_auc

        # Calculate the mean for the entire bootstrap session
        boot_mean = np.mean(bootstrap_roc_aucs)

        # Calculate the bias correction and acceleration factors for the
        # bias correction confidence interval.
        w = calculate_bias_correction_factor(bootstrap_roc_aucs,
        subsample_roc_auc_estimate, B)
        a = calculate_acceleration_factor(X_subsample, y_subsample, rf_synthetic_params,
        all_classes)

        # Calculate the lambda threshold for the CI level.
        lambda_low  = norm.ppf(alpha / 2)
        lambda_high = norm.ppf(1 - alpha / 2)

        # The low and high z-scores for the percentile calculation
        z_low  = w + (lambda_low + w) / (1 - a * (lambda_low + w))
        z_high = w + (lambda_high + w) / (1 - a * (lambda_high + w))

        # The percentile calculations to determine their rank
        P_low  = norm.cdf(z_low)
        P_high = norm.cdf(z_high)

        # Find the indices corresponding to the percentile ranks
        low_index = int(np.clip(np.floor(P_low * B), 0, B - 1))
        high_index = int(np.clip(np.floor(P_high * B), 0, B - 1))

        # Sort the scores to find the lower and higher bounds
        sorted_scores = np.sort(bootstrap_roc_aucs)

        # Final CI bounds
        CI_low  = sorted_scores[low_index]
        CI_high = sorted_scores[high_index]
        CI_mean = (CI_low + CI_high) / 2

        print(f"Simulation {i+1}/{n_simulations}: BCa CI = ({CI_low:.4f}, {CI_high:.4f}),
        Mean = {CI_mean:.4f}")

        # Check if it captured the truth
        if CI_low <= TRUE_AUC <= CI_high:
            coverage_count += 1

    print(f"BCa Interval Coverage Probability: {coverage_count / n_simulations}")
```

```
# Ideally, it should be close to the 1-alpha = 0.95

# %%
#Load the data from the csv file into a pandas DataFrame.
data = pd.read_csv('./data/Land mines.csv')
# Print some information about the dataset
data.info()
data.head()

# Determine the number of unique classes in the categorical feature 'S'
ohe_classes = len(data['S'].unique())
# Prepare the feature matrix X and target vector y.
X = np.zeros(shape=(data.shape[0], 2 + ohe_classes))

# Populate the feature matrix X and target vector y
# with data. Features 'V' and 'H' are numerical and can be used directly.
X[:,0] = data['V'].values
X[:,1] = data['H'].values
# THe third feature 'S' is categorical and needs to be one-hot encoded.
X[:,2:] = OneHotEncoder(sparse_output=False).fit_transform(data['S'].values.reshape
    (-1,1))

# y is the target.
y = data['M'].values.reshape(-1, 1)

# Generate train-test split for the parameter search.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
    SEED)


# %%
# Check class balance in the full dataset
all_mines = y.ravel()
mine_types = np.unique(all_mines)
mine_types_count = dict()
for mine in all_mines:
    if mine in mine_types_count:
        mine_types_count[mine] += 1
    else:
        mine_types_count[mine] = 1

mine_types_freq = dict()
for c, count in mine_types_count.items():
    mine_types_freq[c] = np.round(count / len(all_mines), 4)

print("Class balance in the full dataset")
print("-" * 65)
for c in mine_types:
    print(f"Class: {c:<3} | Count: {mine_types_count.get(c)}  | Percentage:  {np.mean
    (100*mine_types_freq.get(c)):.2f}%")
print(f"Total: {len(mine_types):<3} | Count: {sum(mine_types_count.values())} |
    Percentage: {100*sum(mine_types_freq.values()):.2f}%")
print("-" * 65)



# %%
# Define the parameter grid
param_grid = {
    'n_estimators': [10, 25, 50, 100, 200, 300, 500],    # Number of trees
    'max_depth': [5, 10, 15, 20, None],                  # Maximum depth of the tree
    'min_samples_split': [2, 5, 10],                     # Minimum number of samples
    required to split an internal node
    'min_samples_leaf': [1, 2, 3, 4, 5],                 # Minimum number of samples
    required to be at a leaf node
    'criterion': ['gini', 'entropy', 'log_loss'],        # Function to measure the
    quality of a split
    'max_features': [0.3, 0.5, 0.7, 'sqrt', 'log2'],     # Number of features to
```

```
    consider at every split
    'class_weight': ['balanced', None]                  # Weights associated with
    classes
}

# Initiate the model and then use grid search with the parameters above
# to find the optimal setting for the data.
rf = RandomForestClassifier(random_state=SEED)
grid_search = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    scoring='roc_auc_ovr',  # Use 'roc_auc' as evaluation
    cv=10,                  # Use 10-fold Cross-Validation
    n_jobs=-1               # Split up work among available CPU cores
)

# Fit the grid search on the training data to find
# the optimal setting.
grid_search.fit(X_train, y_train.ravel())

# This is the best model
best_rf = grid_search.best_estimator_

# It is used to make a prediction on the test set
# to obtain information about the performance
y_pred = best_rf.predict(X_test)
y_probs = best_rf.predict_proba(X_test)
roc_auc = roc_auc_score(y_test, y_probs, multi_class='ovr', average='macro')

# Save the best parameters into a variable
best_params = grid_search.best_params_

# Print information about the results from the best Random Forest model
print("Confusion Matrix and Classification Report for Best Random Forest Model:")
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(f"ROC AUC Score: {roc_auc:.4f}")
print("Best Random Forest Parameters:")
print(best_params)

# %%
def BCa_bootstrap_experiment(X, y, rf_best_params, all_classes, ns, B=1000, alpha
    =0.05, bootstrap_distribution=None):
    print(f"Total Training Samples available: {X.shape[0]}")
    print(f"Using bootstrap distribution: {bootstrap_distribution}")
    print(f"Starting BCa Bootstrap (B={B})...")
    print("-" * 65)

    # Define a matrix to hold all experiment results
    n_results = np.zeros((len(ns), 6))

    # The experiment loop for each n in ns
    for idx, n in enumerate(ns):

        # Define the current set of n samples from
        # the full data.
        if n >= n_samples:
            # Instead of crashing, use the full data if n > sample size
            X_subsample = X
            y_subsample = y
            current_n = n_samples # Update correct size
        else:
            # Generate a Stratified Subsample with n datapoints from the
            # full data. The StratifiedShuffleSplit keeps the proportion of
            # samples among the classes.
            sss = StratifiedShuffleSplit(n_splits=1, train_size=n, random_state=SEED)
            for train_index, test_index in sss.split(X, y):
                X_subsample = X[train_index]
```

```
            y_subsample = y[train_index]
        current_n = X_subsample.shape[0]

    # Calculate the original estimate (theta hat) for the subsample
    # Use the oob proba for the test score.
    rf_subsample = RandomForestClassifier(**rf_best_params)
    rf_subsample.fit(X_subsample, y_subsample.ravel())
    y_subsample_proba = get_oob_proba(rf_subsample, y_subsample, all_classes)

    # Hide warning which may occur for small n
    with warnings.catch_warnings():
        warnings.simplefilter("ignore", category=UndefinedMetricWarning)
        subsample_roc_auc_estimate = roc_auc_score(y_subsample, y_subsample_proba,
 multi_class='ovr', average='macro', labels=all_classes)

    # Initiate variable for the bootstrap loop
    bootstrap_roc_aucs = np.zeros(B)
    y_subsample_flat = y_subsample.ravel()

    # Create list of subsample indices
    indices_pool = np.arange(current_n)
    # Create list of all class indices
    class_indices = {c: np.where(y_subsample_flat == c)[0] for c in np.unique(
y_subsample_flat)}

    for b in range(B):
        if bootstrap_distribution == 'non-parametric':
            # Generate the bootstrap sample and use the class indices to
            # ensure a proportional sample of classes are drawn from the
            # original subsample. The draws are with replacement.
            bootstrap_indices_list = []
            for c, cls_idxs in class_indices.items():
                if len(cls_idxs) > 0:
                    bootstrap_indices_list.append(np.random.choice(cls_idxs, size=
len(cls_idxs), replace=True))

            bootstrap_idx = np.concatenate(bootstrap_indices_list)

            # Create training set based on the indices
            X_boot_train = X_subsample[bootstrap_idx]
            y_boot_train = y_subsample[bootstrap_idx]

        if bootstrap_distribution == 'parametric':
            # Create the parametric bootstrap sample based on the model's
            # predicted probabilities for each class.
            # THe indices are drawn with replacement to keep the sample size the
same.
            bootstrap_idx = np.random.choice(indices_pool, size=current_n, replace
=True)

            # Generate new y values for the indices based on the model's
probabilities
            y_new_list = []
            for i in bootstrap_idx:
                # Use the probability from the original subsample fit for index i
                new_label = np.random.choice(all_classes, p=y_subsample_proba[i])
                y_new_list.append(new_label)

            # Create training set based on the indices
            X_boot_train = X_subsample[bootstrap_idx]
            y_boot_train = np.array(y_new_list).reshape(-1, 1)

        # The test set (oob samples) are generated based on the indices
        # which were not drawn from the original sample.
        test_idx = np.setdiff1d(indices_pool, bootstrap_idx)

        # If the draws from the test set are not representative then
        # continue to the next round.
```

```
        if len(test_idx) == 0 or len(np.unique(y_subsample[test_idx])) < 2:
            bootstrap_roc_aucs[b] = 0.5
            continue

        # Define the sets based on the previous indexing.
        X_boot_test = X_subsample[test_idx]
        y_boot_test = y_subsample[test_idx]

        # Create a model for the bootstrap iteration and fit the data.
        rf_best_bootstrap = RandomForestClassifier(**rf_best_params)
        rf_best_bootstrap.fit(X_boot_train, y_boot_train.ravel())

        # Evaluate on the test set previously created.
        y_proba = get_full_proba(rf_best_bootstrap, X_boot_test, all_classes)

        # Ignore the terminal print of the warning which may occur for small n.
        with warnings.catch_warnings():
            warnings.simplefilter("ignore", category=UndefinedMetricWarning)
            try:
                roc_auc = roc_auc_score(y_boot_test, y_proba, multi_class='ovr',
average='macro', labels=all_classes)
            except ValueError:
                roc_auc = 0.5

        # If the result is nan then it is replaced with a proper
        # value to ensure the continuation of the experiment.
        if np.isnan(roc_auc):
            roc_auc = 0.5

        # Save the bootstrap iteration result
        bootstrap_roc_aucs[b] = roc_auc

    # Calculate the mean for the entire bootstrap session
    boot_mean = np.mean(bootstrap_roc_aucs)

    # Calculate the bias correction and acceleration factors for the
    # bias correction confidence interval.
    w = calculate_bias_correction_factor(bootstrap_roc_aucs,
subsample_roc_auc_estimate, B)
    a = calculate_acceleration_factor(X_subsample, y_subsample, rf_best_params,
all_classes)

    # Calculate the lambda threshold for the CI level.
    lambda_low  = norm.ppf(alpha / 2)
    lambda_high = norm.ppf(1 - alpha / 2)

    # The low and high z-scores for the percentile calculation
    z_low  = w + (lambda_low + w) / (1 - a * (lambda_low + w))
    z_high = w + (lambda_high + w) / (1 - a * (lambda_high + w))

    # The percentile calculations to determine their rank
    P_low  = norm.cdf(z_low)
    P_high = norm.cdf(z_high)

    # Find the indices corresponding to the percentile ranks
    low_index = int(np.clip(np.floor(P_low * B), 0, B - 1))
    high_index = int(np.clip(np.floor(P_high * B), 0, B - 1))

    # Sort the scores to find the lower and higher bounds
    sorted_scores = np.sort(bootstrap_roc_aucs)

    # Final CI bounds
    CI_low  = sorted_scores[low_index]
    CI_high = sorted_scores[high_index]
    CI_mean = (CI_low + CI_high) / 2

    # Output
    print(f"n={n:<3}  | Theta_hat: {subsample_roc_auc_estimate:.4f} | Boot Mean: {
```

```
    np.mean(bootstrap_roc_aucs):.4f}")
        print(f"       | w={w:.4f} | a={a:.4f} | P_low={P_low:.4f} | P_high={P_high:.4
    f}")
        print(f"       | BCa CI: [{CI_low:.4f}, {CI_high:.4f}]")
        print("-" * 65)

        # Save the result to matrix
        n_results[idx, :] = [current_n, subsample_roc_auc_estimate, boot_mean, CI_low,
     CI_high, CI_mean]


    print(f"Experiment finished.")
    print("-" * 65)

    return n_results

# %%
# Define the best parameters, found previously, as a parameter dict
rf_best_params = {
    'n_estimators': 500,
    'criterion': 'entropy',
    'max_depth': 10,
    'min_samples_split': 2,
    'min_samples_leaf': 2,
    'max_features': 0.7,
    'class_weight': 'balanced',
    'random_state': SEED,
    'oob_score': True,
    'n_jobs': -1
}

# Count the number of classes in the full dataset
all_classes = np.unique(y)
# Count the number of samples in the full dataset
n_samples = X.shape[0]
# Set the confidence level of the CI
alpha = 0.05
# Set the number of bootstrap iterations
B = 3000
# Define a list of sample sizes to evaluate
ns = [25, 50, 75, 100, 150, 200, 250, n_samples]


# Run the experiment for Non-Parametric Bootstrap
n_non_parametric_results = BCa_bootstrap_experiment(
    X, y, rf_best_params, all_classes, ns, B=B, alpha=alpha, bootstrap_distribution='
    non-parametric'
)
try:
    # Save results to CSV
    results_df = pd.DataFrame(n_non_parametric_results, columns=[
        'n_samples', 'Theta_hat', 'Boot_Mean', 'CI_Low', 'CI_High', 'CI_Mean'
    ])
    results_df.to_csv('./experiment_non_parametric_results.csv', index=False)
except Exception as e:
    print(f"Error saving results to CSV: {e}")


# Run the experiment for Parametric Bootstrap
n_parametric_results = BCa_bootstrap_experiment(
    X, y, rf_best_params, all_classes, ns, B=B, alpha=alpha, bootstrap_distribution='
    parametric'
)
try:
    # Save results to CSV
    results_df = pd.DataFrame(n_parametric_results, columns=[
        'n_samples', 'Theta_hat', 'Boot_Mean', 'CI_Low', 'CI_High', 'CI_Mean'
    ])
```

```
    results_df.to_csv('./experiment_parametric_results.csv', index=False)
except Exception as e:
    print(f"Error saving results to CSV: {e}")

# %%
# Ensure the output directory exists
output_dir = 'images'
os.makedirs(output_dir, exist_ok=True)

# Load the dataframes
try:
    df_np = pd.read_csv('./experiment_non_parametric_results.csv')
    df_p = pd.read_csv('./experiment_parametric_results.csv')
except FileNotFoundError as e:
    print(e)

# Set style for academic reporting
sns.set_style("whitegrid")
plt.rcParams.update({'font.size': 12, 'figure.dpi': 120})

# --- Plot 1: Performance & Confidence Band Comparison ---
plt.figure(figsize=(11, 7))

# 1. Plot Non-Parametric (Original)
plt.fill_between(df_np['n_samples'], df_np['CI_Low'], df_np['CI_High'],
                 color='teal', alpha=0.15, label='95% BCa CI (Non-Parametric)')
plt.plot(df_np['n_samples'], df_np['Theta_hat'], color='teal', linewidth=2,
         marker='o', label=r'Non-parametric estimate ($\hat{\theta}$)')

# 2. Plot Parametric
plt.fill_between(df_p['n_samples'], df_p['CI_Low'], df_p['CI_High'],
                 color='darkorange', alpha=0.15, label='95% BCa CI (Parametric)')
plt.plot(df_p['n_samples'], df_p['CI_Mean'], color='darkorange', linestyle='--',
         linewidth=1.5, marker='^', alpha=0.8, label='Parametric CI Mean')

plt.title('Bootstrap Comparison: Performance & Confidence Shrinkage', fontsize=14, pad
    =15)
plt.ylabel('ROC AUC Score')
plt.xlabel('Sample Size (n)')
plt.legend(loc='lower right', frameon=True)
plt.ylim(0.4, 1.0)

plt.tight_layout()
plt.savefig(os.path.join(output_dir, 'comparison_performance_shrinkage.png'))

# --- Plot 2: Bias (Optimism) Comparison ---
plt.figure(figsize=(9, 6))

bias_np = df_np['Theta_hat'] - df_np['Boot_Mean']
bias_p = df_p['Theta_hat'] - df_p['Boot_Mean']

plt.plot(df_np['n_samples'], bias_np, color='teal', marker='s', label='Non-Parametric
    Bias')
plt.plot(df_p['n_samples'], bias_p, color='darkorange', marker='d', label='Parametric
    Bias')
plt.axhline(0, color='black', linestyle=':', linewidth=1)

plt.title('Estimator Optimism (Bias) Comparison', fontsize=14, pad=15)
plt.ylabel(r'Optimism ($\hat{\theta} - \bar{\theta}^*$)')
plt.xlabel('Sample Size (n)')
plt.legend()

plt.tight_layout()
plt.savefig(os.path.join(output_dir, 'comparison_bias.png'))

# --- Plot 3: CI Width (Uncertainty) Comparison ---
plt.figure(figsize=(9, 6))
```

```
width_np = df_np['CI_High'] - df_np['CI_Low']
width_p = df_p['CI_High'] - df_p['CI_Low']

plt.plot(df_np['n_samples'], width_np, color='teal', marker='^', label='Non-Parametric
    Width')
plt.plot(df_p['n_samples'], width_p, color='darkorange', marker='v', label='Parametric
    Width')

plt.title('Uncertainty Comparison: CI Width', fontsize=14, pad=15)
plt.ylabel('Confidence Interval Width')
plt.xlabel('Sample Size (n)')
plt.legend()

plt.tight_layout()
plt.savefig(os.path.join(output_dir, 'comparison_ci_width.png'))

print(f"Comparison plots saved to: {os.path.abspath(output_dir)}")
```