UNIVERSITY OF NEW BRUNSWICK

FREDERICTON

FACULTY OF COMPUTER SCIENCE

CS6735: MACHINE LEARNING AND DATA MINING

PROGRAMMING PROJECT REPORT

GROUP MEMBERS

PRIYANKA BHAMARE (3741282)
RISHABH KALAI (3740704)
SADHANA SURESH CHETTIAR (3733403)

Table of Contents

1. Abstract

This study evaluates the performance of four machine learning algorithms – AdaBoost with Tree Stumps, Random Forest, Artificial Neural Networks (ANN) with Backpropagation, and K Nearest Neighbor (kNN) on five diverse datasets from the UCI repository, spanning medical, automotive, and microbiology domains (Breast Cancer Wisconsin, Car Evaluation, Letter Recognition, Mushroom, and Ecoli datasets). Our primary aim is to conduct a comparative analysis of these algorithms, assessing their classification accuracy, precision, recall, and F1 score. Results highlight algorithm strengths and weaknesses across datasets, providing insights for practitioners and researchers in leveraging machine learning effectively for data analysis and decision-making.

2. Introduction

This programming project aims to explore and evaluate the performance of four distinct machine learning algorithms: AdaBoost with Tree Stumps, Random Forest, Artificial Neural Networks (ANN) with Backpropagation, and K-Nearest Neighbors (kNN) using two different distance measures. This experimental study utilizes datasets from the UCI machine learning repository, focusing on implementing these algorithms from scratch in Python and assessing their effectiveness through 10 times 5-fold cross-validation. The project seeks to deepen our understanding of each algorithm's strengths and limitations when applied to different types of data.

This report outlines the comprehensive process undertaken from the initial implementation of the algorithms to the detailed analysis of their performance across various datasets. It includes sections on the algorithms' theoretical background, implementation specifics, dataset descriptions, experimental results, and a comparative analysis of the findings. The objective is to offer insights that could guide future research in machine learning algorithm application and performance optimization.

3. Learning Algorithms Implementation

3.1 AdaBoost on Tree Stumps

Description:

AdaBoost (Adaptive Boosting) is an ensemble method fundamentally designed for binary classification problems. It sequentially trains weak learners, specifically tree stumps (a form of decision tree characterized by a single decision node and two leaves), with each learner focusing more on the instances that were previously misclassified. By doing so, AdaBoost adapts to the 'hard' examples within the training set, ensuring that the final model, which is a weighted combination of all learners, achieves strong classification performance.

Implementation Details:

- Weak Learner: The algorithm employs tree stumps due to their simplicity and effectiveness as weak learners. Each stump bases its decision on a single feature, effectively partitioning the data into two groups based on a threshold.
- Weight Update: After each training round, the weights of the training instances are updated to reflect their classification status: weights are increased for misclassified instances and decreased for correctly classified ones. This iterative weight adjustment ensures that subsequent stumps prioritize the more challenging instances.
- Combining Weak Learners: The final model aggregates the predictions of all tree stumps through a weighted vote, where each stump's vote is weighted according to its accuracy in classifying the training set. This mechanism allows the ensemble to leverage the strengths of each learner.
- Parameters: The key parameter in AdaBoost is the number of iterations, which determines how many tree stumps are included in the final model. The optimal number of iterations typically depends on the specific dataset and is usually determined through cross-validation.

3.2 Random Forest

Description:
Random Forest is a versatile ensemble learning method that can be used for both classification and regression tasks. It builds upon the concept of bagging (bootstrap aggregating) by creating a 'forest' of decision trees, each trained on a random subset of the data and features. This randomness introduces diversity among the trees, enhancing the model's ability to generalize and reduce overfitting.
Implementation Details:

- Decision Trees: The model constructs multiple decision trees, each trained on a different bootstrap sample of the training data (i.e., a sample drawn with replacement). This approach ensures that each tree learns from a varied subset of the data.
- Feature Selection: When splitting nodes during the construction of the trees, a random subset of the features is considered. This randomness helps to further increase the diversity among the trees, which is beneficial for the model's overall performance.
- Voting for Classification: For classification tasks, each tree in the forest casts a vote for the class of a given input. The class that receives the majority of votes is selected as the model's prediction.
- Parameters: Key parameters include the number of trees in the forest, the maximum number of features considered for splitting at each node, the maximum depth of the trees, and the minimum number of samples required to split a node. These parameters must be carefully tuned, often using cross-validation, to optimize the model's performance.

3.3 ANN with Backpropagation

Description:
Artificial Neural Networks (ANN) mimic the neural processing of the human brain to learn complex patterns and make predictions. The backpropagation algorithm is a cornerstone of ANN training, allowing the network to adjust its weights and biases in response to the error between the predicted and actual outcomes.
Implementation Details:

- Architecture: The architecture of an ANN is defined by its layers and the number of neurons within each layer. A typical starting point is a network with one hidden layer, though the architecture may be adjusted based on the complexity of the task.
- Activation Function: Activation functions introduce non-linearity into the network, enabling it to learn complex patterns. Common choices include ReLU (Rectified Linear Unit) for hidden layers and softmax for the output layer of multi-class classification tasks.
- Weight Initialization: Proper initialization of weights is crucial to prevent the vanishing or exploding gradients problem. Techniques such as He initialization or Xavier initialization are often used.
- Learning Rate: The learning rate determines the size of the steps taken during weight updates. It may be fixed or adaptively changed using optimization techniques such as Adam or RMSprop.
- Backpropagation: This algorithm calculates the gradient of the loss function for each weight by the chain rule, efficiently propagating the error back through the network and updating the weights to minimize the loss.

3.4 K-nearest Neighbors (kNN)

Description:
K-Nearest Neighbors (kNN) is a straightforward, instance-based learning algorithm that classifies a new instance based on the majority class among its k-nearest neighbors in the feature space. It's a non-parametric method that is particularly useful for classification and regression tasks where the relationship between the feature variables and the output is complex or unknown.
Implementation Details:

- Distance Functions: The choice of distance function (e.g., Euclidean, Manhattan) plays a crucial role in determining the similarity between instances. Different functions may be better suited to different types of datasets.

- Choosing k: The number of neighbors, k, is a critical parameter. A small k makes the algorithm sensitive to noise, while a large k may smooth over important nuances in the data. The optimal k is usually determined through cross-validation.
- Weighted Voting: In weighted kNN, the vote of each neighbor is weighted by its distance from the query point, giving closer neighbors more influence on the classification.
- Efficiency Considerations: For large datasets, the basic kNN algorithm can be computationally expensive.

4. Platform Description

For this project, we utilized Python due to its extensive support for data analysis and machine learning, coupled with its readability and simplicity. We chose Python 3.12 for its latest features and optimal performance.

4.1 Programming Language

Python Version: 3.12

4.2 Libraries Used

Our implementation leveraged the following Python libraries for data manipulation, numerical computations, and evaluating the performance of our models:

- Pandas and NumPy for data manipulation and numerical calculations.
- Scikit-learn (specifically, its metrics module) for model evaluation metrics like accuracy_score and classification_report.
- A custom utility, fetch_ucirepo, facilitated direct access to datasets from the UCI machine learning repository.

4.3 Development Platform

Our development environment consisted of various platforms that supported both individual coding efforts and collaborative development:

- PyCharm for comprehensive code development and debugging.
- Jupyter Notebook for interactive coding sessions, data exploration, and visualization.
- Google Colab Pro for leveraging cloud-based resources, including free GPU access, which was particularly useful for training computationally intensive models.

This setup provided a versatile and efficient framework for implementing and evaluating the machine learning algorithms explored in this project.

5. Datasets Description

In this project, we leverage five distinct datasets from the UCI Machine Learning Repository for various classification tasks. Each dataset is unique in its features, target variable, and the problem it presents. This section of our work presents a summarised description of the various datasets used in this experiment. For each dataset, the attribute characteristics, number of class variables, number of attributes, number of instances, missing values and associated task is presented.

5.1 Breast Cancer Wisconsin (Diagnostic)

Description: This dataset comprises diagnostic information extracted from images of fine needle aspirates of breast masses. The data describes attributes of cell nuclei, aiding in the diagnosis of breast cancer.
Classes: Binary (Malignant, Benign).
Target Variable: Tumor type (Malignant or Benign).

Characteristic	Details
Data Set Characteristics	Multivariate
Attribute Characteristics	Real
Associated Tasks	Classification
Number of Classes	2
Number of Instances	569
Number of Attributes	32
Number of Missing Values	None

Type of Attributes: Continuous attributes describing characteristics of cell nuclei.
Attribute Information:
Ten real-valued features are computed for each cell nucleus:

1. radius (mean of distances from the center to points on the perimeter)
2. texture (standard deviation of gray-scale values)
3. perimeter
4. area
5. smoothness (local variation in radius lengths)
6. compactness (perimeter2ˆ area 1.0)
7. concavity (severity of concave portions of the contour)
8. concave points (number of concave portions of the contour)
9. symmetry
10. fractal dimension (“coastline approximation” - 1)

5.2 Car Evaluation

Description: The Car Evaluation dataset involves assessing the acceptability of cars based on several attributes related to cost, technical specifications, comfort, and safety.
Classes: Categorical (Unacceptable, Acceptable, Good, Very Good).
Target Variable: Car acceptability rating.

Characteristic	Details
Data Set Characteristics	Multivariate
Attribute Characteristics	Categorical
Associated Tasks	Classification
Number of Classes	4
Number of Instances	1728
Number of Attributes	6
Number of Missing Values	None

Type of Attributes: Categorical attributes related to car specifications and evaluations.
Attribute Information:

1. buying: vhigh, high, med, low.
2. maint: vhigh, high, med, low.
3. doors: 2, 3, 4, 5more.
4. persons: 2, 4, more.
5. lug boot: small, med, big.
6. safety: low, med, high.

5.3 Ecoli

Description: This dataset is concerned with the prediction of protein localization sites in Gram-negative bacteria, based on various sequence and composition attributes.
Classes: Categorical (Various localization sites).
Target Variable: Protein localization site.

Characteristic	Details
Data Set Characteristics	Multivariate
Attribute Characteristics	Real
Associated Tasks	Classification
Number of Classes	8
Number of Instances	336
Number of Attributes	8
Number of Missing Values	None

Type of Attributes: Continuous and categorical attributes related to protein properties.
Attribute Information:

1. Sequence Name: Accession number for the SWISS-PROT database.
2. mcg: McGeoch's method for signal sequence recognition.
3. gvh: von Heijne's method for signal sequence recognition.
4. lip: von Heijne's Signal Peptidase II consensus sequence score. Binary attribute.
5. chg: Presence of charge on N-terminus of predicted lipoproteins. Binary attribute.
6. aac: score of discriminant analysis of the amino acid content of outer membrane and periplasmic proteins.
7. alm1: score of the ALOM membrane spanning region prediction program.
8. alm2: score of ALOM program after excluding putative cleavable signal regions from the sequence.

5.4 Letter Recognition

Description: The Letter Recognition dataset is utilized for optical character recognition, where each instance is a set of attributes derived from an image of a letter.
Classes: Categorical (26 capital letters of the English alphabet).
Target Variable: Capital letter.

Characteristic	Details
Data Set Characteristics	Multivariate
Attribute Characteristics	Integer
Associated Tasks	Classification
Number of Classes	26
Number of Instances	20,000
Number of Attributes	16
Number of Missing Values	None

Type of Attributes: Continuous attributes derived from black-and-white rectangular pixel displays.
Attribute Information:

1. x-box : horizontal position of box (integer)
2. y-box : vertical position of box (integer)
3. width : width of box (integer)
4. high : height of box (integer)
5. onpix : total # on pixels (integer)

6. x-bar : mean x of on pixels in box (integer)
7. y-bar : mean y of on pixels in box (integer)
8. x2bar : mean x variance (integer)
9. y2bar : mean y variance (integer)
10. xybar : mean x y correlation (integer)
11. x2ybr:meanofx*x*y(integer)
12. xy2br:meanofx*y*y(integer)
13. x-ege : mean edge count left to right (integer)
14. xegvy : correlation of x-ege with y (integer)
15. y-ege : mean edge count bottom to top (integer)
16. yegvx : correlation of y-ege with x (integer)

5.5 Mushroom

Description: The Mushroom dataset comprises descriptions of hypothetical mushroom samples categorized by their edibility, which is an essential task for foraging and mushroom consumption safety.
Classes: Binary (Edible, Poisonous).
Target Variable: Mushroom edibility.

Characteristic	Details
Data Set Characteristics	Multivariate
Attribute Characteristics	Categorical
Associated Tasks	Classification
Number of Classes	2
Number of Instances	8124
Number of Attributes	22
Number of Missing Values	2480

Type of Attributes: Categorical attributes describing mushroom features.
Attribute Information:

1. cap-shape: b,c,x,f,k,s
2. cap-surface: f,g,y,s
3. cap-color: n,b,c,g,r,p,u,e,w,y
4. bruises?: t,f
5. odor: a,l,c,y,f,m,n,p,s
6. gill-attachment: a,d,f,n
7. gill-spacing: c,w,d
8. gill-size: b,
9. gill-color: k,n,b,h,g,r,o,p,u,e,w,yellow=y
10. stalk-shape: e,t
11. stalk-root: b,c,u,e,z,r,?
12. stalk-surface-above-ring: f,y,k,s
13. stalk-surface-below-ring: f,y,k,s
14. stalk-color-above-ring: n,b,c,g,o,p,e,w,y
15. stalk-color-below-ring: n,b,c,g,o,p,e,w,y
16. veil-type: p,u
17. veil-color: n,o,w,y
18. ring-number: n,o,t
19. ring-type: c,e,f,l,n,p,s,z
20. spore-print-color: n,b,h,r,o,u,w,y
21. population: a,c,n,s,v,y

22. habitat: g,l,m,p,u,w,d

6. Implementation Details

6.1 Pre-processing of Datasets

The pre-processing of datasets in our project is comprehensive and crucial for ensuring the data is suitable for feeding into machine learning models. Here are the key steps involved in the pre-processing pipeline:

- Missing Value Treatment: Depending on the dataset and the nature of missing values, we choose between dropping rows with missing values or imputing them. The SimpleImputer from sklearn.impute allows for various strategies such as mean, median, or mode imputation. For categorical data, missing values are imputed using the most frequent category (mode), which ensures minimal bias in the categorical distributions.
- Data Transformation: Categorical variables are transformed into numerical formats using encoding schemes. Depending on the analysis requirement, we use LabelEncoder for ordinal encoding, OneHotEncoder for nominal categories where no ordinal relationship exists, and OrdinalEncoder for categories with a clear ranking. This transformation is critical as most machine learning algorithms perform better or require input variables to be numeric.
- Data Scaling: Numerical features are scaled to ensure that no variable dominates others due to its scale. Using StandardScaler or MinMaxScaler, features are standardized to have zero mean and unit variance or scaled to lie between a given minimum and maximum value, respectively.
- Outlier Treatment: Outliers can skew the results by affecting the mean and standard deviation of the dataset. We employ methods like the Interquartile Range (IQR) method or Z-score methods to detect and treat outliers, thereby normalizing the data distribution and improving the model's robustness.

6.2 Parameter Setting and Hyperparameter Selection Process

Parameter tuning and hyperparameter selection are executed to optimize model performance. Here’s how these are approached:

- Model-Specific Parameters: For each machine learning model, relevant parameters such as the number of estimators in ensemble methods, the depth of trees, learning rates, or the number of neighbors in kNN are tuned. These parameters significantly influence the learning capacity and performance of the models.
- Cross-Validation: To ensure that our models generalize well to new data, we use 10 times 5-fold cross-validation techniques to evaluate the effectiveness of different hyperparameters. This method helps in identifying the best set of parameters that provide the most stable and accurate predictions across different subsets of the dataset.
- Grid Search and Random Search: These are among the methods used for hyperparameter optimization. Grid search evaluates a model with all combinations of hyperparameter values, providing a comprehensive exploration of the parameter space. Random search, on the other hand, samples parameter combinations randomly for a given number of iterations. It is less exhaustive but can be more efficient for larger datasets.

6.3 Performance Metrics:

- Breast Cancer Wisconsin (Diagnostic) Dataset: Accuracy is crucial in distinguishing between malignant and benign breast tumors, impacting treatment decisions and patient outcomes. High accuracy reduces the risk of unnecessary treatments and ensures timely intervention for malignant conditions, enhancing patient care in oncological settings.
- Car Evaluation Dataset: Accuracy in this dataset assesses the model’s ability to correctly classify cars based on acceptability. This influences manufacturing decisions and consumer choices by ensuring cars meet expected standards and preferences.

- Ecoli Dataset: In the Ecoli dataset, accuracy is vital for correctly predicting protein localization sites, which supports accurate scientific research and applications in biotechnology, impacting advancements in medicine and drug development.
- Letter Recognition Dataset: High accuracy is essential for correctly recognizing and classifying letters from image data, crucial for applications like automated postal sorting, where it ensures efficient mail processing and delivery.
- Mushroom Dataset: For the Mushroom dataset, high accuracy is critical to distinguish safely between edible and poisonous mushrooms, directly affecting consumer health and safety. It is especially important to minimize false negatives to prevent hazardous misclassifications.

7. Program Design

The program is structured in a modular fashion to ensure clarity, maintainability, and scalability. Below are the key components and their functionalities:

7.1 AdaBoost Implementation (adaboost.py)

TreeStump Class:

This class serves as the weak learner within the AdaBoost ensemble. It makes binary decisions by comparing a particular feature's value to a learned threshold.

- Threshold Learning: The fit method iteratively tests every feature at every unique value to determine the threshold that results in the lowest weighted classification error.
- Prediction: Once trained, the predict method applies this decision rule to classify new samples based on the learned threshold and feature.

AdaBoostMultiClass Class:

It orchestrates multiple TreeStumps to form a strong classifier for multi-class problems using the One vs. Rest strategy.

- Training: During the fitting process, it initializes weights, iterates through the creation of tree stumps, updates weights based on misclassification errors, and calculates the alpha (performance weight) for each stump.
- Multi-Class Strategy: For each class, it trains a binary classifier and during prediction, it aggregates the weighted votes to make a final decision.

7.2 Random Forest Implementation (random_forest.py)

Node Class:

Represents a single node in the decision tree, which could be a decision node with a feature index and threshold to split on, or a leaf node with a predicted value.

DecisionTree Class:

Constructs a decision tree with methods to build the tree using recursive splitting.

- Gini Index: A method for calculating impurity of a node, an essential part of the split decision.
- Information Gain: Determines the quality of a split based on reduction in impurity and is used to find the best threshold and feature to split on.
- Best Split: Searches for the best way to divide the data into two groups, which will then become child nodes.

RandomForest Class:

Combines multiple decision trees to reduce variance and improve generalization.

- Bootstrap Aggregation: Each tree is trained on a random subset of the data with replacement, known as bootstrapping.
- Feature Randomness: When building trees, each split considers a random subset of features to increase diversity among trees.

- Prediction Aggregation: For a given input, each tree makes a prediction and the class with the most votes across all trees is chosen.

7.3 Artificial Neural Network Implementation (ann.py)

ANN Class:

Simulates a simple feedforward neural network with a hidden layer.

- Weight Initialization: Uses He initialization strategy (scaled normal distribution) for weight matrices to ensure proper gradient flow at the start of training.
- Feedforward Propagation: Implements the forward pass of data through the network using matrix operations for efficiency.
- Activation Functions: Employs ReLU for hidden layer activations and softmax for the output layer to handle multi-class classification problems.
- Backpropagation: The core learning process, updating the weights and biases in the opposite direction of the gradient to minimize loss.

7.4 K-Nearest Neighbors Implementation (knn.py)

KNN Class:

A non-parametric instance-based learning algorithm that classifies samples based on the labels of their nearest neighbors.

- Distance Calculation: Allows for the use of different distance functions. The default is the Euclidean distance, but it can be swapped out for Manhattan distance or any other metric.
- 'k' Value Determination: 'k' can be set manually or determined automatically based on the cube root of the number of training samples. This is a heuristic that often yields good results without extensive tuning.
- Voting Mechanism: Uses a simple majority vote from the nearest 'k' neighbors to classify a new sample. In the case of ties, it would typically choose the class of the closest neighbor among the ties.

Each module is carefully designed to encapsulate the functionality of each algorithm and is called upon by the controllers.py script, which acts as the orchestrator for running the machine learning pipeline from preprocessing to evaluation. Parameters and constants are fetched from constants.py, ensuring a single source of truth for hyperparameters and configurations that are used across all modules. The modularity of this design allows for easy experimentation with different models and parameters as well as clear debugging and maintenance pathways.

7.5 Data Preprocessing Module

Responsible for data cleaning, handling missing values, encoding categorical variables, scaling, and outlier treatment. This module uses the preprocessing functions as described, utilizing scikit-learn and pandas for most of the heavy lifting.

7.6 Model Training Module

Contains the implementation of various machine learning algorithms. It integrates with the preprocessing module to receive cleaned and prepared data. The module leverages scikit-learn’s estimator API for consistency and uses different classes for different algorithms.

7.7 Hyperparameter Tuning and Model Selection Module

This module is tasked with optimizing model parameters through techniques such as grid search and random search. It interfaces with the model training module to evaluate different parameter configurations and select the best-performing models based on cross-validation scores.

7.8 Evaluation Module

Once models are trained and tuned, the evaluation module assesses their performance on a hold-out test set or through additional rounds of cross-validation. This module utilizes various metrics appropriate to the task (accuracy, F1 score, recall, etc.) and provides detailed reports and visualizations of model performance.

7.9 Utility and Helper Functions

A collection of utility functions that support data handling, logging, and system operations such as file I/O, logging configurations, and error handling. These functions ensure that the system runs smoothly and that users can easily track and understand the process flow.

7.10 Data Structures Used

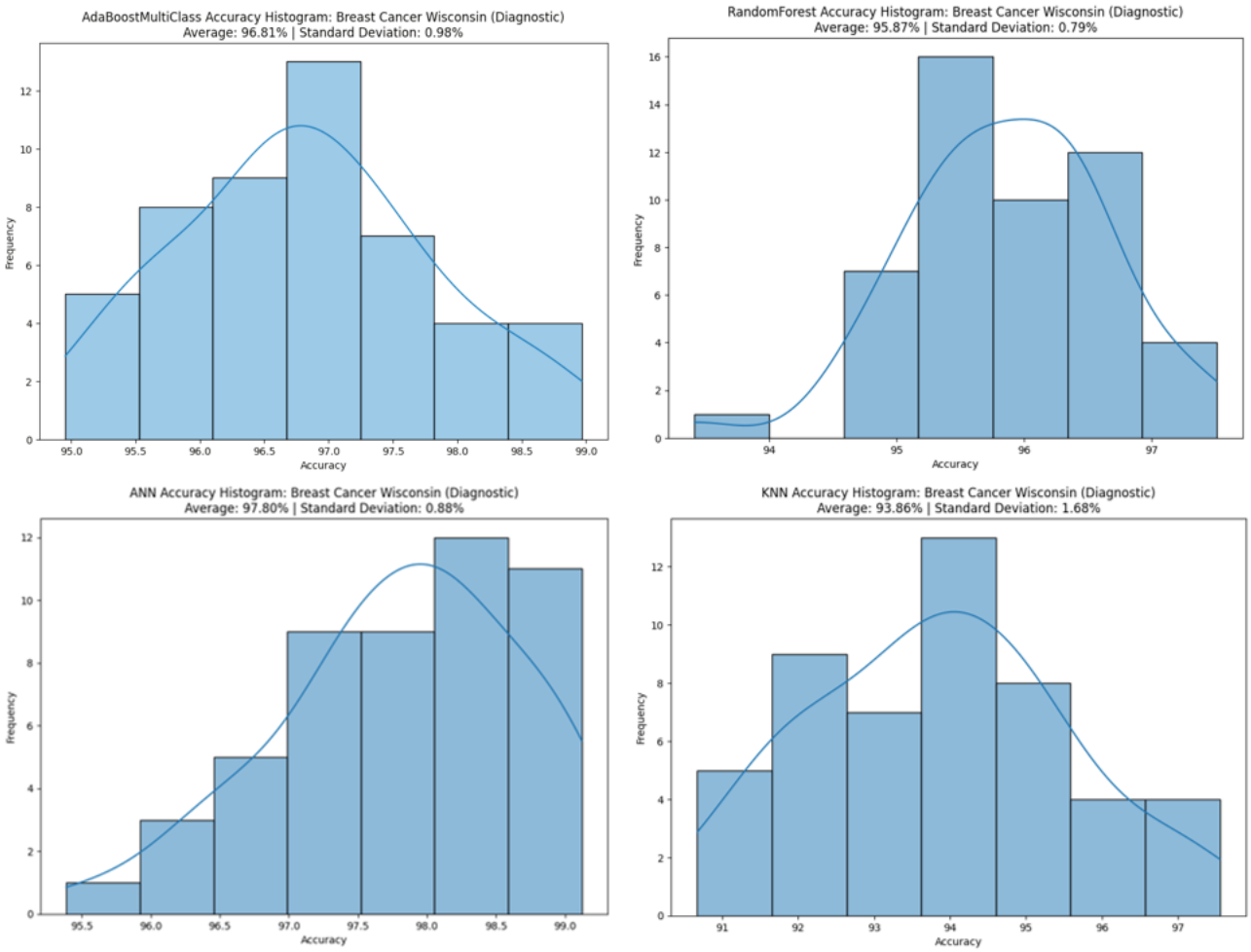
The choice of data structures is critical to the efficiency and scalability of our machine learning application. Here's an overview of the data structures employed:

- DataFrames (pandas): Pandas DataFrames are extensively used for data manipulation and analysis. They provide a two-dimensional, size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). DataFrames are ideal for handling structured data, allowing for sophisticated indexing, merging, and time series functionalities which are essential for preprocessing and analysis steps.
- Numpy Arrays: Numpy arrays provide a powerful N-dimensional array structure that offers fast array-oriented arithmetic operations and flexible broadcasting capabilities. These arrays are used for numerical operations, especially in the transformation and scaling phases of preprocessing, where performance is critical.
- Dictionaries: Used for mapping and encoding categorical data where keys represent categories and values represent numerical equivalents. Dictionaries are beneficial for quick lookups and mappings, which are crucial during the data encoding and feature engineering phases.
- Lists: Lists are used for more flexible operations where sequences of data are needed temporarily, especially when iterating over parameters, storing results for comparison, or handling data that do not require the structure of arrays or DataFrames.

8. Experimental Results and Analysis

This section presents the findings from the experimental evaluation of four machine learning algorithms: AdaBoost, Random Forest, Artificial Neural Network (ANN), and K-Nearest Neighbors (KNN). Each algorithm was applied to five distinct datasets: Breast Cancer Wisconsin (Diagnostic), Car Evaluation, Ecoli, Letter Recognition, and Mushroom. The primary performance metric used in this evaluation is accuracy, assessed through 10 times 5-fold cross-validation to ensure the robustness of the results.

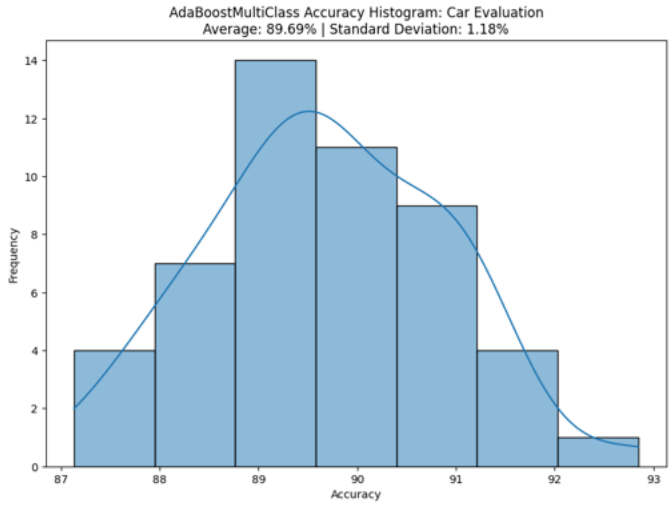
8.1 Breast Cancer Wisconsin (Diagnostic)

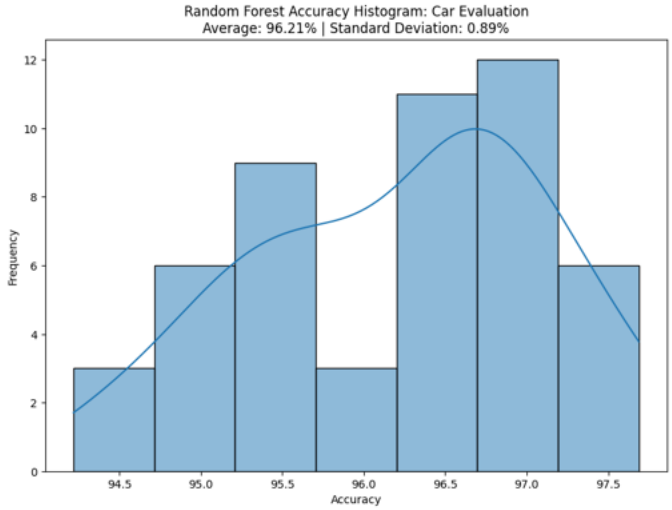


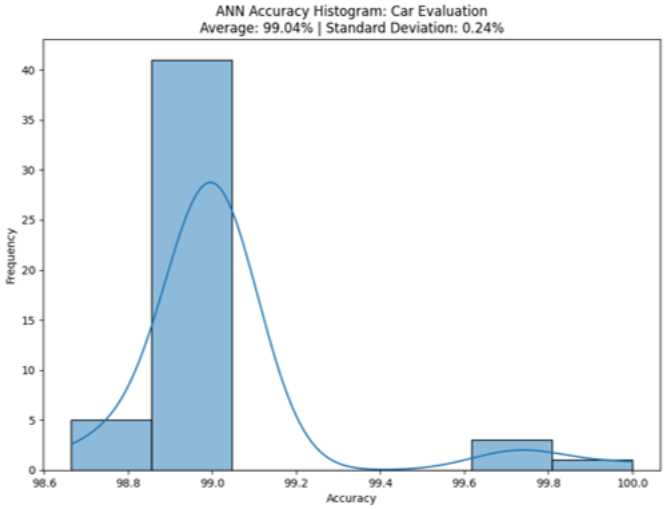
- AdaBoost: With an average accuracy of 96.81% and a standard deviation of 0.98%, AdaBoost demonstrated high accuracy with relatively low variance, indicating reliable performance across different data folds.
- Random Forest: This algorithm showed an average accuracy of 95.87% with a standard deviation of 0.79%, suggesting consistent performance, albeit with slightly less accuracy compared to AdaBoost.
- ANN (Artificial Neural Network): The ANN yielded the highest average accuracy of 97.80% with a standard deviation of 0.88%, outperforming the other algorithms in terms of accuracy and showing comparably low variability.
- KNN (k-Nearest Neighbors): With the lowest average accuracy of 93.86% and the highest standard deviation of 1.68%, the KNN algorithm was the least consistent and had the lowest performance among the evaluated models.

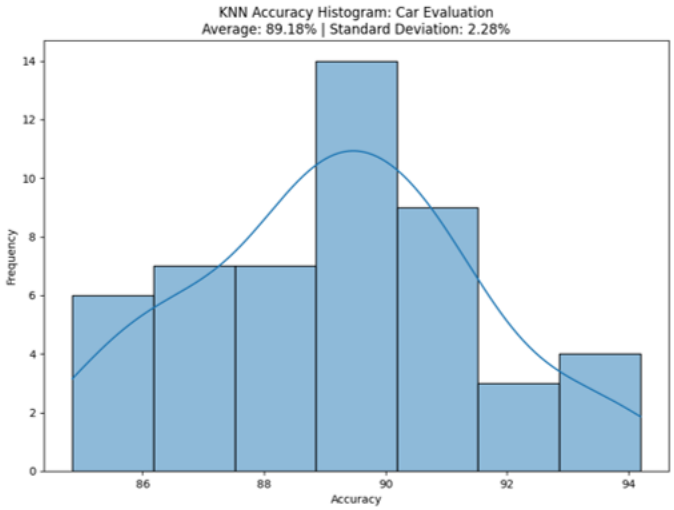
The **ANN algorithm** performed the best in terms of both average accuracy and consistent results across the 10 times 5-fold cross-validation. This suggests that for the Breast Cancer Wisconsin (Diagnostic) dataset, ANN was the most effective model among those tested.

8.2 Car Evaluation





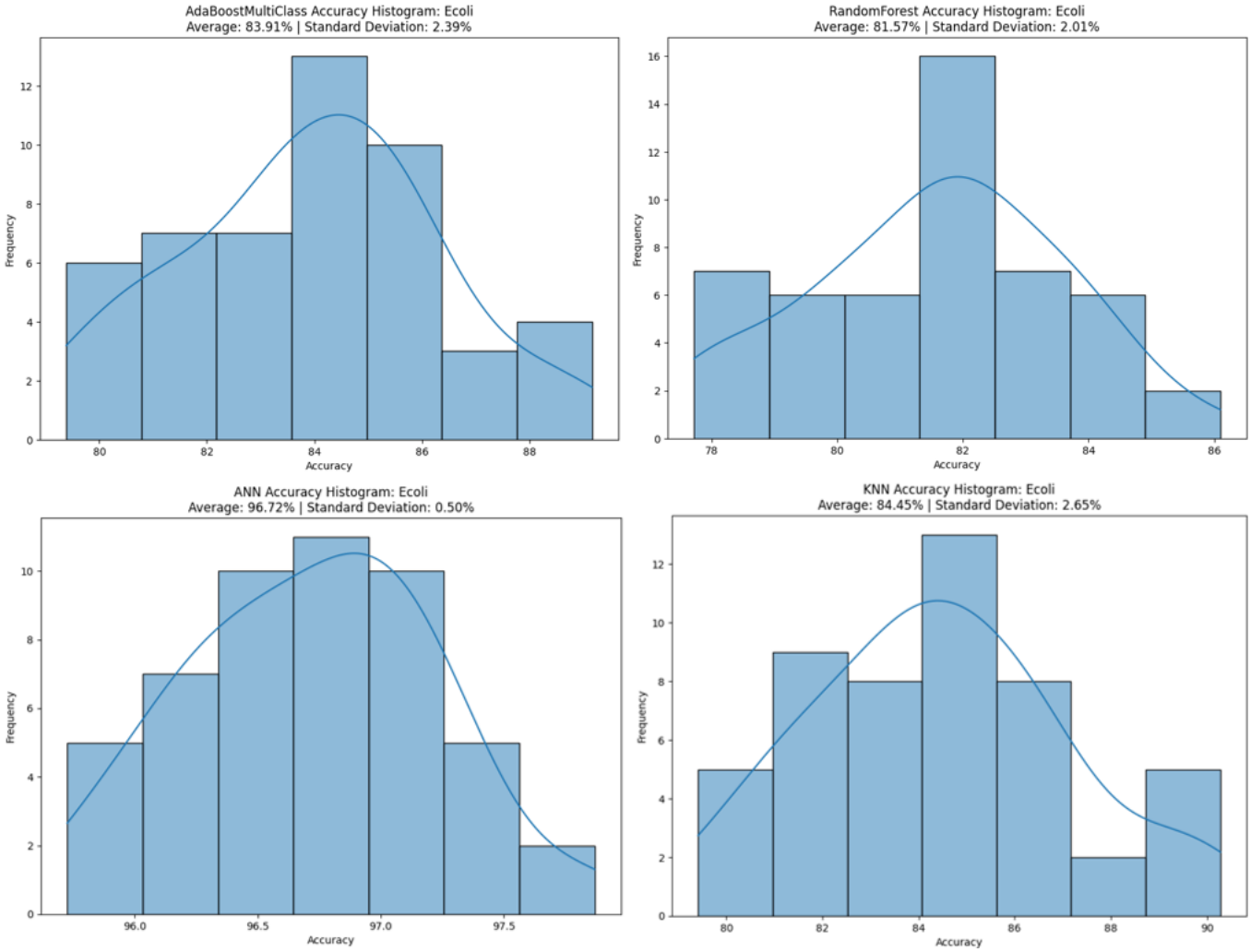




- AdaBoost: The average accuracy here is 89.69% with a standard deviation of 1.18%, showing that the model has moderate accuracy with some variability in performance.
- Random Forest: This model has an average accuracy of 96.21% with a standard deviation of 0.89%, indicating higher accuracy and consistent performance.
- ANN (Artificial Neural Network): The ANN significantly outperforms the other models with an average accuracy of 99.04% and a very low standard deviation of 0.24%, suggesting both high accuracy and reliability.
- KNN (k-Nearest Neighbors): KNN has an average accuracy of 89.18% with a standard deviation of 2.28%, which is the highest variability and one of the lower averages in this group.

Based on these histograms, the **ANN algorithm** performed best for the Car Evaluation dataset, achieving the highest average accuracy with the least variability among the tested models.

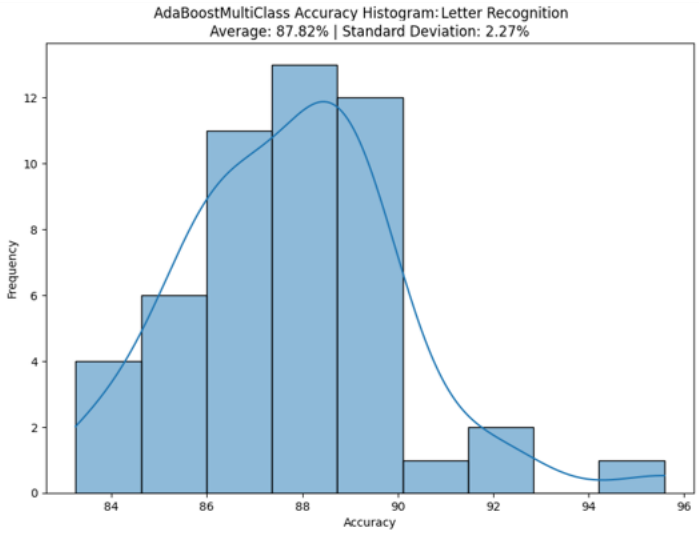
8.3 Ecoli

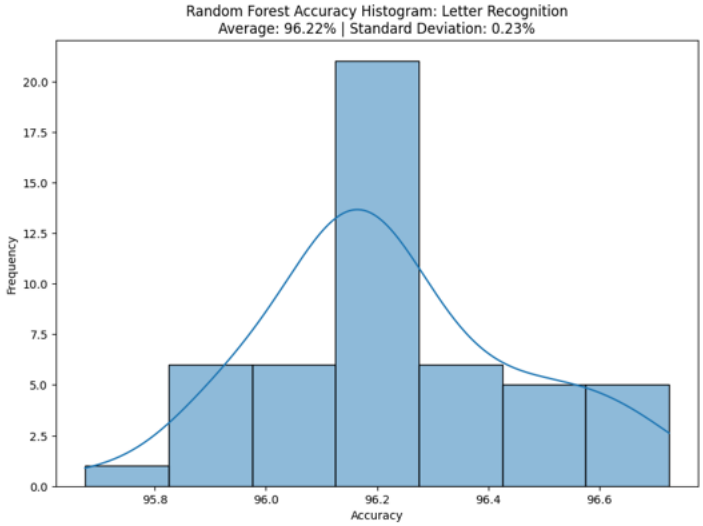


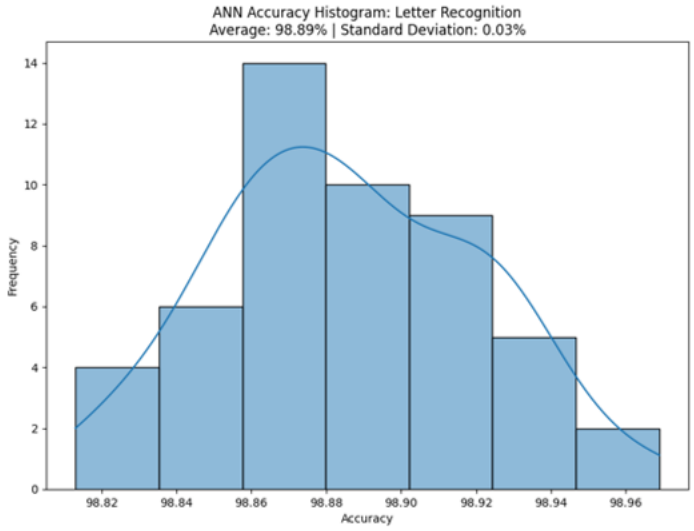
- AdaBoost: The histogram shows an average accuracy of 83.91% with a standard deviation of 2.39%, indicating moderate average accuracy with some variability.
- Random Forest: The average accuracy is 81.57%, and the standard deviation is 2.01%, which is the lowest average accuracy among the four algorithms and also shows a fair amount of variability.
- ANN (Artificial Neural Network): The ANN outperforms other algorithms with an average accuracy of 96.72% and a low standard deviation of 0.50%, suggesting very high accuracy and consistent performance.
- KNN (k-Nearest Neighbors): It has an average accuracy of 84.45% and a standard deviation of 2.65%, indicating reasonable accuracy but with noticeable variability in its performance.

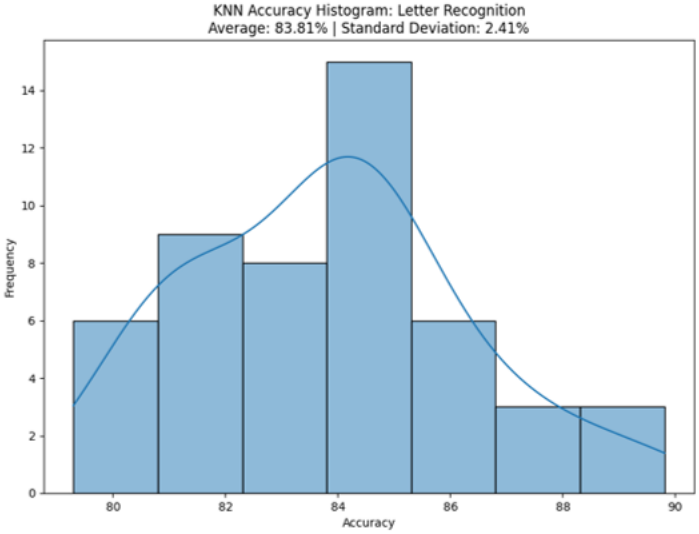
Considering these results, the **ANN algorithm** demonstrated the best performance on the Ecoli dataset, with the highest average accuracy and lowest variability in the accuracy distribution.

8.4 Letter Recognition





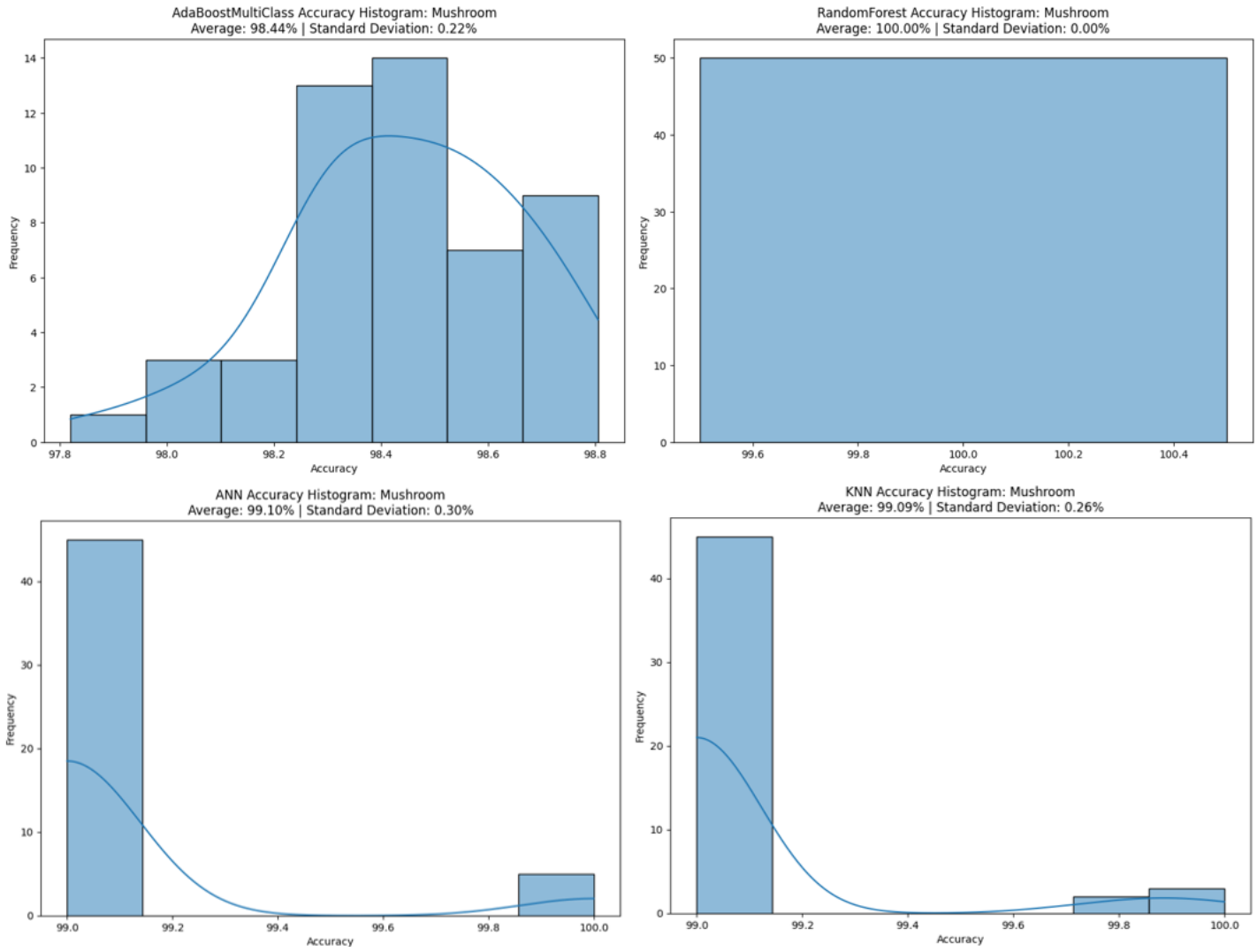




- AdaBoost: The average accuracy is 87.82%, with a standard deviation of 2.27%. This shows relatively high accuracy with some variability in the results.
- Random Forest: With an average accuracy of 96.22% and a standard deviation of 0.23%, RandomForest exhibits high accuracy and very consistent performance.
- ANN (Artificial Neural Network): ANN stands out with the highest average accuracy of 98.89% and an extremely low standard deviation of 0.03%, indicating both very high accuracy and very reliable results.
- KNN (k-Nearest Neighbors): KNN shows an average accuracy of 83.81% and a standard deviation of 2.41%, which is the lowest average accuracy and one of the higher variabilities among the algorithms tested.

Given these distributions, the **ANN algorithm** performed the best on the Letter Recognition dataset, achieving the highest average accuracy with remarkably low variability.

8.5 Mushroom



- AdaBoost: Exhibits an average accuracy of 98.44% with a low standard deviation of 0.22%, indicating high accuracy and consistent performance.
- Random Forest: Stands out with perfect average accuracy of 100.00% and a standard deviation of 0.00%, suggesting that it classified all instances correctly in every run.
- ANN (Artificial Neural Network): Achieves an average accuracy of 99.10% with a standard deviation of 0.30%, which is very high accuracy with minimal variation.
- KNN (k-Nearest Neighbors): Shows an average accuracy of 99.09% with a standard deviation of 0.26%, nearly matching the ANN in terms of average accuracy and consistency.

For the Mushroom dataset, the **Random Forest** algorithm performed the best with consistently perfect accuracy across all validations.

9. Conclusion

This study aimed to evaluate and compare the performance of four distinct machine learning algorithms across five diverse datasets from the UCI repository. The selected algorithms include AdaBoost with Tree Stumps, Random Forest, Artificial Neural Networks (ANN) with Backpropagation, and K Nearest Neighbor (kNN). The datasets span various domains such as medical, automotive, and microbiology, specifically the Breast Cancer Wisconsin, Car Evaluation, Letter Recognition, Mushroom, and Ecoli datasets. The primary focus was on assessing classification accuracy for each algorithm.

We found that ANN exhibited strong accuracy across diverse datasets. For instance, it excelled notably on the Letter Recognition dataset, achieving an average accuracy of 98.89% with minimal variance. Random Forest performed exceptionally well with the Mushroom dataset, consistently achieving a perfect score, indicating its suitability for this specific data type. AdaBoost demonstrated commendable performance with the Ecoli dataset, boasting an average accuracy of 83.91%, showcasing its efficacy in handling biological data.

We encountered the following challenges during the implementation:

1. Increased Computational Demand:
    - Issue: Implementing repeated stratified k-fold cross-validation significantly increases computational requirements due to multiple iterations and fits required across various subsets of data.
    - Impact: This approach strained system resources, leading to extended computational times and potentially increased costs.
    - Mitigation Strategy: Employed parallel processing to distribute the computation across all available CPU cores and utilized efficient coding practices to minimize unnecessary computations.
2. Extensive Hyperparameter Search:
    - Issue: Hyperparameter tuning via grid search involved exploring a vast array of parameters, especially for complex models like ANN, leading to large-scale computations.
    - Impact: The exhaustive nature of grid search exponentially increased the total computation time and resource utilization.
    - Mitigation Strategy:
        - Parallel Grid Search: Used parallel processing capabilities of GridSearchCV to perform hyperparameter tuning concurrently across different parameter combinations.
        - Search Space Optimization: Refined the search space based on preliminary results and expert knowledge to reduce the number of parameter combinations tested.

Recommendations for Future Work:

- Optimize Computational Resources: Refine parallel processing setups and explore more sophisticated distributed computing solutions.
- Explore Efficient Hyperparameter Tuning Techniques: Implement more efficient techniques such as randomized search or Bayesian optimization that could potentially reduce computational overhead while still effectively exploring the parameter space.
- Cross-Dataset Learning: Investigate learning strategies that leverage patterns across datasets to improve the generalizability and performance of models.

This study sets the foundation for choosing the right algorithms in machine learning projects and suggests ways to make these models more advanced and useful in making decisions based on data.

10. Appendix

adaboost.py

```
"""
Implementation of AdaBoost with Tree Stumps from scratch.

Author
------
@priyankabhamare, priyanka.bhamare@unb.ca
"""
import numpy as np
from sklearn.metrics import accuracy_score
import constants as CONSTANTS

# Create a Tree Stump class to represent a decision stump in AdaBoost algorithm (Decision Tree with
max_depth=1)
class TreeStump:
    def __init__(self):
        self.threshold = None
        self.feature_index = None
        self.decision_rule = None  # Store the decision rule for the leaf nodes

    def predict(self, X):
        # Make predictions using the trained tree stump.
        X_column = X[:, self.feature_index]  # Get the feature column
        predictions = np.where(X_column < self.threshold, self.decision_rule['left'], self.decision_rule['right'])
        return predictions

    def fit(self, X, y, weights):
        # Fit the tree stump to the binary labeled data.
        n_samples, n_features = X.shape
        best_err = float('inf')  # Initialize the best error as positive infinity

        # Determine unique classes (binary classification assumed)
        unique_classes = np.unique(y)
        if len(unique_classes) != 2:
            raise ValueError("TreeStump supports binary classification. Ensure y contains two unique classes.")

        # Iterate over all features and possible thresholds to find the best rule and split point that minimizes error
        for feature_index in range(n_features):
            X_column = X[:, feature_index]
            thresholds = np.unique(X_column)

            # Iterate over all unique values in the feature column
            for threshold in thresholds:
                for left_class in unique_classes:
                    right_class = unique_classes[unique_classes != left_class][0]  # Pick the other class
                    predictions = np.where(X_column < threshold, left_class, right_class)
                    y = y.reshape(-1)
                    misclassified = predictions != y
                    err = np.sum(weights[misclassified]) / np.sum(weights)
```

```
                if err < best_err:
                    best_err = err
                    self.threshold = threshold
                    self.feature_index = feature_index
                    self.decision_rule = {'left': left_class, 'right': right_class}

# Create a custom AdaBoost classifier for multi-class classification using Tree Stumps as weak learners
class AdaBoostMultiClass:
    def __init__(self, n_estimators=CONSTANTS.ADABOOST_N_ESTIMATORS,
learning_rate=CONSTANTS.ADABOOST_LEARNING_RATE):
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.estimators = []
        self.alphas = []
        self.classes_ = None
        self.is_one_hot_encoded = False

    # Check if the target is one-hot encoded. Assumes y is a NumPy array.
    def _check_one_hot_encoded(self, y):
        if y.ndim == 1 or (y.ndim == 2 and y.shape[1] == 1):
            return False  # Label encoded
        return True  # One-hot encoded

    # Train the AdaBoost classifier for multi-class classification
    def fit(self, X, y):
        # Determine if y is one-hot encoded and get unique classes
        one_hot_encoded = self._check_one_hot_encoded(y)
        if one_hot_encoded:
            self.classes_ = np.arange(y.shape[1])
            self.is_one_hot_encoded = True
        else:
            self.classes_ = np.unique(y)
            self.is_one_hot_encoded = False
        n_samples, _ = X.shape

        # Train a classifier for each class
        for cls_idx, cls in enumerate(self.classes_):
            W = np.ones(n_samples) / n_samples  # Initialize weights uniformly
            estimators_cls = []
            alphas_cls = []

            # Train n_estimators number of weak classifiers (stumps)
            for _ in range(self.n_estimators):
                stump = TreeStump()  # Initialize a new decision stump
                # Prepare y_binary for the current class vs. rest
                if one_hot_encoded:
                    y_binary = y[:, cls_idx] * 2 - 1  # Convert from [0, 1] to [-1, 1]
                else:
                    y_binary = np.where(y == cls, 1, -1)
                stump.fit(X, y_binary, W)
                predictions = stump.predict(X)
```

```
            # Reshape y_binary if it is not one-hot encoded
            if not one_hot_encoded:
                y_binary = y_binary.reshape(-1)

            # Calculate errors and update weights
            incorrect = predictions != y_binary
            weighted_error = np.dot(W, incorrect) / np.sum(W)

            # Alpha calculation with smoothing to avoid division by zero
            alpha = self.learning_rate * 0.5 * np.log((1.0 - weighted_error) / (max(weighted_error, 1e-10)))

            # Update weights and normalize
            W *= np.exp(-alpha * y_binary * predictions)
            W /= np.sum(W)  # Normalize weights

            estimators_cls.append(stump)
            alphas_cls.append(alpha)

        # Store the estimators and alphas for the current class
        self.estimators.append(estimators_cls)
        self.alphas.append(alphas_cls)

# Make predictions using the trained AdaBoost classifier for multi-class classification
def predict(self, X):
    class_votes = np.zeros((X.shape[0], len(self.classes_)))

    for cls_idx, cls in enumerate(self.classes_):
        for estimator, alpha in zip(self.estimators[cls_idx], self.alphas[cls_idx]):
            predictions = estimator.predict(X)
            class_votes[:, cls_idx] += alpha * predictions

    if self.is_one_hot_encoded:
        # Return one-hot encoded predictions
        return (class_votes == class_votes.max(axis=1)[:, None]).astype(int)
    else:
        return self.classes_[np.argmax(class_votes, axis=1)]

def set_params(self, **parameters):
    for parameter, value in parameters.items():
        setattr(self, parameter, value)
    return self

def get_params(self, deep=True):
    return {
        "n_estimators": self.n_estimators,
        "learning_rate": self.learning_rate,
    }

def score(self, X, y):
    y_pred = self.predict(X)
    return accuracy_score(y, y_pred)
```

random_forest.py

"""
Implementation of Random Forest module. This module is responsible for implementing the Random Forest algorithm from scratch.

Author

@rkalai, rishabh.kalai@unb.ca
"""

```
# Suppress All Warnings
import warnings

warnings.filterwarnings("ignore")
# Import the necessary libraries
# 3rd party libraries
import numpy as np
import random
from sklearn.metrics import accuracy_score
from math import sqrt
# Custom libraries
import utilities as UTILITIES
import constants as CONSTANTS

class Node:
    def __init__(
            self, feature_index=None, threshold=None, left=None, right=None, value=None
    ):
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left  # Left subtree
        self.right = right  # Right subtree
        self.value = value  # Value at leaf node

class DecisionTree:
    def __init__(self, max_depth=None, max_features=None):
        self.root = None
        self.max_depth = max_depth
        self.max_features = max_features

    def fit(self, X, y):
        self.root = self._build_tree(X, y)

    def _gini_index(self, y):
        # Calculate the Gini Index for a node
        m = len(y)
        return 1.0 - sum((np.sum(y == c) / m) ** 2 for c in np.unique(y))

    def _information_gain(self, parent, l_child, r_child):
        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
```

```
        gain = self._gini_index(parent) - (
            weight_l * self._gini_index(l_child) + weight_r * self._gini_index(r_child)
        )
        return gain

    def _get_best_split(self, X, y, num_features):
        best_split = {}
        max_gain = -float("inf")
        features = random.sample(range(num_features), self.max_features)
        for feature_index in features:
            feature_values = X[:, feature_index]
            thresholds = np.unique(feature_values)
            for threshold in thresholds:
                left_indices = np.where(feature_values <= threshold)
                right_indices = np.where(feature_values > threshold)
                gain = self._information_gain(y, y[left_indices], y[right_indices])
                if gain > max_gain:
                    best_split["feature_index"] = feature_index
                    best_split["threshold"] = threshold
                    best_split["dataset_left"] = left_indices
                    best_split["dataset_right"] = right_indices
                    best_split["gain"] = gain
                    max_gain = gain
        return best_split

    def _build_tree(self, X, y, depth=0):
        num_samples, num_features = X.shape
        # Criteria to stop splitting
        if num_samples >= 2 and depth <= self.max_depth:
            best_split = self._get_best_split(X, y, num_features)
            if best_split["gain"] > 0:
                left_subtree = self._build_tree(
                    X[best_split["dataset_left"][0]],
                    y[best_split["dataset_left"][0]],
                    depth + 1,
                )
                right_subtree = self._build_tree(
                    X[best_split["dataset_right"][0]],
                    y[best_split["dataset_right"][0]],
                    depth + 1,
                )
                return Node(
                    best_split["feature_index"],
                    best_split["threshold"],
                    left_subtree,
                    right_subtree,
                )
        leaf_value = self._calculate_leaf_value(y)
        return Node(value=leaf_value)

    def _calculate_leaf_value(self, y):
        # Calculate the most common target value in the segment
```

```
        leaf_value = max(y, key=list(y).count)
        return leaf_value

    def _traverse_tree(self, X, node):
        if node.value is not None:
            return node.value
        if X[node.feature_index] <= node.threshold:
            return self._traverse_tree(X, node.left)
        else:
            return self._traverse_tree(X, node.right)

    def predict(self, X):
        # Predict function to traverse the tree for each sample and return prediction
        return np.array([self._traverse_tree(x, self.root) for x in X])

class RandomForest:
    def __init__(self, n_estimators: int = CONSTANTS.RF_N_ESTIMATORS, max_depth: int =
CONSTANTS.RF_MAX_DEPTH,
                 max_features: int = None):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.max_features = max_features
        self.trees = []
    def fit(self, X, y):
        num_features = X.shape[1]
        # Set max_features to sqrt(num_features) if not specified, this is because it is a common rule of thumb to
use sqrt(num_features) for classification tasks
        self.max_features = (
            int(sqrt(num_features)) if not self.max_features else self.max_features
        )
        # Initialize the trees in the Random Forest
        self.trees = [
            DecisionTree(max_depth=self.max_depth, max_features=self.max_features)
            for _ in range(self.n_estimators)
        ]
        for tree in self.trees:
            X_sample, y_sample = UTILITIES.bootstrap_samples(X, y)
            tree.fit(X_sample, y_sample)

    def predict(self, X):
        tree_predictions = np.array([tree.predict(X) for tree in self.trees])
        # Return the mode of the predictions
        return np.array(
            [
                np.bincount(tree_predictions[:, i]).argmax()
                for i in range(tree_predictions.shape[1])
            ]
        )

    def get_params(self, deep=True):
        # deep=True is set by default, as the Random Forest does not have any Sub-Models
        # It has no effect on the function
```

```
    return {
        "n_estimators": self.n_estimators,
        "max_depth": self.max_depth,
        "max_features": self.max_features,
    }

def set_params(self, **parameters):
    for parameter, value in parameters.items():
        setattr(self, parameter, value)
    return self

def score(self, X, y):
    y_pred = self.predict(X)
    return accuracy_score(y, y_pred)
```

ann.py

```
"""
Implementation of ANN (Artificial Neural Network) model. The model is a Multi-Layer Perceptron (MLP) model, and is
implemented from scratch.

Author
------
@rkalai, rishabh.kalai@unb.ca
"""
# 3rd party libraries
import numpy as np
from sklearn.metrics import accuracy_score

# Import the necessary libraries
# Custom libraries
import constants as CONSTANTS
from utilities import softmax, cross_entropy_loss, derivative_cross_entropy_softmax

# GLOBAL SETTINGS
# Set Seed for reproducibility
np.random.seed(CONSTANTS.RANDOM_STATE)

class ANN:
    def __init__(
            self,
            input_size: int = None,
            hidden_size: int = CONSTANTS.HIDDEN_SIZE,
            output_size: int = None,
            learning_rate: float = CONSTANTS.LEARNING_RATE,
            epochs: int = CONSTANTS.EPOCHS,
    ):
        self.weights_input_hidden = np.random.randn(input_size, hidden_size) * np.sqrt(
            2.0 / input_size
        )
        self.biases_input_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.randn(
            hidden_size, output_size
        ) * np.sqrt(2.0 / hidden_size)
        self.biases_hidden_output = np.zeros((1, output_size))
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

    def feedforward(self, X):
        self.hidden_layer_output = (
                np.dot(X, self.weights_input_hidden) + self.biases_input_hidden
        )
        # ReLU Activation Function
```

```
        self.hidden_layer_output = np.maximum(
            0, self.hidden_layer_output
        )
        self.output_layer_output = (
                np.dot(self.hidden_layer_output, self.weights_hidden_output)
                + self.biases_hidden_output
        )
        self.output_layer_output = softmax(self.output_layer_output)
        return self.output_layer_output

    def backpropagation(self, X, y):
        y_pred = self.feedforward(X)
        error = derivative_cross_entropy_softmax(y_pred, y)
        # Backpropagation: Update the weights and biases
        weights_hidden_output_update = (
                np.dot(self.hidden_layer_output.T, error) / X.shape[0]
        )
        biases_hidden_output_update = np.sum(error, axis=0, keepdims=True) / X.shape[0]
        # Backpropagation: Calculate the error in the hidden layer
        error_hidden_layer = np.dot(error, self.weights_hidden_output.T)
        error_hidden_layer[self.hidden_layer_output <= 0] = 0
        # Backpropagation: Update the weights and biases
        weights_input_hidden_update = np.dot(X.T, error_hidden_layer) / X.shape[0]
        biases_input_hidden_update = (
                np.sum(error_hidden_layer, axis=0, keepdims=True) / X.shape[0]
        )
        # Update the weights and biases, based on the learning rate
        self.weights_hidden_output -= self.learning_rate * weights_hidden_output_update
        self.biases_hidden_output -= self.learning_rate * biases_hidden_output_update
        self.weights_input_hidden -= self.learning_rate * weights_input_hidden_update
        self.biases_input_hidden -= self.learning_rate * biases_input_hidden_update

    def train(self, X, y, epochs):
        for epoch in range(epochs):
            self.backpropagation(X, y)

    def fit(self, X, y=None):
        self.train(X, y, self.epochs)
        return self

    def predict(self, X):
        y_pred = self.feedforward(X)
        return (y_pred > 0.5).astype(int)

    def score(self, X, y):
        y_pred = self.predict(X)
        return accuracy_score(y, y_pred)

    def get_params(self, deep=True):
        return {
            "input_size": self.input_size,
            "hidden_size": self.hidden_size,
```

```
        "output_size": self.output_size,
        "learning_rate": self.learning_rate,
        "epochs": self.epochs,
    }

def set_params(self, **parameters):
    for parameter, value in parameters.items():
        setattr(self, parameter, value)
    return self
```

knn.py

```
"""
Implementation of K-Nearest Neighbors (KNN) algorithm from scratch.

Author
------
@schettiar, sadhana.chettiar@unb.ca
"""
from collections import Counter
import math

def euclidean_distance(point1, point2):
    distance = 0.0
    for i in range(len(point1)):
        distance += (point1[i] - point2[i]) ** 2
    return math.sqrt(distance)

def manhattan_distance(point1, point2):
    distance = 0.0
    for i in range(len(point1)):
        distance += abs(point1[i] - point2[i])
    return distance

class KNN:
    def __init__(self, k='auto', distance_fn=euclidean_distance):
        self.k = k
        self.distance_fn = distance_fn
        self.X_train = None
        self.y_train = None

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train
        # Determine k as the cube root of the number of training samples
        if self.k == 'auto':
            self.k = int(len(X_train) ** (1 / 3))
        return self  # Return self to comply with sklearn's fit method requirements

    def predict(self, X_test):
        predictions = []
        for x in X_test:
            distances = [(self.distance_fn(x, x_train), y) for x_train, y in zip(self.X_train, self.y_train)]
            distances.sort(key=lambda x: x[0])
            k_nearest_neighbors = distances[:self.k]
            k_nearest_labels = [neighbor[1] for neighbor in k_nearest_neighbors]
            most_common_label = Counter(k_nearest_labels).most_common(1)[0][0]
            predictions.append(most_common_label)
        return predictions

    def get_params(self, deep=True):
        return {'k': self.k, 'distance_fn': self.distance_fn}
```

```
def set_params(self, **parameters):
    for parameter, value in parameters.items():
        setattr(self, parameter, value)
    return self
```

constants.py

```
"""
File Path Configurations, Constants, and Global Variables for the Project
"""

import os
import numpy as np

# Project Root Directory
ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
# Model Directory
MODEL_DIR = os.path.join(ROOT_DIR, "models")

# Random State Seed
RANDOM_STATE = 42

# Dataset ID Constants
BREAST_CANCER_ID = 17
CAR_EVALUATION_ID = 19
ECOLI_ID = 39
LETTER_RECOGNITION_ID = 59
MUSHROOM_ID = 73

# Configuration Settings
# Data Preprocessing
DROP_MISSING_VALUES = True
IMPUTE_MISSING_VALUES_METHOD = "mean"
ONE_HOT_ENCODING = "one-hot"
ORDINAL_ENCODING = "ordinal"
SCALE_DATA = False
OUTLIER_TREATMENT_METHOD = None
# Train-Test Split
TEST_SIZE = 0.2
TRAIN_SIZE = 1 - TEST_SIZE
# Cross-Validation Settings
N_SPLITS = 5
N_REPEATS = 10

# Model Evaluation Metrics
# Artificial Neural Network
# ANN Default Hyperparameters
HIDDEN_SIZE = 100
LEARNING_RATE = 0.01
EPOCHS = 10000
# ANN Hyperparameters
ANN_HYPERPARAMETER_GRID = {
    "hidden_size": [50, 100, 150, 200],
    "learning_rate": [0.01, 0.1, 0.5],
    "epochs": [10000, 25000, 50000],
}
# Best Hyperparameters for ANN that were found using Grid Search
```

```
ANN_HYPERPARAMETERS = {
    ECOLI_ID: {
        'epochs': 10000,
        'hidden_size': 150,
        'learning_rate': 0.1
    },
    BREAST_CANCER_ID: {
        'epochs': 50000,
        'hidden_size': 50,
        'learning_rate': 0.5
    },
    LETTER_RECOGNITION_ID: {
        'epochs': 1500,
        'hidden_size': 150,
        'learning_rate': 0.5
    },
    MUSHROOM_ID: {
        'epochs': 25000,
        'hidden_size': 150,
        'learning_rate': 0.1
    },
    CAR_EVALUATION_ID: {
        'epochs': 50000,
        'hidden_size': 100,
        'learning_rate': 0.1
    }
}
# Random Forest
# Random Forest Default Hyperparameters
RF_N_ESTIMATORS = 50
RF_MAX_DEPTH = 10
# Random Forest Hyperparameters
RANDOM_FOREST_HYPERPARAMETER_GRID = {
    'n_estimators': np.linspace(10, 60, 5).astype(int),
    'max_depth': np.linspace(5, 60, 5).astype(int)
}
# Best Hyperparameters for Random Forest that were found using Grid Search
RANDOM_FOREST_HYPERPARAMETERS = {
    CAR_EVALUATION_ID: {
        'max_depth': 40,
        'n_estimators': 20
    },
    ECOLI_ID: {
        'max_depth': 46,
        'n_estimators': 35
    },
    LETTER_RECOGNITION_ID: {
        'max_depth': 10,
        'n_estimators': 20
    },
    MUSHROOM_ID: {
        'max_depth': 10,
```

```
        'n_estimators': 15
    },
    BREAST_CANCER_ID: {
        'max_depth': 10,
        'n_estimators': 15
    }
}

# AdaBoost
# AdaBoost Default Hyperparameters
ADABOOST_N_ESTIMATORS = 100
ADABOOST_LEARNING_RATE = 0.1
# AdaBoost Hyperparameters
ADABOOST_HYPERPARAMETER_GRID = {
    'n_estimators': np.linspace(100, 5100, 200).astype(int),
    'learning_rate': [0.1]
}
# Best Hyperparameters for AdaBoost that were found using Grid Search
ADABOOST_HYPERPARAMETERS = {
    BREAST_CANCER_ID: {
        'learning_rate': 0.1,
        'n_estimators': 100
    },
    CAR_EVALUATION_ID: {
        'learning_rate': 0.1,
        'n_estimators': 2500
    },
    ECOLI_ID: {
        'learning_rate': 0.1,
        'n_estimators': 300
    },
    LETTER_RECOGNITION_ID: {
        'learning_rate': 5,
        'n_estimators': 4000
    },
    MUSHROOM_ID: {
        'learning_rate': 0.1,
        'n_estimators': 300
    }
}
# Data Encoding Scheme
# Random Forest Data Encoding Scheme
RANDOM_FOREST_DATA_ENCODING_SCHEME = {
    BREAST_CANCER_ID: "label",
    CAR_EVALUATION_ID: "ordinal",
    ECOLI_ID: "label",
    LETTER_RECOGNITION_ID: "ordinal",
    MUSHROOM_ID: "label",
}
# Artificial Neural Network Data Encoding Scheme
ANN_DATA_ENCODING_SCHEME = {
    BREAST_CANCER_ID: "label",
```

```
    CAR_EVALUATION_ID: "ordinal",
    ECOLI_ID: "label",
    LETTER_RECOGNITION_ID: "ordinal",
    MUSHROOM_ID: "label",
}
# AdaBoost Data Encoding Scheme
ADABOOST_DATA_ENCODING_SCHEME = {
    BREAST_CANCER_ID: "one-hot",
    CAR_EVALUATION_ID: "one-hot",
    ECOLI_ID: "one-hot",
    LETTER_RECOGNITION_ID: "one-hot",
    MUSHROOM_ID: "one-hot",
}
# KNN Data Encoding Scheme
KNN_DATA_ENCODING_SCHEME = {
    BREAST_CANCER_ID: "label",
    CAR_EVALUATION_ID: "ordinal",
    ECOLI_ID: "label",
    LETTER_RECOGNITION_ID: "ordinal",
    MUSHROOM_ID: "label",
}
```

controllers.py

```
"""
Implementation of the Controller Layer for the Machine Learning Model Training and Evaluation Pipeline.
This module is responsible for orchestrating the training and evaluation of the machine learning models.

Author
------
@rkalai, rishabh.kalai@unb.ca
"""
# Import the necessary libraries
# 3rd party libraries
import numpy as np

# Custom libraries
import constants as CONSTANTS
import data_preprocessing as PREPROCESSING
import model_evaluation as EVALUATION
# Model Libraries
from adaboost import AdaBoostMultiClass
from ann import ANN
from knn import KNN
from random_forest import RandomForest


def perform_model_training(model, ID: str = None, hyperparameters: dict = None, encoding_scheme: str =
None, scale_data: bool = False):
    # Select the dataset from the UCIML Repository
    X, y, feature_headers, target_headers, dataset_name = PREPROCESSING.select_uci_dataset(dataset_id=ID,
                                                                                           categorical_encoding_scheme=encoding_scheme,
scale_data=scale_data)
    if model is None:
        # The model(usually ANN) is not initialized as it needs the input size and output size
        model = ANN(input_size=X.shape[1], output_size=len(np.unique(y)), **hyperparameters)
    # Initialize the model
    if hyperparameters is None:
        best_model, best_params, accuracy = EVALUATION.hyperparameter_search_and_train(
            model=model,
            param_grid=hyperparameters,
            X=X,
            y=y
        )
    else:
        best_model = model
        best_params = hyperparameters
    # Train the model using Cross-Validation
    best_model_metrics, accuracies, recalls, precisions, fscores =
EVALUATION.evaluate_model_performance(best_model, X, y)
    # Calculate the Standard Deviation & Mean of the Accuracies
    # Find the Average and Standard Deviation of the Data
    average_accuracy = np.mean(accuracies)
    std_dev_accuracy = np.std(accuracies)
```

```
    # Create the Plot Title for the Histogram of the Accuracies from the Cross-Validated Model Performance
    plot_title = f"{model.__class__.__name__} Accuracy Histogram: {dataset_name}\nAverage:
{average_accuracy:.2f}% | Standard Deviation: {std_dev_accuracy:.2f}%"
    # Plot the Histogram of the Accuracies from the Cross-Validated Model Performance
    EVALUATION.plot_histogram(accuracies, horizontal_axis_label="Accuracy",
vertical_axis_label="Frequency",
                              plot_title=plot_title)
    # Plot the Confusion Matrix of the Best Model
    EVALUATION.plot_confusion_matrix(y_test=best_model_metrics["true_values"],
y_pred=best_model_metrics["predictions"],
                                     target_names=target_headers, dataset_name=dataset_name)
    return best_model_metrics, best_params, average_accuracy, std_dev_accuracy


def train_and_validate_random_forest(ID: str = None, hyperparameter_search: bool = False):
    if not hyperparameter_search:
        hyperparameters_rf = CONSTANTS.RANDOM_FOREST_HYPERPARAMETERS[ID]
        random_forest = RandomForest(**hyperparameters_rf)
    else:
        hyperparameters_rf = None
        random_forest = RandomForest()
    encoding_scheme_rf = CONSTANTS.RANDOM_FOREST_DATA_ENCODING_SCHEME[ID]
    return perform_model_training(random_forest, ID, hyperparameters_rf, encoding_scheme_rf)


def train_and_validate_adaboost(ID, hyperparameter_search: bool = False):
    if not hyperparameter_search:
        hyperparameters_adaboost = CONSTANTS.ADABOOST_HYPERPARAMETERS[ID]
        adaboost = AdaBoostMultiClass(**hyperparameters_adaboost)
    else:
        hyperparameters_adaboost = None
        adaboost = AdaBoostMultiClass()
    encoding_scheme_adaboost = CONSTANTS.ADABOOST_DATA_ENCODING_SCHEME[ID]
    return perform_model_training(adaboost, ID, hyperparameters_adaboost, encoding_scheme_adaboost)


def train_and_validate_ann(ID, hyperparameter_search: bool = False):
    if not hyperparameter_search:
        hyperparameters_ann = CONSTANTS.ANN_HYPERPARAMETERS[ID]
        ann = None
    else:
        hyperparameters_ann = None
        ann = None
    encoding_scheme_ann = CONSTANTS.ANN_DATA_ENCODING_SCHEME[ID]
    return perform_model_training(ann, ID, hyperparameters_ann, encoding_scheme_ann, scale_data=True)


def train_and_validate_knn(ID):
    knn = KNN()
    encoding_scheme_knn = CONSTANTS.KNN_DATA_ENCODING_SCHEME[ID]
    return perform_model_training(knn, ID, {}, encoding_scheme_knn)
```

data_preprocessing.py

```
"""
Implemention of the Data Preprocessing module. This module is responsible for performing the data preprocessing on the dataset.

Author
------
@rkalai
"""
# 3rd party libraries
import numpy as np
import pandas as pd
import scipy.stats as stats
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (
    StandardScaler,
    MinMaxScaler,
    LabelEncoder,
    OrdinalEncoder,
    OneHotEncoder,
)
# Import the libraries for the UCIML Repository
from ucimlrepo import fetch_ucirepo

# Import the necessary libraries
# Custom libraries
import constants as CONSTANTS

# GLOBAL SETTINGS
# Set Seed for reproducibility
np.random.seed(CONSTANTS.RANDOM_STATE)


def select_uci_dataset(
        dataset_id: str = None,
        categorical_encoding_scheme: str = 'label',
        impute_missing_values_method: str = "mean",
        drop_missing_values: bool = True,
        scale_data: bool = False,
        outlier_treatment_method=None,
):
    """
    Select the dataset from the UCIML Repository based on the dataset ID.

    Parameters
    ----------
    dataset_id (str): The ID of the dataset to be selected.
    categorical_encoding_scheme (str): The encoding scheme for the categorical variables. Default is 'label', but can also be 'one-hot' or 'ordinal'.
    impute_missing_values_method (str): The method to impute the missing values. Default is 'mean', but can also be 'median' or 'mode'.
```

```
    drop_missing_values (bool): Whether to drop the missing values or impute them.
    scale_data (bool): Whether to scale the data or not.
    outlier_treatment_method (str): The method to treat the outliers. Default is None, but can also be 'iqr' or 'z-score'.

    Returns
    -------
    pd.DataFrame: The dataset from the UCIML Repository retrieved based on the dataset ID.
    """
    uci_dataset = fetch_ucirepo(id=dataset_id)
    # Create a Mapping of the type of encoding that is to be performed for each Dataset
    features = uci_dataset.data.features.columns
    targets = uci_dataset.data.targets.columns
    # Drop any columns that are not features or targets
    modelling_data = uci_dataset.data.original[list(features) + list(targets)]
    dataset_name = uci_dataset.metadata.name
    # Preprocess the data
    modelling_data = data_preprocessing(
        data=modelling_data,
        drop_missing_values=drop_missing_values,
        impute_missing_values_method=impute_missing_values_method,
        encode_categorical_variables=categorical_encoding_scheme,
        scale_data=scale_data,
        outlier_treatment_method=outlier_treatment_method,
    )
    # Feature Headers, Target Headers
    feature_headers = [
        header for header in modelling_data.columns if any(feature_name in header for feature_name in features)
    ]
    target_headers = [
        header for header in modelling_data.columns if any(target_name in header for target_name in targets)
    ]
    # Assuming `data` is your DataFrame after preprocessing
    X = modelling_data[feature_headers].reset_index(drop=True)
    y = modelling_data[target_headers].reset_index(drop=True)
    # Convert to numpy arrays if they're not already
    X = X.to_numpy()
    y = y.to_numpy()
    # If either of the dimensions is 1, then convert it to a 1D array
    if X.shape[1] == 1:
        X = X.ravel()
    if y.shape[1] == 1:
        y = y.ravel()
    return X, y, feature_headers, target_headers, dataset_name


def data_preprocessing(
        data: pd.DataFrame = None,
        drop_missing_values: bool = False,
        impute_missing_values_method: str = "mean",
        encode_categorical_variables: str = "label",
        scale_data: bool = False,
```

```
    outlier_treatment_method=None,
):
    """
    Perform data preprocessing on the dataset. This includes:
    1. Missing Value Treatment: Imputation with the mean/mode/median of the column or drop the missing values.
    2. Data Transformation: Encode the categorical variables to numerical variables using Label Encoding or One-Hot Encoding. Default is Label Encoding.
    3. Data Scaling: Scale the data using Standardization or Min-Max Scaling.
    4. Outlier Treatment: Treat the outliers using the specified method, if None then no treatment is performed.

    Parameters
    ----------
    data (pd.DataFrame): The dataset to be preprocessed.
    drop_missing_values (bool): Whether to drop the missing values or impute them.
    impute_missing_values_method (str): The method to impute the missing values. Default is 'mean', but can also be 'median' or 'mode'.
    encode_categorical_variables (str): The method to encode the categorical variables. Default is 'label', but can also be 'one-hot' or 'ordinal'. If None, then no encoding is performed.
    scale_data (bool): Whether to scale the data or not.
    outlier_treatment_method (str): The method to treat the outliers. Default is None, but can also be 'iqr' or 'z-score'.

    Returns
    -------
    pd.DataFrame: The preprocessed dataset.
    """
    # Segregate the data into Categories and Numerical variables
    # For Categories, impute the missing values with the mode, regardless of the impute_missing_values_method
    # For Numerical variables, impute the missing values with the mean/median, based on the impute_missing_values_method
    if drop_missing_values:
        data = data.dropna(how="any", axis=0)
        data = data.reset_index(drop=True)
    categorical_data = data.select_dtypes(include="object")
    categorical_columns = categorical_data.columns
    numerical_data = data.select_dtypes(include="number")
    if not numerical_data.empty:
        if impute_missing_values_method == "mean":
            imputer = SimpleImputer(strategy="mean")
        elif impute_missing_values_method == "median":
            imputer = SimpleImputer(strategy="median")
        elif impute_missing_values_method == "mode":
            imputer = SimpleImputer(strategy="most_frequent")
        else:
            raise ValueError(
                'Invalid imputation method. Please specify either "mean", "median" or "mode".'
            )
        # Impute the values in the numerical data, and return a DataFrame with the imputed values
        numerical_data = pd.DataFrame(
            imputer.fit_transform(numerical_data), columns=numerical_data.columns
        )
```

```
        # Treat the outliers in the dataset using the specified method
        if outlier_treatment_method:
            numerical_data = outlier_treatment(
                numerical_data, method=outlier_treatment_method
            )
        if scale_data:
            numerical_data = perform_data_scaling(numerical_data)
    if encode_categorical_variables and not categorical_data.empty:
        categorical_data = categorical_data.apply(lambda x: x.fillna(x.mode().iloc[0]))
        # Encode the categorical variables to numerical variables using Label Encoding or One-Hot Encoding
        # Label Encoding: Convert the categories to numerical values - Set as Default
        # One-Hot Encoding: Create dummy variables for each category
        if encode_categorical_variables == "label":
            encoder = LabelEncoder()
            encoded_data = categorical_data.apply(encoder.fit_transform)
            encoded_data_columns = categorical_columns
        elif encode_categorical_variables == "ordinal":
            encoder = OrdinalEncoder()
            encoded_data = encoder.fit_transform(categorical_data)
            encoded_data_columns = categorical_columns
        elif encode_categorical_variables == "one-hot":
            encoder = OneHotEncoder(sparse_output=False)
            encoded_data = encoder.fit_transform(categorical_data)
            encoded_data_columns = encoder.get_feature_names_out(categorical_columns)
        else:
            raise ValueError(
                'Invalid encoding method. Please specify either "label" or "one-hot".'
            )
        # Convert the encoded data into a DataFrame with appropriate column names
        encoded_df = pd.DataFrame(data=encoded_data, columns=encoded_data_columns)
        # Convert all the columns of the Encoded DataFrame to int type
        encoded_df = encoded_df.astype(int)
        # Concatenate the original numerical data with the new one-hot encoded data
        data = pd.concat([numerical_data, encoded_df], axis=1).reset_index(drop=True)
    return data


def perform_data_scaling(data: pd.DataFrame = None, scaling_method: str = "min-max"):
    """
    Scale the dataset using the specified scaling method.

    Parameters
    ----------
    data (pd.DataFrame): The dataset to be scaled.
    scaling_method (str): The method to scale the dataset. Default is 'standard'.

    Returns
    -------
    pd.DataFrame: The scaled dataset.
    """
    if scaling_method == "standard":
        scaler = StandardScaler()
```

```
    elif scaling_method == "min-max":
        scaler = MinMaxScaler()
    else:
        raise ValueError(
            'Invalid scaling method. Please specify either "standard" or "min-max".'
        )
    data = pd.DataFrame(scaler.fit_transform(data), columns=data.columns)
    return data

def outlier_treatment(data: pd.DataFrame = None, method: str = "iqr"):
    """
    Treat the outliers in the dataset using the specified method.

    Parameters
    ----------
    data (pd.DataFrame): The dataset to be treated for outliers.
    method (str): The method to treat the outliers. Default is 'iqr', but can also be 'z-score'.

    Returns
    -------
    pd.DataFrame: The dataset with the treated outliers.
    """
    if method == "iqr":
        # The way to treat the outliers is by removing them from the dataset
        # if the value is less than Q1 - 1.5 * IQR or greater than Q3 + 1.5 * IQR
        Q1 = data.quantile(0.25)
        Q3 = data.quantile(0.75)
        IQR = Q3 - Q1
        data = data[
            ~((data < (Q1 - 1.5 * IQR)) | (data > (Q3 + 1.5 * IQR))).any(axis=1)
        ]
    elif method == "z-score":
        z = np.abs(stats.zscore(data))
        data = data[(z < 3).all(axis=1)]
    else:
        raise ValueError('Invalid outlier treatment method. Please specify "iqr".')
    data.reset_index(drop=True, inplace=True)
    return data
```

model_evaluation.py

```
"""
Implement the preprocessing of the data.

Author:
@rkalai
"""
# Plotting Libraries
import matplotlib.pyplot as plt
# 3rd party libraries for Model Evaluation
import numpy as np
import seaborn as sns
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import GridSearchCV, train_test_split, RepeatedStratifiedKFold

# Import the necessary libraries
# Custom libraries
import constants as CONSTANTS

# GLOBAL SETTINGS
# Set Seed for reproducibility
np.random.seed(CONSTANTS.RANDOM_STATE)

import numpy as np
from sklearn.base import clone
from sklearn.metrics import classification_report
from concurrent.futures import ProcessPoolExecutor

def train_and_evaluate_fold(ml_model, X_train_fold, X_test_fold, y_train_fold, y_test_fold):
    cloned_model = clone(ml_model)
    cloned_model.fit(X_train_fold, y_train_fold)
    y_pred = cloned_model.predict(X_test_fold)
    # If y_pred is 2D, convert it to 1D
    if isinstance(y_pred, np.ndarray):
        if y_pred.ndim == 2:
            y_pred = y_pred.ravel()
            y_test_fold = y_test_fold.ravel()
    cr = classification_report(y_test_fold, y_pred, output_dict=True)
    accuracy = cr['accuracy'] * 100
    recall = cr['macro avg']['recall']
    precision = cr['macro avg']['precision']
    fscore = cr['macro avg']['f1-score']

    return accuracy, recall, precision, fscore, cloned_model,y_pred

def evaluate_model_performance(ml_model=None, X: np.array = None, y: np.array = None,
                               n_splits: int = CONSTANTS.N_SPLITS, n_repeats: int = CONSTANTS.N_REPEATS):
    accuracies = []
    recalls = []
    precisions = []
    fscores = []
```

```
    best_model = None

    rskf = RepeatedStratifiedKFold(n_splits=n_splits, n_repeats=n_repeats,
random_state=CONSTANTS.RANDOM_STATE)

    for train_index, test_index in rskf.split(X, y):
        X_train_fold, X_test_fold = X[train_index], X[test_index]
        y_train_fold, y_test_fold = y[train_index], y[test_index]
        accuracy, recall, precision, fscore, cloned_model, y_pred = train_and_evaluate_fold(ml_model,
X_train_fold, X_test_fold, y_train_fold, y_test_fold)

        if best_model is None or accuracy > best_model['accuracy']:
            best_model = {
                'accuracy': accuracy,
                'recall': recall,
                'precision': precision,
                'fscore': fscore,
                'model': cloned_model,
                'true_values': y_test_fold,
                'predictions': y_pred,
            }
        accuracies.append(accuracy)
        recalls.append(recall)
        precisions.append(precision)
        fscores.append(fscore)
    return best_model, accuracies, recalls, precisions, fscores

def plot_confusion_matrix(y_test: np.array = None, y_pred: np.array = None, target_names: list = None,
                          dataset_name: str = None):
    """
    Plot the confusion matrix as a heatmap.

    Parameters
    ----------
    y_test : array-like
        The true labels.
    y_pred : array-like
        The predicted labels.
    target_names : list
        The names of the target classes.
    dataset_name : str
        The name of the dataset.
    """
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, cmap="Blues", fmt="g")
    plt.xticks(ticks=np.arange(0.5, len(target_names)), labels=target_names)
    plt.yticks(ticks=np.arange(0.5, len(target_names)), labels=target_names)
    plt.xlabel("Predicted")
    plt.ylabel("Gold Standard")
    plt.title(f"Confusion Matrix: {dataset_name}")
    plt.show()
```

```
def plot_histogram(data: list = None,
                   horizontal_axis_label: str = "Metric",
                   vertical_axis_label: str = "Frequency",
                   plot_title: str = None):
    """
    Plot the histogram of the data.
    Parameters
    ----------
    data : list
        The list of data to be plotted.
    horizontal_axis_label : str
        The label of the horizontal axis.
    vertical_axis_label : str
        The label of the vertical axis.
    plot_title : str
        The title of the plot.
    """
    # Plot the Histogram of the Data
    plt.figure(figsize=(10, 7))
    sns.histplot(data, kde=True)
    plt.xlabel(horizontal_axis_label)
    plt.ylabel(vertical_axis_label)
    plt.title(plot_title)
    plt.show()

def hyperparameter_search_and_train(model, param_grid, X, y, test_size=CONSTANTS.TEST_SIZE):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,
random_state=CONSTANTS.RANDOM_STATE)
    # Create an instance of the GridSearchCV
    grid_search = GridSearchCV(model, param_grid, cv=3, n_jobs=-1)
    grid_search.fit(X_train, y_train)
    best_params = grid_search.best_params_
    best_model = grid_search.best_estimator_
    # Train the best model on the entire training set
    best_model.fit(X_train, y_train)
    # Evaluate the best model on the test set
    y_pred = best_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    return best_model, best_params, accuracy
```

utilities.py

```
"""
Implementation of the Utilities module. This module contains the mathematical and data manipulation
functions.

Author
------
@rkalai, rishabh.kalai@unb.ca
"""
# INDEX OF FUNCTIONS:
# 1. MATHEMATICAL FUNCTIONS

# Import the necessary libraries
# Custom libraries

# 3rd party libraries
import numpy as np


# ---------------------------------- 1. MATHEMATICAL FUNCTIONS ---------------------------------- #
def softmax(x):
    """
    Compute the softmax of vector x.

    The softmax function, also known as softargmax or normalized exponential function,
    is a function that takes as input a vector of K real numbers, and normalizes it into
    a probability distribution consisting of K probabilities. That is, prior to applying
    softmax, some vector components could be negative, or greater than one; and might not
    sum to 1; but after applying softmax, each component will be in the interval (0,1),
    and the components will add up to 1, so that they can be interpreted as probabilities.
    Furthermore, the larger input components will correspond to larger probabilities.

    Softmax is often used in neural networks, to map the non-normalized output of a network
    to a probability distribution over predicted output classes.

    Parameters:
    x (numpy array): Input Vector or Matrix.

    Returns:
    numpy array: Softmax of the input vector.
    """
    # Softmax Formula: e^x / sum(e^x)
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)


def cross_entropy_loss(y_pred, y_true):
    """
    Compute the cross-entropy loss.

    Cross-entropy loss, or log loss, measures the performance of a classification model
```

```
    whose output is a probability value between 0 and 1. Cross-entropy loss increases as
    the predicted probability diverges from the actual label. It is used in machine learning
    and statistics as a measure of difference between two probability distributions.
    Parameters:
    y_pred (numpy array): Predicted values.
    y_true (numpy array): True values.

    Returns:
    float: Cross entropy loss.
    """
    m = y_true.shape[0]
    log_y_pred = np.log(y_pred)
    product = y_true * log_y_pred
    epsilon = 1e-12
    sum_product = np.sum(product + epsilon)
    loss = -sum_product / m
    return loss


def derivative_cross_entropy_softmax(y_pred, y_true):
    """
    Compute the derivative of the cross-entropy loss with respect to softmax.

    This function calculates the gradient of the cross-entropy loss with respect to the
    softmax function, which is used in the backpropagation process of training a neural
    network. The derivative is simply the difference between the predicted and true values.

    Parameters:
    y_pred (numpy array): Predicted values.
    y_true (numpy array): True values.

    Returns:
    numpy array: Derivative of the cross-entropy loss with respect to softmax.
    """
    derivative = y_pred - y_true
    return derivative

def mean_squared_error(y_pred, y_true):
    """
    Compute the mean squared error.

    Mean squared error (MSE) is a common loss function used for regression problems.
    It calculates the average squared difference between the predicted and true values,
    giving a measure of prediction error. The squaring ensures that larger errors are
    more significant than smaller ones.

    Parameters:
    y_pred (numpy array): Predicted values.
    y_true (numpy array): True values.

    Returns:
    float: Mean squared error.
```

```
    """
    diff = y_pred - y_true
    squared_diff = np.square(diff)
    mse = np.mean(squared_diff)
    return mse

# ---------------------------------- 1. END OF MATHEMATICAL FUNCTIONS ---------------------------------- #

# ---------------------------------- 2. DATA MANIPULATION FUNCTIONS ---------------------------------- #

def bootstrap_samples(X, y):
    """
    Bootstrapping is the process of sampling with replacement. This function returns the bootstrapped samples of the dataset.
    This usually includes a random subset of the rows of the dataset. The number of samples is the same as the original dataset.
    The reason for the number of samples being the same as the original dataset is to ensure that the Random Forest has the same number of samples as the original dataset.
    The importance of bootstrapping is to ensure that the Random Forest has a diverse set of samples to train on.

    Parameters
    ----------
    X (np.ndarray): The features of the dataset.
    y (np.ndarray): The target variable of the dataset.

    Returns
    -------
    X_bootstrap_samples (np.ndarray): The bootstrapped samples of the features.
    y_bootstrap_samples (np.ndarray): The bootstrapped samples of the target variable.
    """
    n_samples = X.shape[0]
    indices = np.random.choice(n_samples, size=n_samples, replace=True)
    X_bootstrap_samples = X[indices]
    y_bootstrap_samples = y[indices]
    return X_bootstrap_samples, y_bootstrap_samples

# ------------------------------- 2. END OF DATA MANIPULATION FUNCTIONS -------------------------------#
```