CS235 Fall'23 Project Final Report: Income Classification using Census Data

Aditya Gambhir
University of California, Riverside
USA
agamb031@ucr.edu

Ajit Singh
University of California, Riverside
USA
asingh349@ucr.edu

Achala Shashidhara Pandit
University of California, Riverside
USA
apand048@ucr.edu

Anvith Reddy Nemali
University of California, Riverside
USA
asingh349@ucr.edu

Abstract

This project presents a thorough investigation into binary classification using the Adult Census Income Dataset from the UCI ML Repository. The primary objective is to implement four distinct data mining techniques—Random Forest, Artificial Neural Network, AdaBoost, and Decision Tree—both through the widely-used sklearn library and from scratch using Numpy and Pandas.

The study entails a meticulous process, encompassing data preprocessing to address challenges such as duplicate values, missing data, and categorical variable encoding. For each method, we perform hyperparameter tuning during the sklearn implementation and subsequently transfer the optimized parameters to scratch-built models. K-fold Cross-Validation is employed for both implementations to ensure robust model evaluation.

The comparison of the performance metrics, including accuracy, precision, recall, and F1-score, serves as the cornerstone of our analysis. This evaluation is conducted for each implementation of every technique, providing insights into the nuances of library-based versus custom-built models. Furthermore, a cross-technique comparison is performed to discern the strengths and weaknesses inherent in each approach.

The project's findings highlight not only the efficacy of each model but also the impact of hyperparameter tuning on the performance of both sklearn and scratch implementations. This work contributes valuable insights into the trade-offs associated with employing established libraries versus bespoke implementations, offering a comprehensive perspective on the applicability of different data mining techniques for binary classification tasks. The outcomes of this research are poised to inform future endeavors in machine learning and data mining, especially in domains dealing with income prediction and demographic analysis.

Keywords: Data Mining, Census Income, Random Forest, Decision Tree, AdaBoost, Artificial Neural Network

ACM Reference Format:
Aditya Gambhir, Ajit Singh, Achala Shashidhara Pandit, and Anvith Reddy Nemali. 2024. CS235 Fall'23 Project Final Report: Income Classification using Census Data. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In today's data-driven era, understanding and predicting income levels has become essential for a broad spectrum of applications, including macroeconomic policy-making and personal financial planning.

Our project, "Income Classification using Census Data," aims to uncover patterns and obtain insights from an assortment of socio-economic factors, which, in turn, are used to classify an individual's income level. The UCI ML Repository sourced the Adult Census Income Dataset, on which we implemented four prominent data mining techniques, namely - Random Forest, Artificial Neural Networks, AdaBoost, and Decision Tree. We employed a dual implementation approach, with one approach utilizing off-the-shelf functions from the sklearn library and the other involving the construction of these procedures from scratch using the Numpy and Pandas libraries.

We meticulously evaluated each model's performance using key metrics such as accuracy, precision, recall, and F1-score, facilitating an analysis of their advantages and limitations. This study intends to contribute to a better understanding of binary classification in income prediction while also

paving the way for more informed decisions in similar future machine-learning endeavors. The GitHub repository for the DMT Project can be found at: https://github.com/Aditya-gam/DMT-Project.

2 Problem Definition

We aim to address the challenge of accurately categorizing individuals into distinct income brackets based on a diverse set of demographic and employment-related features. The dataset at hand holds significant practical relevance, offering applications in predicting income, optimizing targeted marketing strategies, evaluating credit risk, and gaining insights into income disparities.

3 Related Work

The paper titled *Improving Performance Analysis in Classification with Accuracy of Adult Income Salary using Novel Gated Residual Neural Network in Comparison with Logistic Regression Algorithm* by Bhuvana Chandra. M and N. Deepa[4] focus on using a Novel Gated Residual Neural Network (NGRNN) for predicting adult income and compare its performance against the Logistic Regression algorithm. Utilizing a dataset of 32,561 samples for classifying adult income, the NGRNN achieved a notably high accuracy of 95%, significantly surpassing the 74% accuracy achieved by Logistic Regression. The work titled "Exploring Random Forests: Bridging Theory and Application" authored by Gilles Louppe et. al. delivers a thorough examination of random forests[3]. The exploration commences with an in-depth investigation of decision trees, the fundamental components of random forests, before delving into the intricacies of their assembly and functionality. Notably, the paper introduces an innovative complexity analysis of random forests, shedding light on their computational efficiency and scalability. The paper "Deep Residual Learning for Image Recognition" by Kaiming He et al. introduces deep residual networks (ResNets), a new neural network architecture[2]. The key innovation of ResNets is the use of "residual learning" where layers are reformulated to learn residual functions regarding layer inputs. This approach addresses the degradation problem common in very deep networks, where accuracy saturates and then degrades rapidly. The paper "Study of Salary Differentials by Gender and Discipline" by L. Billard focuses on analyzing salary disparities in academic institutions[1]. It employs multiple regression models to investigate how salaries depend on various factors, including gender and academic discipline. The paper emphasizes the importance of including interaction terms in models to accurately capture the complex dynamics of salary differentials. A notable result is that gender disparities in salary persist even after accounting for factors like rank and experience, suggesting systemic biases. The paper "Induction of Decision Trees" by J.R. Quinlan, published in 1986, is a foundational work in machine learning[5]. It focuses on the methodology for building knowledge-based systems through inductive inference from examples. The paper details the synthesis of decision trees and describes the ID3 system in depth. Key results include the development of methods to handle noisy and incomplete information.

4 Dataset

The Adult Census Income Dataset, meticulously curated by Barry Becker from the 1994 Census database, serves as a comprehensive repository of socio-economic attributes with the overarching goal of predicting whether an individual earns more than 50K a year. Under specific extraction conditions, the dataset encapsulates a diverse set of features, including age, education, occupation, and more.

In terms of key characteristics, the dataset comprises 48,842 records encompassing 15 features, with notable attributes such as age, education level, occupation, capital gains and losses, hours per week worked, and native country. The focal point of analysis revolves around the target variable 'income,' which categorizes individuals based on their earnings.

Descriptive statistics reveal intriguing insights into the dataset. The age of individuals spans from 17 to 90, with an average of approximately 38.6 years. Education levels vary from Preschool to Doctorate, reflecting a broad spectrum of academic achievements. Occupations are diverse, ranging from tech support to Armed Forces, reflecting the varied employment landscape. Work hours per week exhibit an average of around 40, with a minimum of 1 and a maximum of 99.

In light of data preprocessing considerations, several aspects warrant attention. Notably, the 'workclass,' 'occupation,' and 'native-country' columns exhibit missing values, necessitating imputation or removal. Additionally, categorical variables such as 'workclass,' 'education,' 'marital status,' 'occupation,' 'relationship,' 'race,' 'sex,' and 'native country' may require encoding for effective modeling. Outliers, particularly in 'capital-gain' and 'capital-loss,' demand scrutiny due to their potential impact on model performance. Furthermore, encoding considerations may extend to the 'income' variable for effective handling in binary classification tasks.

5 Proposed Approach

Our approach to income classification leverages a multifaceted strategy, integrating cutting-edge machine learning algorithms and meticulous data preprocessing to navigate the complexities of the task. The initial phase involves a meticulous exploration of the dataset, revealing intricate relationships and potential challenges inherent in unbalanced data, notably the dominance of class 0 (income $\leq$ 50K).

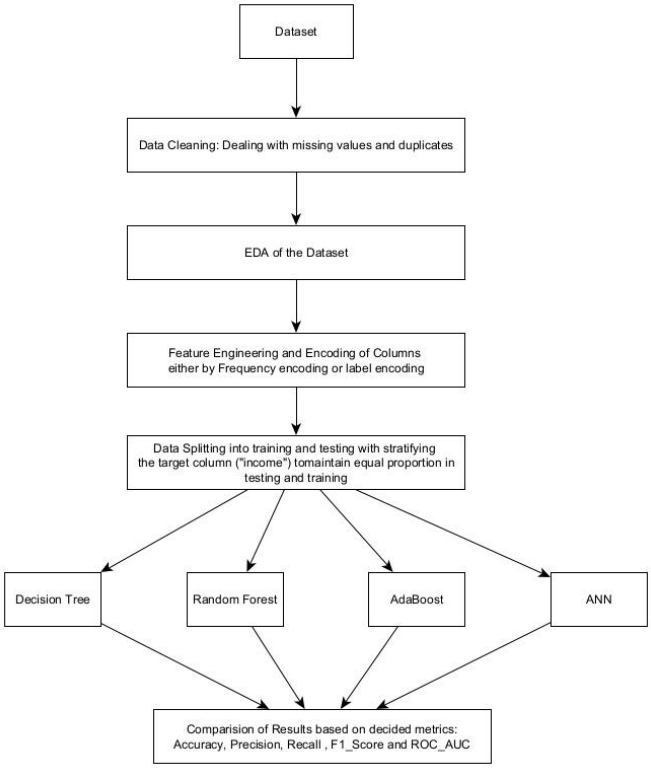

Figure 1. Project Pipeline

5.1 Data Preprocessing

Addressing the intricacies of the dataset is paramount. Our pre-processing pipeline includes the handling of duplicate values, addressing missing data, and encoding categorical variables. to ensure a robust foundation for subsequent analyses. As the total number of duplicates in the whole dataset was fewer than 25, it was dropped as it was inconsequential. Later, we scanned the dataset for missing values (NaN) and special characters like ['?']. Following this identification, we substituted the mode for the missing/special values in the 'native-country' and 'workclass' categories, as well as a new value 'Other' in the 'occupation' category. The target variable 'Income' was encoded where (<=50k) is 0 and (>50k) is 1. Feature Engineering entailed encoding categorical features utilizing approaches such as One-Hot-Encoding, Label Encoding, and Frequency Encoding. Considering we're working with ANN, the numerical features have been scaled using standard and min-max scaling. Additionally, we've dropped irrelevant features, in our case education was dropped as it was redundant, and a numerical representation of the same, called 'education-num' was already present, The dataset is then split into three parts, namely training, validation, and testing, on which we'll run each of our models. The model's performance is compared using a variety of performance metrics such as accuracy, precision, f1-score, recall, and a confusion matrix. These indicators will be used to make the Selection

5.2 Algorithm Selection

Considering the type of classification and scale of our dataset, our algorithmic arsenal comprises Random Forest Classification, Artificial Neural Networks (ANN), Adaptive Boosting (AdaBoost), and Decision Trees. Each team member focuses on one specific algorithm, delving into both off-the-shelf implementations and custom-built models. This approach enables a nuanced comparison, shedding light on the advantages and limitations of each methodology.

5.3 Off-the-Shelf Implementations

We use sci-kit Learn and TensorFlow well-established machine learning models to train models on the dataset at hand. This also includes fine-tuning hyper-parameters, and optimizing each model's performance which are achieved through RandomizedSearchCV. Evaluation metrics, including accuracy, precision, recall, and F1-score, provide a comprehensive understanding of their effectiveness.

5.4 Custom Implementations

In addition to off-the-shelf models, the team undertakes the ambitious task of crafting custom implementations of Random Forest, ANN, AdaBoost, and Decision Trees. This initiative allows for a direct comparison with library-based models, offering insights into the impact of implementation choices on scalability, performance, and interpretability.

6 Experiments

Detail the experiments conducted for each data mining technique, including implementation details.

6.1 Random Forest Classification

6.1.1 Introduction: A random forest (RF) is a meta-estimator that fits several decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the max samples parameter if bootstrap = True (default), otherwise, the whole dataset is used to build each tree.

6.1.2 Baseline Model: Off-the-shelf baseline model: The library used to import the baseline model is 'sklearn.ensemble'. RF is a bagging ensemble method of decision trees. Following the base model it was determined that hyper tuning of the parameters of RF should be done to determine if it would help increase the performance of the model. The following are the top 5 feature classifications:

Table 1. Feature Importance

Feature	Importance
fnlwgt	0.23530280519077287
education-num	0.1247166693670773
capital-gain	0.11665151176241312
relationship	0.1026755730504701
hours-per-week	0.09222174834296967

6.1.3 Hyperparameter Tuning.

The RF model has a lot of hyperparameters to fine-tune such as n_estimators, max_depth, min_samples_split, max_features, bootstrap, max_samples, etc.

Choosing the parameters to fine-tune was a difficult task. For this understanding how the chosen parameters affect the performance of the RF is important. The following parameters were chosen: Max_depth: max depth of the trees Max_features: no of features to consider for the best split Min_samples_split: min sample required to split an internal node Bootstrap: boolean value that dictates whether to bootstrap samples or not criterion: The function used to measure the quality of a split. It can be "gini" for Gini impurity or "entropy" for information gain.

To tune the parameter and to select the best tuning, the Random Search tuning method was used from the sklearn library. In this, the parameter is defined and a range is given the random search algorithm tries to randomize the selection process of the tuning and chooses the best possible combination of tuning for each parameter.

The above parameters were tuned and the best possible combination was: 'bootstrap': True, 'criterion': 'gini', 'max_depth': 5, 'max_features': 9, 'min_samples_split': 28 The decision to use random search over grid search was based on computational limitations as well as time to execute the algorithm. Moreover, random search is more efficient and flexible. The choice of the hyperparameters in this case was determined by factors like computational resources, and influence on the tree moreover these are some of the most commonly tuned hyperparameters, and tuning these for the given data set has shown a small increase in accuracy and a significant increase in precision however recall power of the model reduces.

6.1.4 Custom Implementations.

To make a custom RF a custom Decision tree was needed so both Anvith and I worked on both the sections below:

We created a custom Decision Tree structure that was defined by the following parameters: **Node Class**: A decision tree node's representation. It keeps track of data such as the value for leaf nodes, gain, child nodes, feature index, and splitting threshold.

Decision Tree Constructor: Sets up the tree's initial conditions, including the maximum tree depth, minimum sample split, and splitting criterion (entropy or Gini).

Fit Method: Uses a recursive `build_decision_tree` method to construct the decision tree after combining features and target variables into a single dataset.

Building the Tree: Using the given criterion (information gain or Gini impurity), the optimum split is found iteratively to build the tree.

Prediction: Based on the value of the leaf node, a prediction is produced for a particular input once the tree has been walked to reach it.

The Random forest was an ensemble of Decision Trees, so we used bagging. The '`RandomForestClassifier_custom`', the custom implementation entails:

Random Forest Constructor: Sets up initial conditions such as minimum sample split, maximum depth, criteria, and the number of trees. **Random Forest Constructor (`RandomForestClassifier_custom`)**: This is the initialization method that sets up the Random Forest classifier with essential parameters like the number of trees (`n_trees`), minimum samples for a split (`min_samples_split`), maximum depth of trees (`max_depth`), number of features to consider for each split (`n_features`), and the criterion for splitting (`criterion`). These parameters are critical as they control the complexity of the model, its ability to generalize, and its performance on various types of data.

Bootstrap Sampling: To provide unpredictability to the model, a bootstrap sample of the data is made for each tree. One important component of the Random Forest algorithm is bootstrap sampling, which is a replacement-based random sampling technique. A bootstrap sample of the training data is made for every tree in the forest. As a result, every tree receives a slightly different interpretation of the data, which promotes diversity in the tree population and lessens overfitting.

Fit Method: A bootstrap sample of the dataset is used to train each tree in the forest. The fit method is responsible for constructing the forest. It iteratively creates decision trees on different subsets of the data (achieved through bootstrap sampling) and stores them in a list. Each tree is an instance of the `DecisionTreeClassifier_custom` class, which is trained independently on its respective sample.

Decision Tree Integration: The Random Forest utilizes the custom decision tree implementation for its base learners. Each tree in the forest is a fully grown decision tree unless constrained by `max_depth`. We have used 5 decision trees. The decision trees make predictions independently, and their results are aggregated to make a final prediction.

Prediction: Combines forecasts from each tree and determines the final prediction by majority vote. The prediction process involves aggregating predictions from all the trees in the forest. For classification tasks, this usually means using a majority voting system. Each tree provides a vote for the class, and the class receiving the majority of votes becomes the model's prediction.

6.2 Artificial Neural Network

Artificial Neural Networks (ANNs) draw inspiration from the structure and functionality of the human brain. They are comprised of interconnected nodes, often referred to as artificial neurons or perceptrons, organized into layers. [1] ANNs serve multiple purposes, including tasks such as pattern recognition, classification, and regression. In the context of this analysis, a specific dataset designed for binary classification was used, and an off-the-shelf model was imported from the TensorFlow[2] library.

6.2.1 Baseline Model: Initially, a baseline neural network model was trained without hyperparameter tuning. The model consisted of one hidden layer with 64 units and a ReLU activation function in the input layer, 32 units with a ReLU activation function in the hidden layer, and one unit with a sigmoid activation function in the output layer. The training process employed the Adam optimizer and binary cross-entropy loss function.

6.2.2 Hyperparameter Tuning: Customizing an ANN involves fine-tuning hyperparameters as the model's performance significantly relies on finding the right combination of these parameters. To pinpoint the optimal hyperparameters, a 'RandomSearch' tuner from the 'keras_tuner'[3] library was employed with the primary goal of maximizing validation accuracy. This process led to the discovery of the most suitable hyperparameters:

- Number of Hidden Layers: 1
- Units in Hidden Layer 1: 192
- Learning Rate: 0.01

Alongside these hyperparameters, the hidden layer incorporated a ReLU activation function, and the output layer featured a sigmoid activation function. The optimization process utilized the Adam optimizer and binary cross-entropy loss function.

6.2.3 Custom Implementations: In the from-scratch implementation of the Artificial Neural Network (ANN), we delved into key components, beginning with activation functions. Specifically, the sigmoid function and its derivative were employed to introduce non-linearity to hidden layers. The sigmoid function, squashing input values between 0 and 1, facilitated the modeling of intricate data relationships. Its derivative, crucial for backpropagation, adjusted network weights during training.

The binary cross-entropy loss function quantified the error between predicted probabilities and true binary labels. Weight initialization, utilizing methods like He, LeCun, Xavier, and random initialization, set the stage for effective weight optimization. Additionally, the Adam and RMSprop optimizers, implemented from scratch, updated network weights. The training process involved forward and backward passes, and performance metrics like accuracy and loss for both training and validation data were tracked over epochs. The hyperparameters used were taken from the result of the hyperparameter tuning done for the library implementation. Finally, predictions were generated for the test data and the model was evaluated using accuracy, precision, recall, and F1-score.

6.2.4 Cross-Validation: To assess the model's robustness and mitigate overfitting, a comprehensive k-fold cross-validation strategy was employed for both the library and from-scratch implementations of the Artificial Neural Network (ANN). For the library implementation, 'StratifiedKFold' was utilized to partition the data into 10 equally sized folds, ensuring a balanced distribution of classes. The training process involved iteratively training on K-1 folds and validating the remaining fold for 20 epochs. Model performance was rigorously evaluated using diverse metrics such as precision, recall, accuracy, F1-score, ROC-AUC, and class-specific accuracies. These metrics were aggregated to provide a holistic assessment of the library-based model's overall performance.

Similarly, for the from-scratch ANN implementation, a k-fold cross-validation approach was adopted to gauge its generalization capability. The dataset was systematically divided into k folds, with the model undergoing training and evaluation k times. Each iteration involved using a different fold as the validation set, ensuring a robust estimate of the model's performance and revealing potential overfitting concerns. The cross-validation experiments for the from-scratch implementation aimed to demonstrate the correctness and efficacy of the model in handling binary classification tasks. The carefully chosen activation functions, loss function, weight initialization methods, and custom optimizers were pivotal elements contributing to the development of a well-performing neural network. Subsequent sections will present detailed results and analyses derived from these cross-validation experiments for both implementations.

6.3 Adaptive Boosting

6.3.1 Introduction: Adaptive Boosting, also called AdaBoost, is a supervised ensemble learning technique. The philosophy behind ensemble learning techniques is to use the advantages of many models, i.e. a single model may not perform well, however many models together may be able to predict better. In the case of boosting algorithms, an initial classifier is trained and tested. The next classifier takes up all the samples that were mispredicted by the first classifier along with newer samples and trained. 'N' number of such classifiers are trained as shown in Fig that are used

[1] Artificial Neural Networks
[2] TensorFlow Documentation
[3] Keras Documentation

together while predicting the class of a new sample. In the case of AdaBoost, each of these classifiers is a decision tree of height one and is called a stump. Every time samples are misclassified and passed on to the next classifier a higher weight is given to these samples to provide greater emphasis to them. Some stumps can also be better than others, hence a significance value is calculated for each stump. When a new sample has to be classified, every stump predicts a certain class and a weighted average based on significance is taken to make a decision.

6.3.2 Baseline Model: The initial model was trained using off-the-shelf sklearn implementation[4] fits additional copies of the classifier on the same dataset but alters weights of incorrectly classified instances in subsequent classifiers as discussed above. The baseline model was trained with default parameters 50 n_estimators, 1.0 learning rate, and SAMME.R algorithm.

- n_estimators: indicates the number of weak learners used in the ensemble technique
- learning rate: weight applied to each classifier at every boosting iteration. A higher learning rate increases the contribution of each classifier.
- SAMME.R/SAMME: With the SAMME.R algorithm, the final weight is indicated by the probability of a sample belonging to a class. On the other hand, SAMME gives a discrete value like 0 or 1 classification.

6.3.3 Hyperparameter Tuning: Hyperparameters for AdaBoost include several weak learners, the learning rate, and the algorithm used to decide the final classification. RandomizedSearchCV [5] is another tool provided by sklearn that helps to identify good values for hyperparameters. Given a set or range of hyperparameters, it randomly selects a subset of these values and evaluates the model based on them to identify a good set of hyperparameters. In this case, 1000 estimators with a learning rate of 1.0 on the SAMME algorithm were chosen.

6.3.4 Custom Implementations. Implementing Adaboost from scratch involved a lot of steps. Adaboost being an ensemble learning technique involves training multiple models. The weak classifiers used for Adaboost are decision trees with a single root node with two leaf nodes. The first step is to initialize the average weights for all samples. Next for every weak learner to be trained a sample of data is prepared with replacement based on the weight assigned to each training sample. A decision stump is trained and normalized misclassifications are computed as error. Further, every model is given an importance metric based on how well it fits the training samples. Finally, the weights of samples that are misclassified are increased by the degree of importance of the model. The new test sample is run through each of the classifiers to predict an outcome. The outcome is calculated from the sign function of the product of all classifier's predictions and their importance.

As we see Adaboost is implemented with many decision stumps, I have also implemented them from scratch. The split of these stumps is decided based on the Gini index and the two splits result in leaf nodes with the value being the class with maximum samples. The algorithm was implemented with a census dataset and the performance is shared in the results section

6.3.5 Cross-Validation: To ensure the model performs well under most circumstances and does not over-fit on data k-fold cross-validation was implemented. 5-fold cross-validation was implemented on both sklearn and from scratch models.%.

6.4 Decision Trees

6.4.1 Introduction: A decision tree classifier is a tree-structured machine learning model that can be used for classification as well as regression. It functions by recursively splitting data into subsets based on the most relevant features and then classifying the data utilizing a tree structure. Each internal node of the decision tree represents an attribute test, each branch represents a test result, and each leaf node stores a class label. Although the decision tree is very capable it can be prone to overfitting and may lead to the creation of biased trees, especially if the data is noisy or imbalanced.

6.4.2 Baseline Model: The library used to implement the model was from the `sklearn` library, specifically the `DecisionTreeClassifier` class. This class has several hyperparameters, such as `max_depth`, `criterion`, `max_features`, `splitter`, `min_samples_leaf`, etc., that can be tuned to improve the performance of the Decision Tree. The `Decision Tree Classifier` algorithm was the base classifier that was used with the training and testing data, and initially, all the parameters belonging to the base classifier were set to their default values.

The technique of feature importance is used to discover which features in a dataset are most crucial in predicting the target variable. Here are the top 5 feature importances determined for the base model:

Table 2. Decision Tree Classification Feature Importance

Feature	Importance
fnlwgt	0.200955
relationship	0.200154
education-num	0.125126
age	0.119426
capital-gain	0.112687

[4] sklearn AdaBoostClassifier
[5] sklearn RandomizedSearchCV

[6] [7] [8]

6.4.3 Hyperparameter Tuning: Subsequently, hyperparameter tuning was performed to understand how each chosen parameter impacts the model performance. 'criterion,' 'splitter,' 'max_depth,' 'min_samples_split,' and 'min_samples_leaf.' were the parameters that were used. These parameters have been selected as they are essential for the decision tree's behavior, particularly when it pertains to over-fitting, impurity assessment, and granularity of splits.

'criterion': The metric, usually 'entropy' for information gain or 'gini' for Gini impurity, that is employed to evaluate the quality of a split in a decision tree.

'splitter': The method for choosing the most advantageous characteristic to divide the data into two categories: 'random' (chosen at random) or 'best' (chosen based on a quality metric).

'max_depth': The decision tree's maximum growth depth, which restricts its complexity and overfitting potential.

'min_samples_split': The bare minimum of samples necessary to divide a node in a tree-building operation.

'min_samples_leaf': This parameter controls the granularity of splits by indicating the minimum amount of samples needed to form a leaf node in the decision tree.

For hyperparameter tuning, we used Random Search because it enables us to explore a wide range of hyperparameter combinations within a shorter timeframe, which is particularly useful when computational resources are limited. In Random Search, the technique randomly samples various combinations of hyperparameter values to discover the optimal set for a machine learning model.

At the time of writing/running, the best hyperparameters obtained were:

Best Hyperparameters:
{ 'splitter': 'best', 'min_samples_split': 3, 'min_samples_leaf': 5, 'max_depth': 7, 'criterion': 'gini' }
[9]

6.4.4 Custom Implementations. The from-scratch implementation of the `DecisionTreeClassifier_custom` class in Python provides a way to create a decision tree for classification tasks.

Initialization: The class is initialized using parameters such as `minimum_sample_split`, `max_tree_depth`, and `criterion`, which allow the user to select the tree's stopping circumstances as well as the splitting criterion (Gini, Entropy, or the best of both).

Building the Tree: The `build_decision_tree` method is a recursive function that constructs the decision tree by determining the best split at each node till the stopping conditions (minimum samples or maximum depth) are reached.

Split Criterion: The `get_best_split` method iterates through each feature and its potential thresholds to identify the split with the largest gain (Gini gain, Information gain, or the best of both), as indicated by the `criterion` setting.

Splitting the Data: The `split` function separates the dataset into two subsets (left and right) depending on the threshold of a given feature, enabling the decision tree's binary splitting nature.

Gain Calculation: Methods for computing Information Gain (`information_gain`) and Gini Gain (`gini_gain`) are included in the class. Depending on the criterion, it utilizes one of these measures to evaluate the effectiveness of each proposed split.

Calculation of Entropy and Gini Index: The methods `entropy` and `gini_index` are utilized to calculate the corresponding metrics, which are required for the Information Gain and Gini Gain calculations.

Determining Leaf Nodes: In the case of classification, the `calculate_leaf_node_value` method finds the value of a leaf node, which is often the most prevalent class label among the samples that reach the leaf.

Model Training and Prediction: The `fit` approach trains the model on the provided dataset, while the `predict` method makes predictions on new data using the learned tree. The `make_prediction` method explores the tree recursively for each data point until it reaches a leaf node and returns its value as the prediction.

Flexibility and Customizability: This custom solution allows for depth selection/control, minimum samples per node, and split criterion selection, allowing it to be adapted to different datasets and classification challenges.

6.5 Cross-Validation

The implementation of K-Fold Cross-Validation using a Decision Tree classifier involves splitting the validation dataset into five folds, ensuring randomness and reputability. For each fold, the model is trained on one subset of the data and evaluated on another, resulting in five sets of evaluation metrics. These metrics encompass accuracy, precision, recall, and F1-score, providing a comprehensive assessment of the model's performance. The Decision Tree model is configured with specific parameters, including the minimum sample split, maximum tree depth, and the chosen splitting criterion. A robust assessment of the model's overall effectiveness is obtained by averaging the metrics across all folds, mitigating the influence of any single data split. This K-Fold Cross-Validation approach plays a crucial role in guiding model selection and facilitating hyper-parameter tuning, as it provides a more reliable estimate of the model's performance.

[6] sklearn Tree
[7] Decision Tree Algorithm
[8] Decision Tree
[9] sklearn Decision Tree

7 Results

Present the results of each data mining technique, comparing performance metrics and discussing the findings.

7.1 Random Forest Algorithm

Table 3. Evaluation Comparison of Adaptive Boosting

Model	Precision	Recall	Accuracy	F1 score
Baseline Model	71.75%	63.64%	85.30%	68%
Hyperparameter Tuned	76.43%	56.62%	85.43%	67%
RF Custom model (from scratch)	80.01%	54.5%	84.88%	51.20%
K fold (Custom from scratch)	67.45%	65.05%	84.55%	60.21%

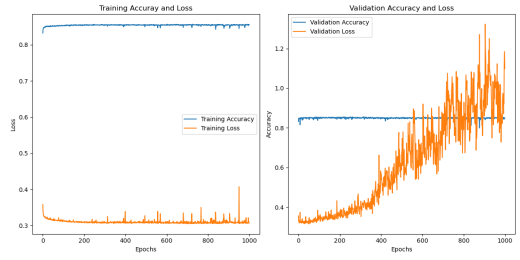

Figure 2. Train and Validation Loss and Accuracy for ANN Library Implementation

7.2 Artificial Neural Network

The outcomes presented encapsulate the performance metrics of diverse models assessed on the provided dataset. The initial model, serving as a baseline, achieved an accuracy rate of 85.06%, along with a recall of 58.85%, precision of 73.49%, and an F1 score of 65.36%. Following optimization, the model experienced a marginal accuracy increase to 85.30%, accompanied by a trade-off in recall (51.60%) and a substantial enhancement in precision (79.86%). Both the K-Fold cross-validated models, utilizing both the library's standard method and a custom approach, demonstrated consistent performance. The library-based K-Fold attained an accuracy of 84.96%, a recall of 59.39%, a precision of 73.27%, and an F1 score of 65.29%. Similarly, the custom K-Fold exhibited an accuracy of 85.40%, recall of 58.06%, precision of 75.33%, and an F1 score of 65.56%. These findings indicate that the custom implementation, while maintaining a competitive accuracy level, excels in precision when compared to both the baseline and library-based models, showcasing its potential in applications where precision holds particular significance. The observed variations in recall values across models underscore the inherent trade-offs that can be navigated based on the specific objectives and demands of the classification task. In summary, these results furnish a holistic comprehension of each model's strengths and weaknesses, facilitating informed decision-making for practical applications involving model selection.

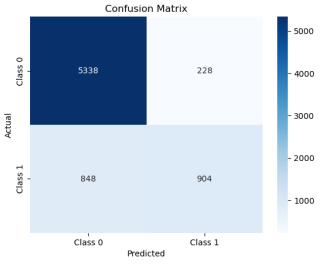

Figure 3. Confusion for ANN Library Implementation

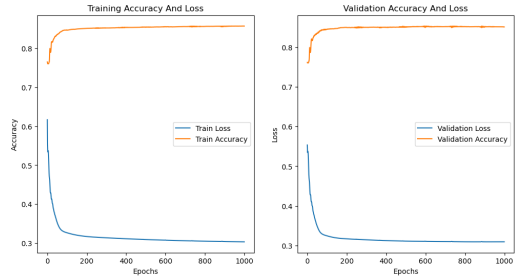

Figure 4. Train and Validation Loss and Accuracy for ANN Custom Implementation

7.3 Adaptive Boosting Algorithm

Table 4. Evaluation Comparison of Artificial Neural Network

Model	Accuracy	Recall	Precision	F1 score
Baseline	0.8506	0.5885	0.7349	0.6536
Optimized	0.8530	0.5160	0.7986	0.6269
K-Fold Lib	0.8496	0.5939	0.7327	0.6529
Custom	0.8522	0.5867	0.7422	0.6554
K-Fold Custom	0.8540	0.5806	0.7533	0.6556

Table 5. Evaluation Comparison of Adaptive Boosting

Model	Precision	Recall	Accuracy	F1 score
Baseline	76%	61%	86%	68%
Optimized	75%	61%	86%	67%
K-Fold Lib	-	-	86%	-
Custom	47%	9%	75%	16%
K-Fold Custom	49%	54%	58%	34%

7.4 Decision Tree Algorithm

Table 6. Evaluation Comparison of Decision Tree Algorithm

Model	Accuracy	Recall	Precision	F1 score
Baseline	0.8136	0.6364	0.6053	0.6205
Optimized	0.8593	0.5616	0.7897	0.6564
K-Fold Lib	0.8574	0.7793	0.5662	0.6559
Custom	0.8390	0.5594	0.7071	0.6246
K-Fold Custom	0.8395	0.4735	0.7667	0.5854

8 Conclusion and Future Work

In conclusion, this data science project on census income effectively demonstrates the power and versatility of machine learning algorithms in predictive analytics. By employing a range of models – AdaBoost, Random Forest (RF), Decision Tree (DT), and Artificial Neural Network (ANN) – the study offers a comprehensive approach to understanding income demographics. Each model contributes unique insights due to its inherent characteristics, such as AdaBoost's emphasis on iterative improvement, RF's robustness through ensemble learning, DT's interpretability, and ANN's capability to model complex, non-linear relationships. The project not only provides accurate income predictions but also deepens our understanding of the socioeconomic factors influencing income levels. The comparative analysis of these models showcases the importance of choosing the right algorithm based on the specific requirements and nature of the dataset, ultimately guiding future research and practical applications in the field of socio-economic data analysis.

References

[1] L. Billard. 2017. Study of Salary Differentials by Gender and Discipline. *Statistics and Public Policy* (2017). https://doi.org/10.1080/2330443X.2017.1317223

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[3] Gilles Louppe, Louis Wehenkel, and Antonio Sutera. 2013. Understanding random forests: From theory to practice. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems (NIPS)*. 1132–1140.

[4] Bhuvana Chandra M and N. Deepa. 2022. Improving Performance Analysis in Classification with Accuracy of Adult Income Salary using Novel Gated Residual Neural Network in Comparison with Logistic Regression Algorithm. In *2022 International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems (ICSES)*. https://doi.org/10.1109/ICSES55317.2022.9914103

[5] J. Ross Quinlan. 1986. Induction of Decision Trees. *Machine Learning* (1986).