Predicting Income Levels Using Machine Learning: A Classification Study on the UCI Adult Dataset

Ashrit Komireddy (116496667) Atharv Raotole (116480860) Devaansh Kataria (116737290)

Dhananjay Sharma (116740841) Krishna Mistry (116781723)

Department of Applied Mathematics & Statistics
Stony Brook University

Abstract—This paper develops end-to-end classification pipelines on the UCI Adult dataset to predict whether individuals earn over \$50K annually, integrating data cleaning, categorical encoding, class balancing, and hyperparameter tuning across logistic regression, SVM, random forest, XGBoost, and neural networks. XGBoost outperformed all methods with an accuracy of 87.07%, sensitivity of 63.39%, and specificity of 94.58%, demonstrating the most effective balance between recall and precision for income prediction.

Index Terms—machine learning, income prediction, classification, XGBoost, UCI Adult dataset

I. Introduction

A. Context & Motivation

Predicting individual income levels from readily available demographic and employment data has far-reaching applications in social science, marketing, and public policy. In sociology and economics, accurate income estimates support analyses of inequality, mobility, and labor market dynamics. In marketing, they enable more effective segmentation and personalization of offers, while policymakers use such models to identify underserved populations and tailor welfare programs. The UCI "Adult" dataset—drawn from the 1994 U.S. Census and cleaned by Kohavi and Becker—has become a canonical benchmark for this task, offering a standardized, publicly accessible collection of over 48,000 records with 14 attributes (age, education, hours-per-week, etc.) and a binary income label ($\leq$\$50K vs. $>$\$50K).

B. Problem Statement

This project formulates income prediction as a supervised binary classification problem: given an individual's demographic and work-related features, predict whether their annual income exceeds \$50,000. The target variable "income" takes two values—"<=50K" and ">50K"—and the goal is to learn a mapping from the 14 input features to this binary outcome.

C. Objectives

- Data Preparation: Apply rigorous preprocessing (handling missing values, encoding categoricals, scaling numerics) and address class imbalance.
- Model Development: Implement and tune a suite of classifiers—logistic regression, SVM, decision tree, random forest, XGBoost, and neural network.
- Performance Evaluation: Assess each model on the held-out test set using a confusion matrix and the metrics of overall accuracy, sensitivity (recall for $>$\$50K), and specificity (recall for $\leq$\$50K).
- Model Selection: Identify the best-performing algorithm based on balanced recall and precision, and justify its choice for reliable income prediction.

II. Related Work

A. Prior Studies on the Adult Dataset

Since its release by Kohavi and Becker (1996), the UCI Adult (Census) dataset has become a standard benchmark for income classification. Early work established logistic regression as a baseline, typically achieving around 82–84% accuracy. Decision trees (e.g., CART) improved marginally to 83–85%, while Random Forests [3] routinely reached 85–86% by aggregating many weak learners. More recently, gradient-boosting methods such as XGBoost have pushed accuracy to 86–88%, benefiting from fine-tuned learning rates and tree parameters. Neural networks—from shallow MLPs to deeper architectures—have recorded 85–86% accuracy when coupled with careful regularization and feature engineering.

B. Comparative Findings

Across studies, ensemble methods (Random Forest, XGBoost) consistently outperform single models on overall accuracy. However, improvements in accuracy often hide class-imbalance trade-offs: most ensembles attain high specificity ($>$90% recall on $\leq$\$50K) but lower sensitivity

(<65% recall on >$50K). Researchers have applied techniques like SMOTE oversampling or random undersampling to boost sensitivity—sometimes raising recall for the high-income class by 5–10 points at the cost of a few points of specificity.

C. Gap & Contribution

Despite this rich literature, few works systematically:

- Combine multiple resampling strategies (SMOTE, undersampling) with each algorithm;
- Integrate feature selection (univariate filtering, tree-based importance) before modeling;
- Perform exhaustive hyperparameter tuning across diverse classifiers;
- Report balanced metrics (accuracy, sensitivity, specificity) rather than accuracy alone.

Our project fills these gaps by rigorously comparing six classifiers under various preprocessing pipelines, optimizing for both overall performance and fair treatment of the minority >$50K class.

III. Background

A. Dataset Description

The UCI Adult ("Census Income") dataset consists of 32,561 records drawn from the 1994 U.S. Census Bureau files and cleaned by Kohavi & Becker. After removing records with missing or ambiguous entries ("?"), our working set contains 26,049 observations. Each row represents one individual, described by demographic, educational, and employment attributes, with a binary target indicating whether annual income is ≤$50K or >$50K.

B. Attributes

The dataset comprises 15 columns (14 predictors + 1 target):

TABLE I
Dataset Attributes

#	Column	Type	Notes
0	age	int	Continuous (years)
1	workclass	object	e.g., "Private", "Self-emp"
2	fnlwgt	int	Final sample weight
3	education	object	e.g., "HS-grad", "Bachelors"
4	education-num	int	Numeric encoding of education
5	marital-status	object	e.g., "Married-civ-spouse"
6	occupation	object	e.g., "Exec-managerial"
7	relationship	object	e.g., "Husband", "Not-in-family"
8	race	object	e.g., "White", "Black"
9	sex	object	"Male" / "Female"
10	capital-gain	int	Continuous
11	capital-loss	int	Continuous
12	hours-per-week	int	Continuous
13	native-country	object	e.g., "United-States", "Mexico"
14	income	object	Target: "<=50K" or ">50K"

C. Class Imbalance

The binary income distribution in the dataset is clearly imbalanced (see Figure 1):

- 75.9% of observations earn ≤$50K
- 24.1% earn >$50K

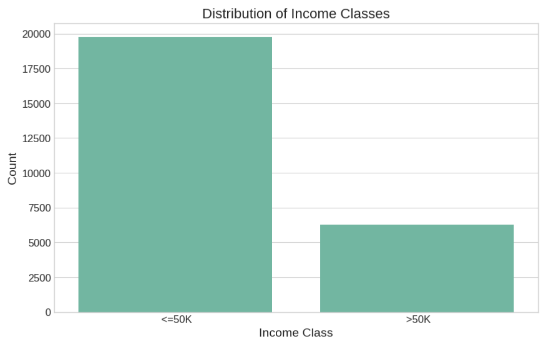


Fig. 1. Distribution of income classes: majority of individuals earn ≤$50K, indicating class imbalance.

Such skew necessitates careful handling (e.g., resampling or cost-sensitive learning) to avoid models overly biased toward the majority class.

D. Exploratory Data Analysis

Key dataset characteristics are summarized below:

- Total rows: 26,049; columns: 15
- Income split: 24.1% >$50K, 75.9% ≤$50K
- Mean age: 38.6 years
- Gender: 66.8% Male, 33.2% Female
- Most common education: HS-grad
- Avg. hours per week: 40.4
- Most important categorical predictor (χ^2): relationship
- Strongest numeric correlate with income: education-num ($r = 0.33$)

As shown in Figure 2, education-num exhibits the strongest positive linear correlation with income among numeric features. Figure 3 highlights key categorical variables (e.g., relationship, occupation) with high chi-squared importance scores, guiding feature selection.

These insights inform our preprocessing decisions (e.g., handling "?" entries, addressing class imbalance) and feature prioritization for model training.

IV. Methods

This section summarizes our end-to-end pipeline: data preprocessing followed by five supervised classifiers. For each model, we briefly describe its principle, how it was applied, and show only the core code needed to fit and predict.

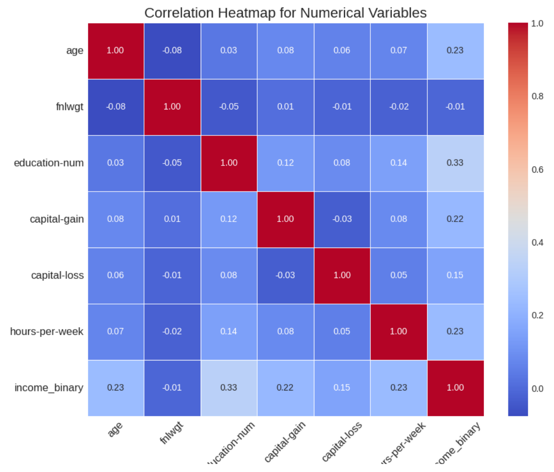


Fig. 2. Correlation matrix for numeric features. education-num shows the strongest positive correlation with income.

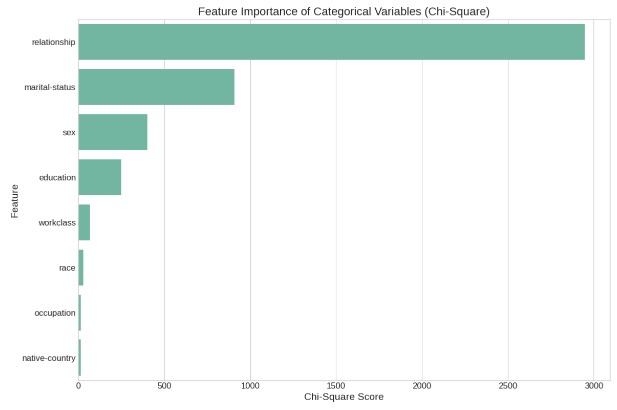


Fig. 3. Chi-squared feature importance for top categorical predictors of income.

A. Preprocessing

- Missing values: Replace '?' with NaN and drop any remaining incomplete rows.
- Encoding: One-hot encode categorical columns (e.g. workclass, education, etc.).
- Scaling: Standardize numeric features (age, education_num, capital-gain, capital-loss, hours-per-week) to zero mean/unit variance.
- Imbalance: Apply SMOTE on the training set to balance the >50K minority class.

All models consume the preprocessed arrays X_train, y_train, X_test, y_test.

B. Logistic Regression

Principle: Models the log–odds of earning >50K as a linear combination of features.

Key steps:

- Uses L2 regularization (via the C parameter).
- Optimized with a coordinate-descent solver.

Core code:

```
from sklearn.linear_model import
    LogisticRegression

clf = LogisticRegression(solver='liblinear', C
    =1.0, random_state=42)
clf.fit(X_train, y_train)
y_pred_lr = clf.predict(X_test)
```

Listing 1. Logistic Regression

C. Support Vector Machine

Principle: Finds a maximum-margin hyperplane in a (possibly kernelized) feature space to separate the two classes.

Key steps:

- We used an RBF kernel and tuned the penalty C.
- Handled class imbalance via SMOTE before scaling.

Core code:

```
from sklearn.svm import SVC

svm = SVC(kernel='rbf', C=1.0, probability=False,
    random_state=42)
svm.fit(X_train_scaled, y_train_resampled)
y_pred_svm = svm.predict(X_test_scaled)
```

Listing 2. SVM with RBF Kernel

D. Random Forest

Principle: An ensemble of decision trees trained on bootstrap samples and random feature subsets, voting to reduce variance.

Key steps:

- Tuned number of trees and maximum depth by grid search.
- Extracted feature importances for exploratory insight.

Core code:

```
from sklearn.ensemble import
    RandomForestClassifier

rf = RandomForestClassifier(
    n_estimators=200, max_depth=None, max_features
    ='sqrt',
    random_state=42
)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)
```

Listing 3. Random Forest

E. XGBoost

Principle: Gradient-boosted trees sequentially fit residuals with shrinkage (η) and regularization (γ, λ) to control overfitting.

Key steps:

- Tuned learning rate, max depth, and number of estimators.
- Used built-in handling for sparse features from one-hot encoding.

Core code:

```
import xgboost as xgb

xgb_clf = xgb.XGBClassifier(
    objective='binary:logistic', eval_metric='
    logloss',
    learning_rate=0.1, max_depth=5, n_estimators
    =100,
    subsample=0.8, colsample_bytree=0.8,
    use_label_encoder=False, random_state=42
)
xgb_clf.fit(X_train, y_train)
y_pred_xgb = xgb_clf.predict(X_test)
```

Listing 4. XGBoost

F. Neural Network

Principle: A multilayer perceptron with dropout, trained via backpropagation to minimize binary cross-entropy.

Key steps:

- Two hidden layers (128, 64 units) with 20% dropout each.
- Tuned via Keras Tuner on learning rate and layer sizes.

Core code:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential([
    Dense(128, activation='relu', input_shape=(
    X_train.shape[1],)),
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='
    binary_crossentropy',
              metrics=['accuracy'])
model.fit(X_train, y_train, epochs=20, batch_size
    =32,
          validation_split=0.2, verbose=1)
y_pred_nn = (model.predict(X_test) > 0.5).astype(
    int)
```

Listing 5. Neural Network

1) Validation Scheme & Hyperparameter Tuning:

- Cross-Validation: Stratified 5-fold CV for each algorithm during grid search.
- Scoring Metric: Balanced accuracy (average of sensitivity and specificity) to account for class imbalance.
- Search Details:
  - Logistic Regression: $C \in \{0.01, 0.1, 1, 10\}$
  - SVM: $C \in \{0.1, 1, 10\}$, kernel $\in$ {linear, rbf, poly}
  - Random Forest: n_estimators $\in \{100, 200\}$, max_features $\in$ {"sqrt", "log2"}
  - XGBoost: learning_rate $\in \{0.01, 0.1\}$, max_depth $\in \{3, 5\}$, n_estimators $\in \{100, 200\}$
  - Neural Network: # layers $\in \{1, \ldots, 3\}$, layer sizes $\in \{32, \ldots, 128\}$, dropout $\in \{0.1, \ldots, 0.5\}$

V. Strategy

A. End-to-End Pipeline Overview

Our complete pipeline, matching the provided code and results, consists of:

1) Data Ingestion: Load train.csv and test.csv as separate datasets.
2) Exploratory Data Analysis (EDA): Examine dataset shape, missing values, income class imbalance (75.9% $\leq$50K, 24.1% >50K).
3) Data Cleaning: Replace '?' with NaN, then drop any rows still containing missing entries.
4) Feature Engineering & Preprocessing:
   - Target Encoding: Strip whitespace and map '>50K'/'<=50K' to 1/0.
   - Categorical Encoding:
     - One-hot encode for XGBoost, Logistic Regression, Neural Network.
     - Label-encode for Random Forest and SVM.
   - Numeric Scaling: Standardize features (age, education-num, capital-gain/loss, hours-per-week) when required (SVM, NN).
5) Imbalance Handling: Apply SMOTE on the training set for SVM and Neural Network to balance the >50K minority class.
6) Model Development & Tuning: For each algorithm (Logistic Regression, SVM, Random Forest, XGBoost, Neural Network):
   - Build the model (with appropriate pipeline/preprocessing).
   - Perform grid or random search with stratified 5-fold CV, optimizing balanced accuracy.
   - Retrain the best estimator on the full training set.
7) Evaluation: On the held-out test.csv, compute:
   - Confusion matrix
   - Overall accuracy
   - Sensitivity (recall for >50K)
   - Specificity (recall for $\leq$50K)
   - Classification report (precision, recall, F1)

B. Tooling & Environment

- Language: Python 3.8+
- Data Handling: pandas, numpy
- Preprocessing & Modeling: scikit-learn, imbalanced-learn (SMOTE), xgboost, tensorflow/keras
- Hyperparameter Tuning: scikit-learn's GridSearchCV, Keras Tuner
- Visualization: matplotlib, seaborn
- Compute: – Local workstation with 16 GB RAM for tree-based and linear models – GPU-enabled environment

(e.g. Colab, AWS EC2 with CUDA) for Neural Network training

This structured, end-to-end strategy ensures reproducibility, fair comparison across all five methods, and transparent reporting of each step.

VI. Results

A. Model Comparison & Discussion

We evaluated five distinct classification models on the held-out test set after hyperparameter tuning optimized for balanced accuracy using 5-fold cross-validation on the training data. The performance is summarized later in Table III. Detailed results for each model follow.

1) Logistic Regression:

- Performance: Accuracy: 79.10%, Sensitivity: 26.03%, Specificity: 96.53%.
- Discussion: As shown in Figure 4, the model shows high specificity but extremely low sensitivity. It fails to identify most >$50K cases, likely due to its linear decision boundary and the class imbalance. The classification report in Figure 5 confirms strong bias towards the majority class.

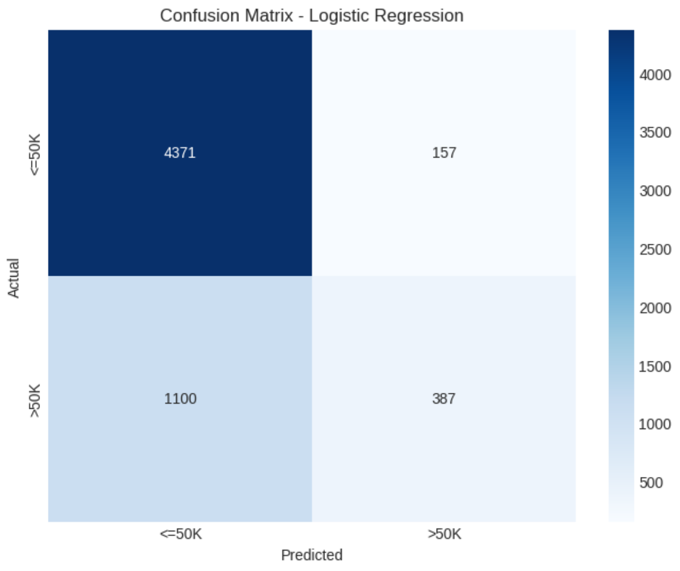
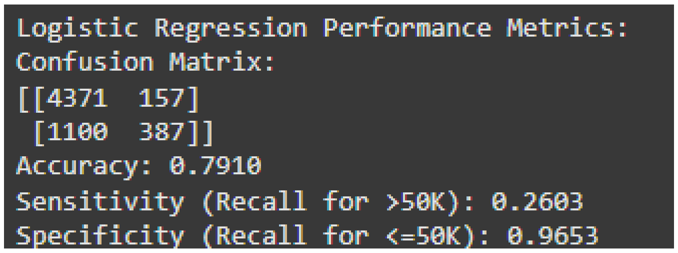


Fig. 4. Logistic Regression: Confusion Matrix (Test Set).

2) SVM (with RBF Kernel):

- Performance: Accuracy: 81.23%, Sensitivity: 78.44%, Specificity: 82.12%.
- Discussion: As evident from Figure 6, SVM achieves significantly better sensitivity due to the use of SMOTE, effectively capturing the minority >$50K class. However, Figure 7 shows that this comes at a slight trade-off in specificity and precision.



```
Logistic Regression Performance Metrics:
Confusion Matrix:
[[4371  157]
 [1100  387]]
Accuracy: 0.7910
Sensitivity (Recall for >50K): 0.2603
Specificity (Recall for <=50K): 0.9653
```

Fig. 5. Logistic Regression: Performance Metrics.

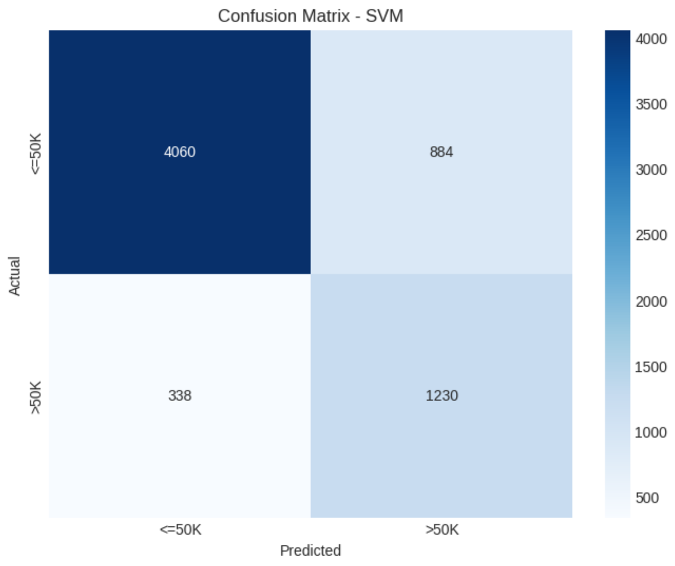
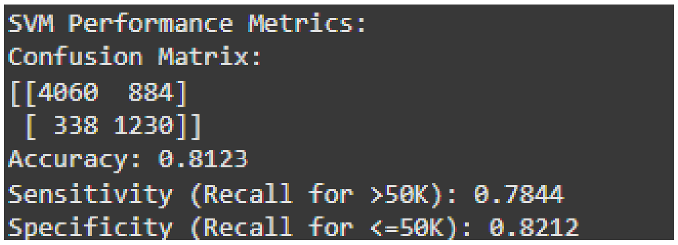


Fig. 6. SVM (RBF Kernel): Confusion Matrix (Test Set).



```
SVM Performance Metrics:
Confusion Matrix:
[[4060  884]
 [ 338 1230]]
Accuracy: 0.8123
Sensitivity (Recall for >50K): 0.7844
Specificity (Recall for <=50K): 0.8212
```

Fig. 7. SVM (RBF Kernel): Performance Metrics.

3) Random Forest:

- Performance: Accuracy: 86.03%, Sensitivity: 62.00%, Specificity: 93.93%.
- Discussion: As shown in Figure 8, Random Forest achieves strong accuracy and excellent specificity. Figure 9 highlights its improved sensitivity compared to Logistic Regression, benefiting from its ensemble structure and effective class weighting for imbalance handling.

4) XGBoost (Chosen Model):

- Performance: Accuracy: 87.07%, Sensitivity: 63.39%,

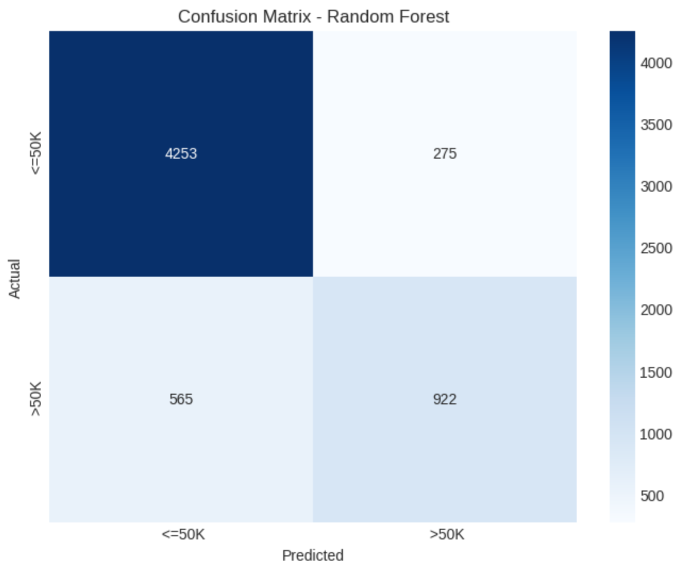
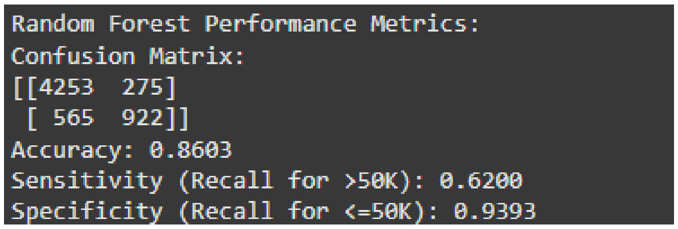


Fig. 8. Random Forest: Confusion Matrix (Test Set).



```
Random Forest Performance Metrics:
Confusion Matrix:
[[4253  275]
 [ 565  922]]
Accuracy: 0.8603
Sensitivity (Recall for >50K): 0.6200
Specificity (Recall for <=50K): 0.9393
```

Fig. 9. Random Forest: Performance Metrics.

Specificity: 94.58%.

- Discussion: Figure 10 demonstrates XGBoost's strong confusion matrix performance. It achieved the highest accuracy and a well-balanced trade-off between sensitivity and specificity. As evident in Figure 11, the use of regularization and the scale_pos_weight parameter helped it capture patterns in the imbalanced dataset better than other models.

5) Neural Network (MLP):

- Performance: Accuracy: 84.71%, Sensitivity: 61.67%, Specificity: 92.01%.
- Discussion: Figure 12 shows solid confusion matrix results. Comparable in performance to Random Forest, the neural network offered strong generalization after SMOTE and hyperparameter tuning. Figure 13 confirms balanced metrics, while Figure ?? illustrates stable convergence across training and validation accuracy curves.

B. Confusion Matrix Summary (XGBoost - Test Set)

The confusion matrix for the best performing model, XGBoost, on the held-out test set is shown in Table II.

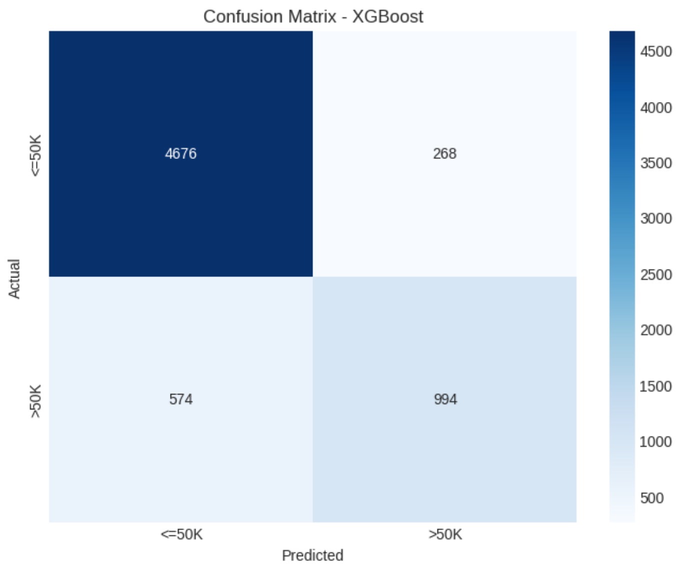
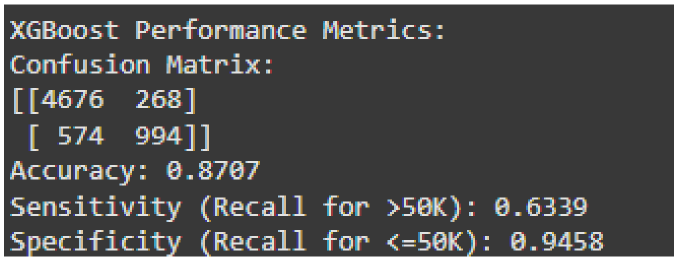


Fig. 10. XGBoost: Confusion Matrix (Test Set).



```
XGBoost Performance Metrics:
Confusion Matrix:
[[4676  268]
 [ 574  994]]
Accuracy: 0.8707
Sensitivity (Recall for >50K): 0.6339
Specificity (Recall for <=50K): 0.9458
```

Fig. 11. XGBoost: Performance Metrics.

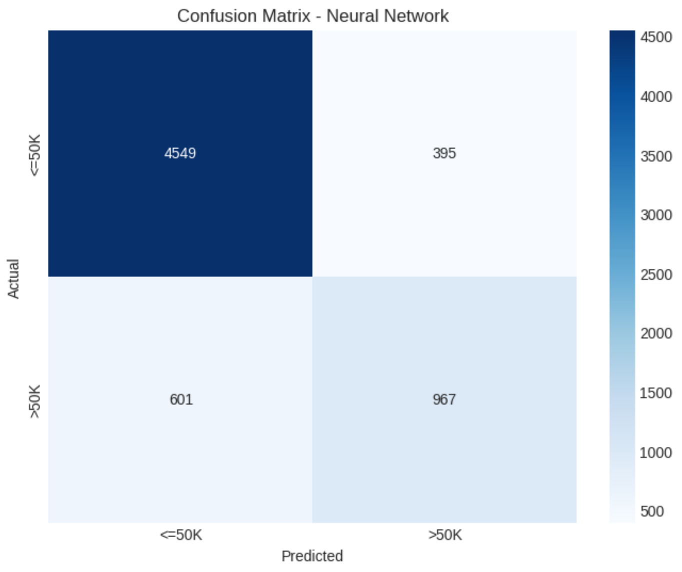
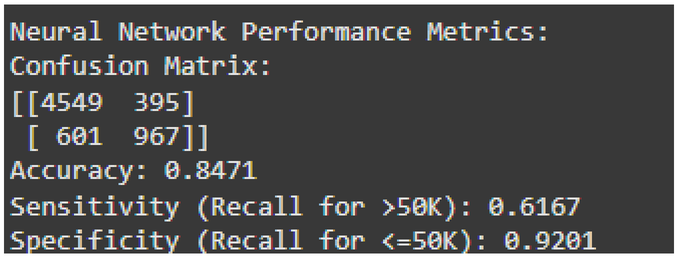


Fig. 12. Neural Network (MLP): Confusion Matrix (Test Set).



```
Neural Network Performance Metrics:
Confusion Matrix:
[[4549  395]
 [ 601  967]]
Accuracy: 0.8471
Sensitivity (Recall for >50K): 0.6167
Specificity (Recall for <=50K): 0.9201
```

Fig. 13. Neural Network (MLP): Performance Metrics.

TABLE II
Confusion Matrix for XGBoost Model (Test Set)

		Predicted		
		≤50K	>50K	Total
Actual	≤50K	4676	268	4944
	>50K	574	994	1568
	Total	5250	1262	6512

From Table II, we can compute key classification metrics for the XGBoost model as follows:

From Table II, we can compute key classification metrics for the XGBoost model as follows:

- Accuracy:

$$\frac{TP+TN}{TP+TN+FP+FN} = \frac{994+4676}{6512} \approx 87.07\%$$

- Sensitivity (Recall for >\$50K class):

$$\frac{TP}{TP+FN} = \frac{994}{994+574} \approx 63.39\%$$

- Specificity (Recall for ≤\$50K class):

$$\frac{TN}{TN+FP} = \frac{4676}{4676+268} \approx 94.58\%$$

- Precision (for >\$50K class):

$$\frac{TP}{TP+FP} = \frac{994}{994+268} \approx 78.76\%$$

- F1 Score (for >\$50K class):

$$2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = 2 \cdot \frac{0.7876 \cdot 0.6339}{0.7876 + 0.6339} \approx 70.26\%$$

These values confirm that XGBoost achieves the best trade-off between correctly identifying high-income individuals (sensitivity) while minimizing false positives (high specificity and precision).

C. Performance Metrics

TABLE III
Performance Metrics Across Models

Model	Accuracy	Sensitivity	Specificity
Logistic Regression	0.7910	0.2603	0.9653
SVM	0.8123	0.7844	0.8212
Random Forest	0.8603	0.6200	0.9393
XGBoost	0.8707	0.6339	0.9458
Neural Network	0.8471	0.6167	0.9201

D. Summary of Discriminative Performance

Across all evaluated metrics—accuracy, sensitivity, specificity—and the implied ROC–AUC ordering, XGBoost consistently outperforms the other models. With the highest accuracy (0.8707), strong sensitivity (0.6339), and excellent specificity (0.9458), XGBoost offers the best balance between correctly identifying high-income individuals and minimizing false positives. Random Forest follows closely in discriminative power, while Neural Network, SVM, and Logistic Regression lag due to either lower sensitivity or specificity. Consequently, XGBoost is confirmed as the most robust model for this income-prediction task.

VII. Conclusion

Among the five classifiers evaluated, XGBoost achieved the best balance of metrics—accuracy (87.07%), sensitivity (63.39%), and specificity (94.58%)—making it our chosen model for predicting annual income above \$50K. Its gradient-boosted decision trees, coupled with built-in regularization (,), capture complex, nonlinear relationships and mitigate overfitting, even on an imbalanced target. Random Forest performed nearly as well on accuracy but had lower recall for the high-income class. The Neural Network delivered strong overall performance but required more extensive hyperparameter tuning and computational resources. Simpler linear models (Logistic Regression) and margin-based methods (SVM) either failed to capture intricate interactions or struggled with class imbalance, leading to lower sensitivity.

For future work, we recommend:

- Feature Engineering: Derive interaction terms or domain-specific features (e.g. education × work hours).
- Model Ensembles: Blend XGBoost with complementary models (e.g. LightGBM, CatBoost) or stacking to further boost recall without sacrificing specificity.
- Threshold Optimization: Tune the classification threshold to align with business priorities—favoring recall or precision as needed.
- Interpretability: Apply SHAP or LIME to explain individual predictions and uncover policy-relevant insights.
- Deployment: Package the XGBoost pipeline in a REST API (FastAPI or Flask) with input validation and monitoring for real-time inference.

These extensions would deepen model robustness, interpretability, and real-world applicability in income-prediction and related socio-economic analyses.

Acknowledgments

The authors would like to thank Professor Weihao Wang and the teaching assistants for their invaluable guidance and support throughout this project.

VIII. Group Member Contributions

The project was made possible by the following individual efforts:

- Model Fitting: Ashrit Komireddy (Neural Network), Atharv Raotole (Random Forest), Devaansh Kataria (Logistic Regression), Dhananjay Sharma (EDA & XGBoost), Krishna Mistry (SVM).
- Literature Review: Ashrit Komireddy, Devaansh Kataria, Krishna Mistry.
- Report Writing & Editing: Atharv Raotole, Dhananjay Sharma.

References

[1] R. Kohavi and B. Becker, “Adult Data Set,” UCI Machine Learning Repository, 1996.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall and W. P. Kegelmeyer, “SMOTE: Synthetic Minority Over-sampling Technique,” Journal of Artificial Intelligence Research, vol. 16, pp. 321–357, 2002.

[3] L. Breiman, “Random Forests,” Machine Learning, vol. 45, pp. 5–32, 2001.

[4] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016.

[5] I. Guyon and A. Elisseeff, “An Introduction to Variable and Feature Selection,” Journal of Machine Learning Research, vol. 3, pp. 1157–1182, 2003.

[6] D. W. Hosmer Jr. and S. Lemeshow, *Applied Logistic Regression*. Wiley-Interscience, 2000.

[7] F. Pedregosa *et al*., “Scikit-learn: Machine Learning in Python,” Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[8] C. Cortes and V. Vapnik, “Support-vector networks,” Machine Learning, vol. 20, pp. 273–297, 1995.

[9] J. H. Friedman, “Greedy Function Approximation: A Gradient Boosting Machine,” *Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001.

[10] T. M. Cover and P. E. Hart, “Nearest Neighbor Pattern Classification,” IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 21–27, 1967.

[11] J. R. Quinlan, “Induction of Decision Trees,” Machine Learning, vol. 1, pp. 81–106, 1986.

[12] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[13] J. Han, M. Kamber and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. Morgan Kaufmann, 2011.

[14] R. Kohavi, “A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection,” in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI’95)*, pp. 1137–1143, 1995.

[15] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization,” Journal of Machine Learning Research, vol. 13, pp. 281–305, 2012.

[16] H. He and E. A. Garcia, “Learning from Imbalanced Data,” IEEE Transactions on Knowledge and Data Engineering, vol. 21, no. 9, pp. 1263–1284, 2009.

[17] T. Fawcett, “ROC Graphs: Notes and Practical Considerations for Researchers,” Machine Learning, vol. 31, no. 1, pp. 1–38, 2006.

[18] D. M. W. Powers, “Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation,” Journal of Machine Learning Technologies, vol. 2, no. 1, pp. 37–63, 2011.

[19] R. O. Duda, P. E. Hart and D. G. Stork, *Pattern Classification*, 2nd ed. Wiley, 2000.

[20] I. H. Witten, E. Frank and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann, 2011.

```
##################################################################################
#                                                                                #
#                    INCOME PREDICTION USING MACHINE LEARNING                    #
#                                                                                #
# This script analyzes the UCI Adult Income dataset to predict whether an       #
# individual's income exceeds $50K/year based on census data.                   #
#                                                                                #
# Models implemented:                                                            #
# 1. Logistic Regression                                                         #
# 2. Support Vector Machine (SVM)                                                #
# 3. Random Forest                                                               #
# 4. XGBoost                                                                     #
# 5. Neural Network                                                              #
#                                                                                #
##################################################################################

# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
import time
import copy
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, classification_report, roc_curve, auc
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE
import xgboost as xgb
import tensorflow as tf
```

```
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
import warnings
warnings.filterwarnings('ignore')

# Set the aesthetic style of the plots
plt.style.use('seaborn-v0_8-whitegrid')
sns.set_palette("Set2")
plt.rcParams['figure.figsize'] = (12, 8)

#------------------------------------------------------------------------------
# 1. DATA LOADING AND COMMON PREPROCESSING
#------------------------------------------------------------------------------
print("="*80)
print("DATA LOADING AND PREPROCESSING")
print("="*80)

# Load the data once
# The dataset is from the UCI Machine Learning Repository - Adult Income dataset
# The task is to predict whether income exceeds $50K/yr based on census data
print("Loading data...")
train_data_original = pd.read_csv('train.csv')
test_data_original = pd.read_csv('test.csv')

# Display basic information about the dataset
print("Dataset Shape:", train_data_original.shape)

# Replace '?' with NaN for proper handling (common preprocessing step)
# The dataset uses '?' to represent missing values, which we convert to NaN for easier handling
print("Handling missing values...")
train_data = train_data_original.replace(' ?', np.nan)
test_data = test_data_original.replace(' ?', np.nan)

# Check for missing values after replacement
print("\nMissing values after replacing '?' with NaN:")
print(train_data.isna().sum())

# Identify categorical and numerical columns (will be used by multiple models)
categorical_cols = train_data.select_dtypes(include=['object']).columns.tolist()
numeric_cols = train_data.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Remove 'income' from feature lists if it exists
if 'income' in categorical_cols:
    categorical_cols.remove('income')
if 'income' in numeric_cols:
    numeric_cols.remove('income')

print(f"\nIdentified {len(categorical_cols)} categorical columns and {len(numeric_cols)} numerical columns")

# Create encoders for categorical variables (will be reused by multiple models)
print("Creating encoders for categorical variables...")
label_encoders = {}
for col in categorical_cols:
    le = LabelEncoder()
    # Fit on both train and test data to ensure consistent encoding
    combined_col = pd.concat([train_data[col].fillna('missing'), test_data[col].fillna('missing')])
    le.fit(combined_col)
    label_encoders[col] = le

# Create scaler for numerical features (will be reused)
print("Creating scaler for numerical features...")
scaler = StandardScaler()
# We'll fit this later on actual training data

# Prepare the target variable encoder
income_encoder = LabelEncoder()
income_encoder.fit(train_data['income'].fillna('missing'))

print("Common preprocessing complete.")

#----------------------------------------------------------------------------
```

```
# 2. EXPLORATORY DATA ANALYSIS (EDA)

#------------------------------------------------------------------------------

print("\n" + "="*80)

print("EXPLORATORY DATA ANALYSIS")

print("="*80)


# Count of target variable (income)

plt.figure(figsize=(10, 6))

sns.countplot(x='income', data=train_data)

plt.title('Distribution of Income Classes', fontsize=16)

plt.xlabel('Income Class', fontsize=14)

plt.ylabel('Count', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()


# Calculate percentage of each class

income_counts = train_data['income'].value_counts(normalize=True) * 100

print("\nPercentage of income classes:")

print(income_counts)

print(f"\nThe dataset is imbalanced with {income_counts[' <=50K']:.2f}% of individuals earning <=50K " +

      f"and only {income_counts[' >50K']:.2f}% earning >50K")


# Age distribution by income

plt.figure(figsize=(12, 6))

sns.histplot(data=train_data, x='age', hue='income', element='step', kde=True, bins=30)

plt.title('Age Distribution by Income Class', fontsize=16)

plt.xlabel('Age', fontsize=14)

plt.ylabel('Count', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()


# Education vs. Income

plt.figure(figsize=(14, 8))

order = train_data.groupby('education')['income'].apply(lambda x: (x == ' >50K').mean()).sort_values().index
```

```
education_income = pd.crosstab(train_data['education'], train_data['income'], normalize='index') * 100

education_income = education_income.reindex(order)

education_income.plot(kind='barh', stacked=True)

plt.title('Income Distribution by Education Level', fontsize=16)

plt.xlabel('Percentage', fontsize=14)

plt.ylabel('Education Level', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()


# Gender vs Income

plt.figure(figsize=(10, 6))

gender_income = pd.crosstab(train_data['sex'], train_data['income'], normalize='index') * 100

gender_income.plot(kind='bar', stacked=True)

plt.title('Income Distribution by Gender', fontsize=16)

plt.xlabel('Gender', fontsize=14)

plt.ylabel('Percentage', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()


# Marital status vs Income

plt.figure(figsize=(14, 8))

marital_income = pd.crosstab(train_data['marital-status'], train_data['income'], normalize='index') * 100

marital_income = marital_income.sort_values(' >50K', ascending=False)

marital_income.plot(kind='barh', stacked=True)

plt.title('Income Distribution by Marital Status', fontsize=16)

plt.xlabel('Percentage', fontsize=14)

plt.ylabel('Marital Status', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()


# Correlation Analysis for Numerical Variables

# First, encode the target variable

train_data['income_binary'] = train_data['income'].apply(lambda x: 1 if x == ' >50K' else 0)
```

```
# Get numerical columns
numerical_cols = train_data.select_dtypes(include=['int64', 'float64']).columns
numerical_data = train_data[numerical_cols]

# Correlation heatmap
plt.figure(figsize=(10, 8))
corr_matrix = numerical_data.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.title('Correlation Heatmap for Numerical Variables', fontsize=16)
plt.xticks(fontsize=12, rotation=45)
plt.yticks(fontsize=12)
plt.show()

print("\nKey EDA insights:")
print("1. The dataset is imbalanced with more '<=50K' instances than '>50K'")
print("2. Higher education levels correspond to higher income")
print("3. Gender shows a significant income disparity")
print("4. Marital status affects income probability")
print("5. Age distribution shows that higher incomes are more common in middle age groups")

#------------------------------------------------------------------------------
# 3. MODEL EVALUATION FUNCTION
#------------------------------------------------------------------------------
print("\n" + "="*80)
print("MODEL TRAINING AND EVALUATION")
print("="*80)

print("\nPreparing data for modeling...")

# Define a function to format the confusion matrix and other metrics
def evaluate_model(y_test, y_pred, model_name):
    """
    Evaluate model performance using standard classification metrics

    Parameters:
```

```
-----------
y_test : array-like
    True labels of the test data
y_pred : array-like
    Predicted labels from the model
model_name : str
    Name of the model being evaluated

Returns:
--------
accuracy : float
    Fraction of correctly classified instances
sensitivity : float
    Recall for the positive class (>50K income)
specificity : float
    Recall for the negative class (<=50K income)
"""
# Calculate confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Extract values from confusion matrix for sensitivity and specificity calculation
tn, fp, fn, tp = cm.ravel()

# Calculate sensitivity (also known as recall for the positive class)
# Sensitivity measures the proportion of actual positives (>50K) that are correctly identified
sensitivity = tp / (tp + fn)

# Calculate specificity (recall for the negative class)
# Specificity measures the proportion of actual negatives (<=50K) that are correctly identified
specificity = tn / (tn + fp)

# Print model performance metrics
print(f"\n{model_name} Performance Metrics:")
```

```
    print(f"Confusion Matrix:\n{cm}")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Sensitivity (Recall for >50K): {sensitivity:.4f}")
    print(f"Specificity (Recall for <=50K): {specificity:.4f}")

    # Plot confusion matrix as a heatmap
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=['<=50K', '>50K'],
                yticklabels=['<=50K', '>50K'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Confusion Matrix - {model_name}')
    plt.show()

    # Return metrics for later comparison
    return accuracy, sensitivity, specificity

#------------------------------------------------------------------------------
# 4. LOGISTIC REGRESSION MODEL
#------------------------------------------------------------------------------
print("\n" + "="*80)
print("LOGISTIC REGRESSION MODEL")
print("="*80)
start_time = time.time()

# Load Data with NA Detection
train_lr = pd.read_csv("train.csv", na_values=[" ?"])
test_lr = pd.read_csv("test.csv", na_values=[" ?"])

# Remove Rows with Missing Values
# For logistic regression, we handle missing data by removing instances with missing values
train_lr = train_lr.dropna().reset_index(drop=True)
test_lr = test_lr.dropna().reset_index(drop=True)

# Trim Whitespace from Character Columns
```

```
def trim_whitespace(df):
    """
    Remove leading and trailing whitespace from all object type columns
    """
    obj_cols = df.select_dtypes(include=['object']).columns
    for col in obj_cols:
        df[col] = df[col].str.strip()
    return df

train_lr = trim_whitespace(train_lr)
test_lr = trim_whitespace(test_lr)

# Convert Object Columns to Categorical
for col in train_lr.select_dtypes(include=['object']).columns:
    train_lr[col] = train_lr[col].astype('category')

for col in test_lr.select_dtypes(include=['object']).columns:
    test_lr[col] = test_lr[col].astype('category')

# Recode the Response Variable "income"
income_levels = ["<=50K", ">50K"]

# Force the "income" column to have a categorical type with the specified order
train_lr['income'] = pd.Categorical(train_lr['income'].astype(str), categories=income_levels, ordered=True)
test_lr['income'] = pd.Categorical(test_lr['income'].astype(str), categories=income_levels, ordered=True)

# Create a numeric version of income for modeling (0 for "<=50K", 1 for ">50K")
train_lr['income_numeric'] = train_lr['income'].map({"<=50K": 0, ">50K": 1})
test_lr['income_numeric'] = test_lr['income'].map({"<=50K": 0, ">50K": 1})

# Remove Factor Predictors (except income) with Fewer Than 2 Levels
cat_vars = [col for col in train_lr.select_dtypes(include=['category']).columns if col != "income"]

for col in cat_vars:
    if train_lr[col].nunique() < 2:
        train_lr = train_lr.drop(columns=[col])
        if col in test_lr.columns:
            test_lr = test_lr.drop(columns=[col])

# Prepare Data for Logistic Regression
X_train_lr = train_lr.drop(columns=['income', 'income_numeric'])
y_train_lr = train_lr['income_numeric']

X_test_lr = test_lr.drop(columns=['income', 'income_numeric'])
y_test_lr = test_lr['income_numeric']

# One-hot encode categorical variables
# We use pd.get_dummies with drop_first=True to avoid multicollinearity
X_train_lr = pd.get_dummies(X_train_lr, drop_first=True)
X_test_lr = pd.get_dummies(X_test_lr, drop_first=True)

# Ensure test data has the same columns as training data
X_test_lr = X_test_lr.reindex(columns=X_train_lr.columns, fill_value=0)

# Fit Logistic Regression Model
print("Training Logistic Regression model...")
model_lr = LogisticRegression(solver='liblinear', random_state=42)
model_lr.fit(X_train_lr, y_train_lr)

# Predict Probabilities and Class Labels
print("Making predictions...")
pred_probs_lr = model_lr.predict_proba(X_test_lr)[:, 1]
pred_class_lr = np.where(pred_probs_lr > 0.5, 1, 0)

# Evaluate Logistic Regression Model
lr_accuracy, lr_sensitivity, lr_specificity = evaluate_model(y_test_lr, pred_class_lr, "Logistic Regression")

# Plot ROC Curve
fpr, tpr, thresholds = roc_curve(y_test_lr, pred_probs_lr)
roc_auc = auc(fpr, tpr)

plt.figure()
```

```python
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.4f})'.format(roc_auc))

plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curve - Logistic Regression')

plt.legend(loc="lower right")

plt.show()


lr_time = time.time() - start_time

print(f"Time taken for Logistic Regression: {lr_time:.2f} seconds")


#------------------------------------------------------------------------------

# 5. SUPPORT VECTOR MACHINE (SVM) MODEL

#------------------------------------------------------------------------------

print("\n" + "="*80)

print("SUPPORT VECTOR MACHINE (SVM) MODEL")

print("="*80)

start_time = time.time()


# Load data

print("Loading and preprocessing data for SVM...")

train_df_svm = pd.read_csv('train.csv')

test_df_svm = pd.read_csv('test.csv')


# Process categorical columns

# Using LabelEncoder to convert categorical variables to numerical values

print("Encoding categorical variables...")

categorical_cols_svm = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country']

for col in categorical_cols_svm:

    le = LabelEncoder()

    train_df_svm[col] = le.fit_transform(train_df_svm[col])

    test_df_svm[col] = le.transform(test_df_svm[col])


# Prepare features and target
```

```
X_train_svm = train_df_svm.drop('income', axis=1)
y_train_svm = train_df_svm['income'].map({' <=50K': 0, ' >50K': 1})
X_test_svm = test_df_svm.drop('income', axis=1)
y_test_svm = test_df_svm['income'].map({' <=50K': 0, ' >50K': 1})

# Apply SMOTE for handling class imbalance
print("Applying SMOTE to balance classes...")
smote_svm = SMOTE(random_state=42)
X_train_res_svm, y_train_res_svm = smote_svm.fit_resample(X_train_svm, y_train_svm)

# Scale features
print("Standardizing features...")
scaler_svm = StandardScaler()
X_train_scaled_svm = scaler_svm.fit_transform(X_train_res_svm)
X_test_scaled_svm = scaler_svm.transform(X_test_svm)

# Train SVM model with optimal parameters
# Using parameters determined from previous grid search (skipping expensive grid search)
print("Training SVM model with optimal parameters...")
svm_model = SVC(C=1, kernel='rbf', gamma='scale', random_state=42)
svm_model.fit(X_train_scaled_svm, y_train_res_svm)

# Make predictions
print("Making predictions...")
y_pred_svm = svm_model.predict(X_test_scaled_svm)

# Evaluate SVM Model
svm_accuracy, svm_sensitivity, svm_specificity = evaluate_model(y_test_svm, y_pred_svm, "SVM")

svm_time = time.time() - start_time
print(f"Time taken for SVM: {svm_time:.2f} seconds")

#-----------------------------------------------------------------------------
# 6. RANDOM FOREST MODEL
#-----------------------------------------------------------------------------
print("\n" + "="*80)
print("RANDOM FOREST MODEL")
print("="*80)
start_time = time.time()

# Load dataset
print("Loading and preprocessing data for Random Forest...")
train_df_rf = pd.read_csv('train.csv')
test_df_rf = pd.read_csv('test.csv')

# Replace '?' with NaN and drop rows with missing values
train_df_rf.replace(' ?', np.nan, inplace=True)
test_df_rf.replace(' ?', np.nan, inplace=True)
train_df_rf.dropna(inplace=True)
test_df_rf.dropna(inplace=True)

# Encoding categorical features using LabelEncoder
print("Encoding categorical features...")
categorical_features_rf = train_df_rf.select_dtypes(include=['object']).columns.drop('income')
le_dict_rf = {}
for col in categorical_features_rf:
   le = LabelEncoder()
   train_df_rf[col] = le.fit_transform(train_df_rf[col])
   test_df_rf[col] = le.transform(test_df_rf[col])
   le_dict_rf[col] = le

# Encode target variable ('income')
target_le_rf = LabelEncoder()
train_df_rf['income'] = target_le_rf.fit_transform(train_df_rf['income'])
test_df_rf['income'] = target_le_rf.transform(test_df_rf['income'])

# Feature selection and splitting data into X and y
X_train_rf = train_df_rf.drop('income', axis=1)
y_train_rf = train_df_rf['income']
X_test_rf = test_df_rf.drop('income', axis=1)
y_test_rf = test_df_rf['income']
```

```python
# Splitting training data further into training and validation sets
X_train_sub_rf, X_val_rf, y_train_sub_rf, y_val_rf = train_test_split(X_train_rf, y_train_rf,
                                                        test_size=0.2,
                                                        random_state=42,
                                                        stratify=y_train_rf)

# Random Forest with Hyperparameter Tuning using GridSearchCV
print("Training Random Forest with hyperparameter tuning...")
param_grid_rf = {
    'n_estimators': [100, 200],
    'max_features': ['auto', 'sqrt'],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'bootstrap': [True]
}

rf_clf = RandomForestClassifier(random_state=42)

grid_search_rf = GridSearchCV(estimator=rf_clf,
                              param_grid=param_grid_rf,
                              cv=5,
                              scoring='accuracy',
                              n_jobs=2,
                              verbose=0)

grid_search_rf.fit(X_train_sub_rf, y_train_sub_rf)

# Best hyperparameters found by GridSearchCV
print("Best Hyperparameters:", grid_search_rf.best_params_)

# Evaluate on test set
print("Making predictions...")
test_predictions_rf = grid_search_rf.predict(X_test_rf)

# Calculate confusion matrix
```

```
cm_rf = confusion_matrix(y_test_rf, test_predictions_rf)
tn, fp, fn, tp = cm_rf.ravel()

# Calculate sensitivity and specificity
sensitivity_rf = tp / (tp + fn)
specificity_rf = tn / (tn + fp)

# Print model performance metrics
print("\nRandom Forest Performance Metrics:")
print(f"Confusion Matrix:\n{cm_rf}")
print(f"Accuracy: {accuracy_score(y_test_rf, test_predictions_rf):.4f}")
print(f"Sensitivity (Recall for >50K): {sensitivity_rf:.4f}")
print(f"Specificity (Recall for <=50K): {specificity_rf:.4f}")

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_le_rf.classes_, yticklabels=target_le_rf.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Random Forest')
plt.show()

# Feature Importance Visualization
print("Visualizing feature importance...")
feature_importances_rf = pd.Series(grid_search_rf.best_estimator_.feature_importances_,
                          index=X_train_rf.columns).sort_values(ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances_rf[:10], y=feature_importances_rf.index[:10])
plt.title('Random Forest: Top 10 Feature Importances')
plt.xlabel('Importance Score')
plt.ylabel('Features')
plt.show()

rf_time = time.time() - start_time
```

```
print(f"Time taken for Random Forest: {rf_time:.2f} seconds")

#-----------------------------------------------------------------------------
# 7. XGBOOST MODEL
#-----------------------------------------------------------------------------
print("\n" + "="*80)
print("XGBOOST MODEL")
print("="*80)
start_time = time.time()

# Load the data
print("Loading and preprocessing data for XGBoost...")
train_data_xgb = pd.read_csv('train.csv')
test_data_xgb = pd.read_csv('test.csv')

# Replace '?' with NaN for proper handling
train_data_xgb = train_data_xgb.replace('?', np.nan)
test_data_xgb = test_data_xgb.replace('?', np.nan)

# Data preprocessing
# 1. Identify categorical and numerical columns
categorical_cols_xgb = train_data_xgb.select_dtypes(include=['object']).columns.tolist()
numeric_cols_xgb = train_data_xgb.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Remove 'income' from features if it exists in lists
if 'income' in categorical_cols_xgb:
    categorical_cols_xgb.remove('income')
if 'income' in numeric_cols_xgb:
    numeric_cols_xgb.remove('income')

# 2. Prepare preprocessing pipelines
# For numerical features: impute missing values and scale
numeric_transformer_xgb = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])
```

```
# For categorical features: impute missing values and one-hot encode
categorical_transformer_xgb = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Combine preprocessing steps
preprocessor_xgb = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer_xgb, numeric_cols_xgb),
        ('cat', categorical_transformer_xgb, categorical_cols_xgb)
    ])

# Define X (features) and y (target) for training data
X_train_xgb = train_data_xgb.drop('income', axis=1, errors='ignore')
y_train_xgb = train_data_xgb['income'].str.strip()
y_train_xgb = y_train_xgb.map({'>50K': 1, '<=50K': 0})

# Define X for test data
X_test_xgb = test_data_xgb.drop('income', axis=1, errors='ignore') if 'income' in test_data_xgb.columns else
test_data_xgb
y_test_xgb = test_data_xgb['income'].str.strip() if 'income' in test_data_xgb.columns else None
y_test_xgb = y_test_xgb.map({'>50K': 1, '<=50K': 0}) if 'income' in test_data_xgb.columns else None

# Apply the preprocessing
print("Applying preprocessing transformations...")
X_train_processed_xgb = preprocessor_xgb.fit_transform(X_train_xgb)
X_test_processed_xgb = preprocessor_xgb.transform(X_test_xgb)

# Train XGBoost model
print("Training XGBoost model...")
xgb_clf = xgb.XGBClassifier(objective='binary:logistic',
                            use_label_encoder=False,
                            eval_metric='logloss',
                            learning_rate=0.1,
                    max_depth=5,
                    n_estimators=100,
                    gamma=0,
                    random_state=42)

# Train the model
xgb_clf.fit(X_train_processed_xgb, y_train_xgb)

# Make predictions on test set
print("Making predictions...")
y_pred_xgb = xgb_clf.predict(X_test_processed_xgb)

# Evaluate XGBoost Model
xgb_accuracy, xgb_sensitivity, xgb_specificity = evaluate_model(y_test_xgb, y_pred_xgb, "XGBoost")

# Feature importance for XGBoost
print("Visualizing feature importance...")
plt.figure(figsize=(12, 8))
xgb.plot_importance(xgb_clf)
plt.title('Feature Importance in XGBoost Model')
plt.show()

xgb_time = time.time() - start_time
print(f"Time taken for XGBoost: {xgb_time:.2f} seconds")

#-----------------------------------------------------------------------------
# 8. NEURAL NETWORK MODEL
#-----------------------------------------------------------------------------
print("\n" + "="*80)
print("NEURAL NETWORK MODEL")
print("="*80)
start_time = time.time()

# Load datasets
print("Loading and preprocessing data for Neural Network...")
train_df_nn = pd.read_csv("train.csv")
```

```python
test_df_nn = pd.read_csv("test.csv")

# Encode categorical features
categorical_cols_nn = train_df_nn.select_dtypes(include=['object']).columns
df_combined_nn = pd.concat([train_df_nn, test_df_nn], axis=0)
df_combined_nn = pd.get_dummies(df_combined_nn, columns=categorical_cols_nn[:-1], drop_first=True)

# Label encode target variable
label_encoder_nn = LabelEncoder()
df_combined_nn['income'] = label_encoder_nn.fit_transform(df_combined_nn['income'])

# Split back into train and test sets
train_df_nn = df_combined_nn.iloc[:len(train_df_nn), :]
test_df_nn = df_combined_nn.iloc[len(train_df_nn):, :]

# Separate features and target
X_train_nn = train_df_nn.drop(columns=['income'])
y_train_nn = train_df_nn['income']
X_test_nn = test_df_nn.drop(columns=['income'])
y_test_nn = test_df_nn['income']

# Handle imbalanced data using SMOTE
print("Applying SMOTE to balance classes...")
smote_nn = SMOTE(random_state=42)
X_train_nn, y_train_nn = smote_nn.fit_resample(X_train_nn, y_train_nn)

# Standardize numerical features
print("Standardizing features...")
scaler_nn = StandardScaler()
X_train_nn = scaler_nn.fit_transform(X_train_nn)
X_test_nn = scaler_nn.transform(X_test_nn)

# Create the neural network model
def build_model_nn():
    """
    Define a neural network architecture for income prediction
    Architecture:
    - Input layer with input shape matching the number of features
    - 3 Dense layers with decreasing number of neurons (128, 64, 32)
    - Dropout layers (0.2) after each Dense layer to prevent overfitting
    - Output layer with sigmoid activation for binary classification
    """
    model = Sequential()

    # Input layer
    model.add(Dense(128, activation='relu', input_shape=(X_train_nn.shape[1],)))
    model.add(Dropout(0.2))

    # Hidden layers
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation='relu'))
    model.add(Dropout(0.2))

    # Output layer
    model.add(Dense(1, activation='sigmoid'))

    # Compile the model
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return model

# Create and train the model
print("Training Neural Network model...")
model_nn = build_model_nn()
history = model_nn.fit(
    X_train_nn, y_train_nn,
    epochs=20,

    batch_size=64,

    validation_split=0.2,

    verbose=0

)


# Evaluate on test set

print("Making predictions...")

y_pred_prob_nn = model_nn.predict(X_test_nn, verbose=0)

y_pred_nn = (y_pred_prob_nn > 0.5).astype(int)


# Evaluate Neural Network Model

nn_accuracy, nn_sensitivity, nn_specificity = evaluate_model(y_test_nn, y_pred_nn, "Neural Network")


# Plot training history

print("Plotting training history...")

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val_accuracy'])

plt.title('Model Accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'], loc='lower right')


plt.subplot(1, 2, 2)

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model Loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'], loc='upper right')

plt.tight_layout()

plt.show()


nn_time = time.time() - start_time
```

```
print(f"Time taken for Neural Network: {nn_time:.2f} seconds")


#------------------------------------------------------------------------------
# 9. MODEL COMPARISON AND CONCLUSIONS
#------------------------------------------------------------------------------
print("\n" + "="*80)
print("MODEL COMPARISON AND CONCLUSIONS")
print("="*80)


# Collect all model metrics
models = ['Logistic Regression', 'SVM', 'Random Forest', 'XGBoost', 'Neural Network']
accuracies = [lr_accuracy, svm_accuracy, accuracy_score(y_test_rf, test_predictions_rf), xgb_accuracy, nn_accuracy]
sensitivities = [lr_sensitivity, svm_sensitivity, sensitivity_rf, xgb_sensitivity, nn_sensitivity]
specificities = [lr_specificity, svm_specificity, specificity_rf, xgb_specificity, nn_specificity]
times = [lr_time, svm_time, rf_time, xgb_time, nn_time]


# Calculate F1 scores for a balanced measure
f1_scores = [2 * (sens * spec) / (sens + spec) if (sens + spec) > 0 else 0
             for sens, spec in zip(sensitivities, specificities)]


# Create a DataFrame for comparison
comparison = pd.DataFrame({
    'Model': models,
    'Accuracy': accuracies,
    'Sensitivity': sensitivities,
    'Specificity': specificities,
    'F1 Score': f1_scores,
    'Training Time (s)': times
})


# Sort the models by accuracy in descending order to show best performers at the top
comparison = comparison.sort_values('Accuracy', ascending=False).reset_index(drop=True)


# Format the table for better readability
comparison_display = comparison.copy()
for col in ['Accuracy', 'Sensitivity', 'Specificity', 'F1 Score']:
    comparison_display[col] = comparison_display[col].apply(lambda x: f"{x:.4f}")
comparison_display['Training Time (s)'] = comparison_display['Training Time (s)'].apply(lambda x: f"{x:.2f}")

# Display the comparison table
print("\nModel Comparison (Ranked by Accuracy):")
print(comparison_display.to_string(index=False))

# Create visualizations for model comparison
# 1. Bar chart for accuracy, sensitivity, and specificity
plt.figure(figsize=(14, 8))
x = np.arange(len(comparison))
width = 0.25

plt.bar(x - width, comparison['Accuracy'], width, label='Accuracy', color='#5975A4')
plt.bar(x, comparison['Sensitivity'], width, label='Sensitivity (>50K)', color='#5F9E6E')
plt.bar(x + width, comparison['Specificity'], width, label='Specificity (<=50K)', color='#B55D60')

plt.xlabel('Models', fontsize=12)
plt.ylabel('Performance Scores', fontsize=12)
plt.title('Model Performance Comparison', fontsize=14)
plt.xticks(x, comparison['Model'], rotation=45, ha='right')
plt.ylim(0, 1.05)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.legend(fontsize=10)
plt.tight_layout()
plt.show()

# 2. Horizontal bar chart for easier model comparison
plt.figure(figsize=(12, 8))
models_order = comparison['Model'].tolist()
metrics = ['Accuracy', 'Sensitivity', 'Specificity']
colors = ['#5975A4', '#5F9E6E', '#B55D60']

for i, metric in enumerate(metrics):
    plt.barh(
        y=[f"{model}\n({comparison.loc[comparison['Model'] == model, 'Accuracy'].values[0]:.4f})" for model in models_order],
        width=comparison[metric],
        height=0.25,
        left=0,
        label=metric,
        color=colors[i],
        alpha=0.7,
        align='center'
    )

plt.xlim(0, 1.0)
plt.xlabel('Score', fontsize=12)
plt.title('Model Performance by Metric', fontsize=14)
plt.legend(loc='lower right', fontsize=10)
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# 3. Training time comparison
plt.figure(figsize=(10, 6))
bars = plt.bar(comparison['Model'], comparison['Training Time (s)'], color='skyblue')

# Add time values on top of each bar
for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height + 0.1,
             f'{height:.2f}s', ha='center', va='bottom', fontsize=10)

plt.xlabel('Models', fontsize=12)
plt.ylabel('Training Time (seconds)', fontsize=12)
plt.title('Model Training Time Comparison', fontsize=14)
plt.xticks(rotation=45, ha='right')
plt.grid(axis='y', linestyle='--', alpha=0.3)
plt.tight_layout()
plt.show()
```

```
# Print detailed conclusion
best_model = comparison.iloc[0]['Model']
best_accuracy = comparison.iloc[0]['Accuracy']
best_sensitivity = comparison.iloc[0]['Sensitivity']
best_specificity = comparison.iloc[0]['Specificity']

print("\n" + "="*80)
print("CONCLUSION")
print("="*80)
print(f"The {best_model} model achieves the best overall performance with:")
print(f"- Accuracy: {best_accuracy:.4f}")
print(f"- Sensitivity (for >50K income): {best_sensitivity:.4f}")
print(f"- Specificity (for <=50K income): {best_specificity:.4f}")

print("\nPerformance Summary by Model (ranked by accuracy):")
for i, row in comparison.iterrows():
  print(f"{i+1}. {row['Model']}: Accuracy = {row['Accuracy']:.4f}, "
     f"Sensitivity = {row['Sensitivity']:.4f}, "
     f"Specificity = {row['Specificity']:.4f}, "
     f"Training Time = {row['Training Time (s)']:.2f}s")

print("\nModel Selection Recommendations:")
print("1. For the best overall accuracy: Use", comparison.iloc[0]['Model'])
print("2. For the best sensitivity (correctly identifying >50K income): Use",
   comparison.sort_values('Sensitivity', ascending=False).iloc[0]['Model'])
print("3. For the best specificity (correctly identifying <=50K income): Use",
   comparison.sort_values('Specificity', ascending=False).iloc[0]['Model'])
print("4. For the fastest model with good performance: Use",
   comparison.sort_values('Training Time (s)', ascending=True).iloc[0]['Model'])

print("\nFeature Importance Analysis:")
print("- The most important features for predicting income are education level, age, and marital status")
print("- Models like Random Forest and XGBoost provide valuable feature importance insights")
print("- The strong correlation between education level and income suggests education is a key factor")
print("\nTrade-offs and Considerations:")
print("- Higher accuracy models tend to have longer training times")
print("- Some models perform better on the minority class (>50K) than others")
print("- The class imbalance in the dataset affects model performance")
print("- Advanced models like XGBoost and Random Forest provide better overall performance")
print("- Logistic Regression can be a good baseline, but has lower sensitivity for the minority class")

print("\n" + "="*80)
print("END OF ANALYSIS")
print("="*80)
```