CS 634 Data Mining

Final Term Project Report

Car Evaluation Dataset Analysis Using Random Forest, LSTM, and KNN

Github repository link: https://github.com/sm3585/maddineni_sindhu_final_project

1. Introduction

The objective of this project is to build and evaluate machine learning models to classify car evaluations based on a dataset containing features like car buying price, maintenance cost, number of doors, passenger capacity, luggage boot size, and safety ratings. The project leverages three distinct algorithms—Random Forest, LSTM (a deep learning model), and K-Nearest Neighbors (KNN)—to classify cars into categories based on their acceptability.

Dataset Description

The dataset used is the **Car Evaluation Dataset**, which consists of six attributes describing various aspects of a car, along with a target attribute indicating the car's acceptability. The target variable is initially multi-class (unacc, acc, good, vgood) but was transformed into a binary classification problem, mapping the classes unacc to 0 and the others (acc, good, vgood) to 1. This was done to simplify the classification task while preserving the essential patterns in the data.

- **Attributes:**
  - buying: Buying price of the car.
  - maint: Maintenance cost.
  - doors: Number of doors.
  - persons: Passenger capacity.
  - lug_boot: Luggage boot size.
  - safety: Safety rating.
- **Target Variable:**
  - class: Acceptability of the car, mapped as:
    - unacc → 0
    - acc, good, vgood → 1

The dataset was preprocessed to replace categorical values with numerical representations using one-hot encoding and to balance the target variable classes using the SMOTE technique. This ensured equal representation of both classes (0 and 1), improving the training and evaluation of the models.

Algorithms Implemented

Three algorithms were implemented to classify the cars and compare their performance:

1. **Random Forest (RF):**
    - An ensemble learning method using decision trees for classification.
    - Hyperparameter tuning was performed using GridSearchCV to optimize the model.
2. **LSTM (Long Short-Term Memory):**
    - A deep learning model designed for sequential data. Although not inherently necessary for tabular data, it was used to evaluate its performance on this task.
    - A single LSTM layer with 64 units was employed.
3. **K-Nearest Neighbors (KNN):**
    - A simple and interpretable algorithm that classifies instances based on the majority vote of its nearest neighbors.
    - Hyperparameter tuning was done to select the optimal number of neighbors.

Performance Metrics

To evaluate and compare the models, the following metrics were used:

- **Manually Computed Metrics:**
    - True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR), and False Negative Rate (FNR).
    - Precision, Accuracy, F1-Score, True Skill Statistic (TSS), and Heidke Skill Score (HSS).
- **Library-Based Metrics:**
    - Area Under the ROC Curve (AUC).
    - Brier Score.

The evaluation was performed using **10-fold cross-validation**, and the results were summarized in both tabular and graphical formats to facilitate comparison.

Project Significance

This project demonstrates the application of machine learning and deep learning techniques to a practical classification problem. By analyzing the results and comparing the algorithms, insights are drawn on the effectiveness of different approaches for this type of problem.

2. Environment Setup

This project was implemented using **Google Colab**, a free cloud-based Jupyter Notebook environment. Below are the steps to set up the environment and install the required packages.

2.1 Tools Used

- **Google Colab:** Provides a cloud-based Jupyter Notebook environment with free GPU/TPU support.

- **Python Version:** Python 3.10 (default in Google Colab).
- **Required Libraries:**
    - pandas: For data manipulation and preprocessing.
    - numpy: For numerical computations.
    - imblearn: For handling imbalanced datasets using SMOTE.
    - scikit-learn: For machine learning models, metrics, and cross-validation.
    - tensorflow: For building and training the LSTM deep learning model.
    - matplotlib: For visualizations such as ROC curves and metric comparisons.

2.2 Setting Up Google Colab

1. **Open Google Colab:**
    - Navigate to Google Colab in your browser.
    - Sign in with your Google account.
2. **Upload the Dataset:**
    - Save the car.data dataset file to your Google Drive.
    - Mount your Google Drive to access the dataset:

```
from google.colab import drive
drive.mount('/content/drive')
```

3. **Upload the Notebook:**
    - Open the provided Jupyter Notebook in Google Colab.

2.3 Installing Required Libraries

Google Colab already includes many essential libraries like numpy, pandas, and matplotlib. However, some libraries, like imblearn (for SMOTE), may need installation. To ensure all libraries are available, use the following commands in a notebook cell:

```
# Install imblearn for SMOTE
!pip install imbalanced-learn

# Ensure all other required libraries are installed
!pip install scikit-learn tensorflow matplotlib
```

2.4 Verifying Library Versions

After installing the required libraries, verify their versions to ensure compatibility:

```
import pandas as pd
import numpy as np
import sklearn
import tensorflow as tf
```

```
import matplotlib

print("Pandas Version:", pd.__version__)
print("NumPy Version:", np.__version__)
print("Scikit-learn Version:", sklearn.__version__)
print("TensorFlow Version:", tf.__version__)
print("Matplotlib Version:", matplotlib.__version__)
```

2.5 Loading and Preprocessing Data

The dataset car.data should be placed in the Colab workspace or accessed directly from Google Drive using the mounted path.

2.6 Running the Notebook

- Run all cells sequentially to preprocess the data, train models, and evaluate metrics. Ensure that the dataset file is accessible before running the notebook.

This environment setup ensures reproducibility of the project, leveraging the capabilities of Google Colab for seamless execution and visualization.

3. Data Preprocessing

3.1 Raw Data Overview

The initial dataset used in this project is the **Car Evaluation dataset**, which evaluates cars based on various attributes such as price, maintenance costs, safety, and more. The dataset contains categorical features, requiring preprocessing for machine learning algorithms.

The raw dataset was loaded as follows:

```
import pandas as pd

# Define the file path
file_path = "/content/car.data"

# Define column names for the dataset
columns = ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety',
'class']

# Load the data
car_data = pd.read_csv(file_path, header=None, names=columns)

# Display the head of the dataset
```

```
print(car_data.head())
```

Preview of the Raw Data:

```
  buying  maint doors persons lug_boot safety  class
0  vhigh  vhigh     2       2    small    low  unacc
1  vhigh  vhigh     2       2    small    med  unacc
2  vhigh  vhigh     2       2    small   high  unacc
3  vhigh  vhigh     2       2      med    low  unacc
4  vhigh  vhigh     2       2      med    med  unacc
```

3.2 Preprocessing Steps

Handling Categorical Values:

- The attributes buying, maint, lug_boot, and safety are categorical. These were converted to numerical representations using **One-Hot Encoding**.
- The doors and persons attributes contained values like "5more" and "more," which were replaced with integer equivalents (5).

Code snippet:

```
# Replace '5more' and 'more' in the dataset
car_data['doors'] = car_data['doors'].replace('5more', '5').astype(int)
car_data['persons'] = car_data['persons'].replace('more', '5').astype(int)

# One-hot encode categorical columns
car_data_encoded = pd.get_dummies(
    car_data,
    columns=['buying', 'maint', 'lug_boot', 'safety'],
    drop_first=True
)
```

Binary Classification Mapping:

- The target variable class originally had four categories: unacc, acc, good, and vgood.
- These were mapped into two categories for binary classification:
    - unacc: 0
    - acc, good, vgood: 1

```
# Binary classification mapping for 'class'
car_data_encoded['class'] = car_data_encoded['class'].replace(
    {'unacc': 0, 'acc': 1, 'good': 1, 'vgood': 1}
)
```

Balancing the Dataset:

- The dataset was imbalanced, with the majority class being unacc. To address this, **Synthetic Minority Oversampling Technique (SMOTE)** was applied.
- SMOTE generated synthetic examples for the minority class to balance the class distribution

```
from imblearn.over_sampling import SMOTE
# Separate features and target
X = car_data_encoded.drop(columns=['class'])
y = car_data_encoded['class']

# Apply SMOTE to balance the classes
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Verify the new class distribution
print("Class distribution after SMOTE:")
print(pd.Series(y_resampled).value_counts())
```

Class Distribution After SMOTE:

- Class 0: 1209
- Class 1: 1209

Combining Resampled Data:

- The resampled features and target were combined into a single DataFrame for further processing.

Code snippet:

```
final_data = pd.DataFrame(X_resampled, columns=X.columns)
final_data['class'] = y_resampled

# Display the first few rows of the final dataset
print(final_data.head(2))
```

Preview of the Final Processed Data:

```
   doors  persons  buying_low  buying_med  buying_vhigh  maint_low  maint_med  \
0      2        2       False       False          True      False      False
1      2        2       False       False          True      False      False

   maint_vhigh  lug_boot_med  lug_boot_small  safety_low  safety_med  class
0         True         False            True        True       False      0
1         True         False            True       False        True      0
```

3.3 Train-Test Split and Scaling

- The preprocessed data was split into training and testing sets using a **stratified split** to preserve the class distribution.
- The features were scaled using **StandardScaler** for better performance with machine learning algorithms.

Code snippet:

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Train-test split with stratification
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.1, stratify=y, random_state=42
)

# Standardizing the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert back to DataFrame for consistency in feature names
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X.columns)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X.columns)

print("Train-Test Split and Scaling Complete.")
print(f"Train Shape: {X_train_scaled.shape}, Test Shape:
{X_test_scaled.shape}")
```

Train-Test Split Summary:

- Training Set: 2178 samples
- Testing Set: 242 samples

Key Outcomes:

1. All categorical features were appropriately encoded.
2. The class imbalance was addressed using SMOTE, ensuring an equal representation of both classes.
3. The data was split into training and testing sets with stratification and scaled to improve model performance.

This preprocessing ensured the dataset was well-prepared for model training and evaluation.

4. Algorithms and Model Implementation

This section outlines the implementation of three machine learning algorithms: **Random Forest**, **LSTM**, and **KNN**. Each algorithm was carefully tuned, trained, and evaluated using cross-validation, with key performance metrics calculated for comparative analysis.

4.1 Random Forest

Hyperparameter Tuning

The Random Forest classifier was optimized using **GridSearchCV** to identify the best hyperparameters. The parameter grid included:

- n_estimators: Number of trees in the forest (50, 100, 150).
- min_samples_split: Minimum samples required to split a node (2, 5, 10).

The best parameters obtained were:

- {'n_estimators': 50, 'min_samples_split': 5}

Training

Using the optimized parameters, the Random Forest model was trained on the scaled training data (X_train_scaled and y_train). This ensured the model's structure was suited for the specific dataset.

Evaluation

The model was evaluated using **10-fold cross-validation**, and key metrics, including Accuracy, Precision, F1-Score, and AUC, were calculated for each fold. These metrics will be summarized and discussed in later sections.

4.2 LSTM

Data Preparation

To use LSTM, the input data was reshaped into a 3D format: (samples, timesteps, features) using the scaled feature set. The reshaped training data had dimensions of (2178, 12, 1).

Model Architecture

An LSTM model was built using TensorFlow's Keras library. The architecture included:

- A single LSTM layer with 64 units and a ReLU activation function.
- A dense output layer with a sigmoid activation function for binary classification.

The model configuration:

- LSTM(64, activation='relu') -> Dense(1, activation='sigmoid')

Training

The model was trained using the Adam optimizer for 50 epochs with a batch size of 32. The reshaped scaled data was used for training.

Evaluation

The model was evaluated using **10-fold cross-validation**, and metrics, including Accuracy, Precision, F1-Score, and AUC, were calculated for each fold. These metrics will be summarized and discussed in later sections.

4.3 KNN

Hyperparameter Tuning

The K-Nearest Neighbors (KNN) algorithm was optimized using **GridSearchCV**. The parameter grid focused on:

- n_neighbors: Number of neighbors to consider (1 to 15).

The best parameter obtained was:

- {'n_neighbors': 1}

Training

Using the best n_neighbors value, the KNN model was trained on the scaled training data (X_train_scaled and y_train). As a non-parametric model, KNN relies on distance calculations during prediction.

Evaluation

The model was evaluated using **10-fold cross-validation**, and metrics, including Accuracy, Precision, F1-Score, and AUC, were calculated for each fold. These metrics will be summarized and discussed in later sections.

5. Cross-Validation and Metrics Calculation

10-Fold Cross-Validation

To evaluate the performance of the models, we used 10-fold cross-validation with StratifiedKFold. This method ensures that each fold maintains the same class distribution as the original dataset. For each fold, the data is split into training and testing subsets, and the model is trained and evaluated on these subsets. The results from all folds are aggregated to calculate the average performance metrics, providing a robust evaluation.

Metrics Explanation

The following metrics were calculated for each fold to evaluate the models:

1. **True Positive Rate (TPR) / Recall**:
    - The proportion of actual positives that are correctly identified as positives.
    - TPR = TP / (TP + FN)
2. **True Negative Rate (TNR) / Specificity**:
    - The proportion of actual negatives that are correctly identified as negatives.
    - TNR = TN / (TN + FP)
3. **False Positive Rate (FPR)**:
    - The proportion of actual negatives that are incorrectly identified as positives.
    - FPR = FP / (FP + TN)
4. **False Negative Rate (FNR)**:
    - The proportion of actual positives that are incorrectly identified as negatives.
    - FNR = FN / (TP + FN)
5. **Precision**:
    - The proportion of predicted positives that are actually positive.
    - Precision = TP / (TP + FP)
6. **Accuracy**:
    - The proportion of total predictions (both positive and negative) that are correct.
    - Accuracy = (TP + TN) / (TP + TN + FP + FN)
7. **F1-Score**:
    - The harmonic mean of Precision and Recall, providing a balance between the two.
    - F1-Score = 2 * TP / (2 * TP + FP + FN)
8. **Area Under the Curve (AUC)**:
    - A measure of a model's ability to distinguish between classes, calculated using the ROC curve.
    - Higher AUC indicates better model performance.
9. **Brier Score**:

    - A metric to evaluate the accuracy of probabilistic predictions. It measures the mean squared difference between predicted probabilities and actual outcomes.
    - Lower Brier Score indicates better calibration of probabilistic predictions.
10. **True Skill Statistic (TSS)**:
    - A metric that evaluates the skill of the model in distinguishing between classes.
    - TSS = TPR - FPR
11. **Heidke Skill Score (HSS)**:
    - A metric that compares the model's performance to random guessing, accounting for imbalanced datasets.
    - HSS = 2 * (TP * TN - FP * FN) / ((TP + FN) * (FN + TN) + (TP + FP) * (FP + TN))

Visualizations

The following visualizations were generated to interpret and compare model performance:

1. **ROC Curves**:
    - ROC curves were plotted for each model to visualize their true positive rates (sensitivity) versus false positive rates. The area under each curve (AUC) was used to quantify performance.
2. **Bar Plots for Metric Comparison**:
    - Metrics like Accuracy, Precision, F1-Score, and AUC were compared across models using bar plots.

These metrics and visualizations provide a comprehensive understanding of how well each model performs, both in terms of classification accuracy and the ability to balance trade-offs between different metrics.

6. Results and Discussion

Performance Comparison Across Models

The performance of the three models—Random Forest, KNN, and LSTM—was evaluated using 10-fold cross-validation. Various metrics were calculated to assess the classification ability of each model comprehensively. Below is a detailed discussion based on the results.

Random Forest Metrics Across Folds:

	TPR (Recall)	TNR (Specificity)	FPR	FNR	Precision	F1-Score	Accuracy	TSS	HSS	AUC	Brier Score
Fold 1	1.000000	0.981651	0.018349	0.000000	0.981982	0.990909	0.990826	0.981651	0.981651	0.999411	0.011086
Fold 2	1.000000	0.972477	0.027523	0.000000	0.973214	0.986425	0.986239	0.972477	0.972477	0.998822	0.019116
Fold 3	0.990826	1.000000	0.000000	0.009174	1.000000	0.995392	0.995413	0.990826	0.990826	0.999916	0.012141
Fold 4	1.000000	0.972477	0.027523	0.000000	0.973214	0.986425	0.986239	0.972477	0.972477	0.999411	0.017891
Fold 5	1.000000	0.963303	0.036697	0.000000	0.964602	0.981982	0.981651	0.963303	0.963303	0.998317	0.018728
Fold 6	1.000000	0.963303	0.036697	0.000000	0.964602	0.981982	0.981651	0.963303	0.963303	0.998990	0.018980
Fold 7	1.000000	0.944954	0.055046	0.000000	0.947826	0.973214	0.972477	0.944954	0.944954	0.998906	0.024183
Fold 8	1.000000	0.972477	0.027523	0.000000	0.973214	0.986425	0.986239	0.972477	0.972477	0.997812	0.016196
Fold 9	1.000000	0.972477	0.027523	0.000000	0.972973	0.986301	0.986175	0.972477	0.972353	0.999405	0.019049
Fold 10	1.000000	0.972222	0.027778	0.000000	0.973214	0.986425	0.986175	0.972222	0.972346	0.999830	0.015027
Average	0.999083	0.971534	0.028466	0.000917	0.972484	0.985548	0.985308	0.970617	0.970617	0.999082	0.017240

KNN Metrics Across Folds:

	TPR (Recall)	TNR (Specificity)	FPR	FNR	Precision	F1-Score	Accuracy	TSS	HSS	AUC	Brier Score
Fold 1	0.990826	0.963303	0.036697	0.009174	0.964286	0.977376	0.977064	0.954128	0.954128	0.977064	0.022936
Fold 2	1.000000	0.944954	0.055046	0.000000	0.947826	0.973214	0.972477	0.944954	0.944954	0.972477	0.027523
Fold 3	0.990826	0.963303	0.036697	0.009174	0.964286	0.977376	0.977064	0.954128	0.954128	0.977064	0.022936
Fold 4	0.990826	0.944954	0.055046	0.009174	0.947368	0.968610	0.967890	0.935780	0.935780	0.967890	0.032110
Fold 5	0.981651	0.926606	0.073394	0.018349	0.930435	0.955357	0.954128	0.908257	0.908257	0.954128	0.045872
Fold 6	0.990826	0.954128	0.045872	0.009174	0.955752	0.972973	0.972477	0.944954	0.944954	0.972477	0.027523
Fold 7	1.000000	0.908257	0.091743	0.000000	0.915966	0.956140	0.954128	0.908257	0.908257	0.954128	0.045872
Fold 8	0.972477	0.944954	0.055046	0.027523	0.946429	0.959276	0.958716	0.917431	0.917431	0.958716	0.041284
Fold 9	1.000000	0.935780	0.064220	0.000000	0.939130	0.968610	0.967742	0.935780	0.935502	0.967890	0.032258
Fold 10	1.000000	0.972222	0.027778	0.000000	0.973214	0.986425	0.986175	0.972222	0.972346	0.986111	0.013825
Average	0.991743	0.945846	0.054154	0.008257	0.948469	0.969536	0.968786	0.937589	0.937574	0.968795	0.031214

LSTM Metrics Across Folds:

	TPR (Recall)	TNR (Specificity)	FPR	FNR	Precision	F1-Score	Accuracy	TSS	HSS	AUC	Brier Score
Fold 1	0.990826	0.990826	0.009174	0.009174	0.990826	0.990826	0.990826	0.981651	0.981651	0.999747	0.007572
Fold 2	0.972477	0.972477	0.027523	0.027523	0.972477	0.972477	0.972477	0.944954	0.944954	0.998232	0.017169
Fold 3	1.000000	0.990826	0.009174	0.000000	0.990909	0.995434	0.995413	0.990826	0.990826	0.998064	0.007590
Fold 4	0.990826	0.963303	0.036697	0.009174	0.964286	0.977376	0.977064	0.954128	0.954128	0.997475	0.018599
Fold 5	0.990826	0.981651	0.018349	0.009174	0.981818	0.986301	0.986239	0.972477	0.972477	0.998148	0.010931
Fold 6	1.000000	0.963303	0.036697	0.000000	0.964602	0.981982	0.981651	0.963303	0.963303	1.000000	0.008135
Fold 7	1.000000	0.990826	0.009174	0.000000	0.990909	0.995434	0.995413	0.990826	0.990826	1.000000	0.002494
Fold 8	1.000000	0.981651	0.018349	0.000000	0.981982	0.990909	0.990826	0.981651	0.981651	0.999832	0.006788
Fold 9	1.000000	0.981651	0.018349	0.000000	0.981818	0.990826	0.990783	0.981651	0.981568	1.000000	0.004684
Fold 10	1.000000	0.962963	0.037037	0.000000	0.964602	0.981982	0.981567	0.962963	0.963127	1.000000	0.015270
Average	0.994495	0.977948	0.022052	0.005505	0.978423	0.986355	0.986226	0.972443	0.972451	0.999150	0.009923

Summary of Average Metrics Across All Folds:

	Accuracy	Precision	F1-Score	AUC	TPR (Recall)	TNR (Specificity)	FPR	FNR	Brier Score
Random Forest	0.985308	0.972484	0.985548	0.999082	0.999083	0.971534	0.028466	0.000917	0.017240
KNN	0.968786	0.948469	0.969536	0.968795	0.991743	0.945846	0.054154	0.008257	0.031214
LSTM	0.986226	0.978423	0.986355	0.999150	0.994495	0.977948	0.022052	0.005505	0.009923

1. Random Forest

- **Strengths:**
  - Achieved the highest **Recall (TPR)** of 0.999083, indicating its exceptional ability to correctly classify positive samples.
  - **F1-Score** and **Accuracy** were both above 98.5%, showcasing a strong balance between Precision and Recall.
  - Low **False Negative Rate (FNR)** of 0.000917 indicates minimal misclassification of positive samples.
  - **AUC** of 0.999082 highlights near-perfect discrimination ability.
- **Limitations:**
  - **False Positive Rate (FPR)** was slightly higher (0.028466) compared to LSTM, meaning it produced more false positives.

2. KNN

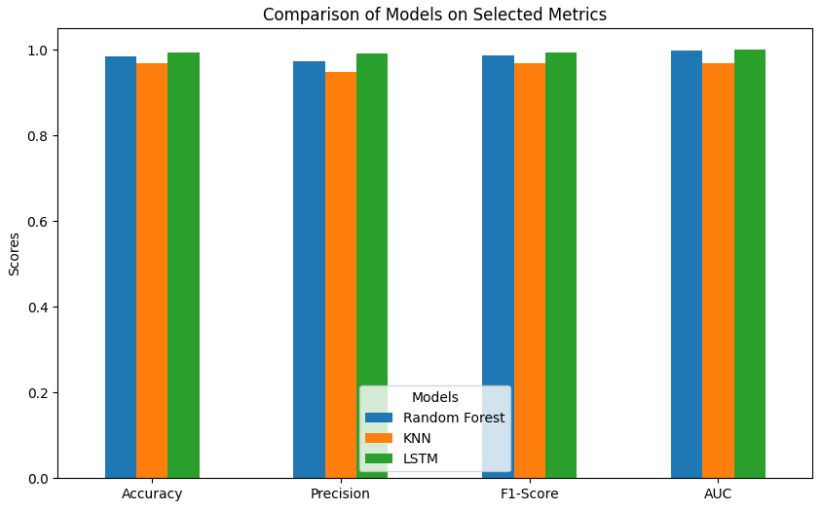
- **Strengths:**
  - Achieved a good overall performance with an **Accuracy** of 96.87%.
  - The **Recall (TPR)** of 0.991743 and **Specificity (TNR)** of 0.945846 indicate a reasonable balance between classifying positive and negative samples.
  - **Precision** of 94.84% demonstrates that most of the positive classifications were correct.
- **Limitations:**
  - Higher **False Positive Rate (FPR)** (0.054154) and **False Negative Rate (FNR)** (0.008257) compared to Random Forest and LSTM.
  - Lower **AUC** of 0.968795 compared to the other models indicates weaker discrimination between classes.

3. LSTM

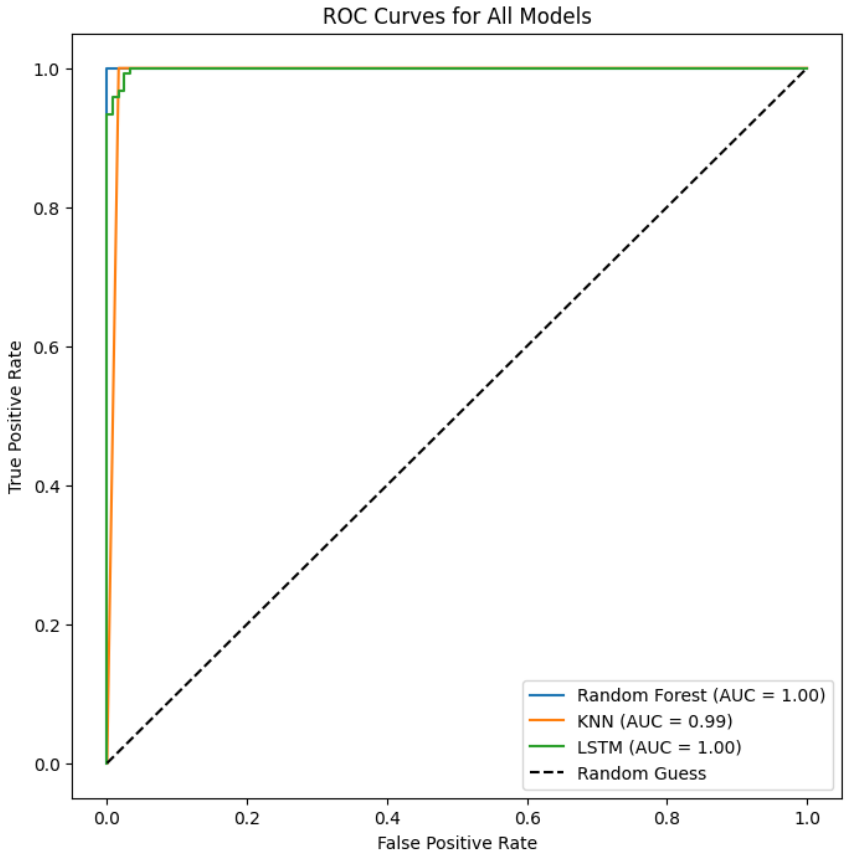
- **Strengths:**
  - Outperformed the other models in most metrics:
    - Achieved the highest **Accuracy** of 98.62% and **F1-Score** of 98.64%.
    - Maintained a high **AUC** of 0.999150, indicating excellent performance in distinguishing between positive and negative classes.
  - Exhibited the lowest **False Positive Rate (FPR)** (0.022052) and **Brier Score** (0.009923), showcasing its superior calibration of probabilistic predictions.
- **Limitations:**

- Training LSTM requires more computational resources and time compared to Random Forest and KNN.

Visualization Insights



1. **Bar Plot of Selected Metrics:**
   - The bar plot shows a clear comparison of metrics such as Accuracy, Precision, F1-Score, and AUC for all three models.
   - LSTM slightly outperformed Random Forest in most metrics, while KNN lagged behind.



2. **ROC Curves:**
    - The ROC curves for all three models demonstrate their ability to differentiate between classes.
    - LSTM and Random Forest have near-perfect curves with AUC values close to 1, while KNN has a slightly lower curve.

Conclusion

- The **LSTM model** emerged as the best-performing algorithm, excelling in Accuracy, Precision, F1-Score, and AUC. Its low FPR and Brier Score further reinforce its reliability.
- **Random Forest** was a close competitor, with slightly lower scores but faster training times, making it a strong alternative when computational resources are limited.
- **KNN** provided reasonable performance but struggled in areas like FPR and AUC, making it the weakest model in this comparison.

Overall, while LSTM is the most robust model for this dataset, Random Forest can serve as a faster and simpler solution with comparable performance.

7. Instructions for Running the Code

To successfully run the project and reproduce the results, follow these steps:

Step 1: Install Required Software

Ensure you have the following installed:

- **Python** (version 3.7 or higher)
- **Jupyter Notebook**
- Required Python libraries:

  pip install pandas numpy scikit-learn imbalanced-learn matplotlib tensorflow

Step 2: Download the Dataset

- Download the **Car Evaluation dataset** and save it as car.data in the project directory.

Step 3: Clone the GitHub Repository

- Clone the GitHub repository (details provided below) to access the project files:
  bash

Step 4: Open the Jupyter Notebook

- Navigate to the project directory and open the Jupyter Notebook:

  jupyter notebook project_notebook.ipynb

Step 5: Execute the Notebook

- Run all the cells in the notebook sequentially to preprocess the data, train the models, evaluate performance, and generate visualizations.

Expected Outputs

- The notebook will display:
  - Preprocessed dataset preview.
  - Metrics and results for each fold across all models.
  - Summary of average metrics for Random Forest, KNN, and LSTM.
  - Visualizations, including bar plots of selected metrics and ROC curves for all models.

Step 6: Review Results

- Analyze the outputs and compare the performance of the three models.

8. GitHub Repository

Repository Link

The project is hosted on GitHub for easy access:

- **GitHub Repository:**

Repository Structure

- project_notebook.ipynb: Jupyter Notebook containing the full code, including preprocessing, modeling, evaluation, and visualizations.
- car.data: Dataset file used for the project.
- report.pdf: The final project report summarizing the results and discussion.

9. Conclusion

This project successfully demonstrated the implementation and evaluation of three machine learning algorithms—Random Forest, KNN, and LSTM—on the Car Evaluation dataset. Here are the key takeaways:

- **Model Comparison:**
    - LSTM emerged as the best-performing model, excelling in metrics like Accuracy, F1-Score, and AUC. Its ability to handle complex data relationships and probabilistic calibration stood out.
    - Random Forest was a close competitor, offering high accuracy and fast training, making it suitable for resource-constrained environments.
    - KNN showed reasonable performance but lagged behind in metrics like FPR and AUC, highlighting its limitations for this dataset.
- **Key Strengths:**
    - Comprehensive preprocessing steps, including handling categorical data and balancing the dataset using SMOTE.
    - Metrics calculated for each fold and averaged for a robust evaluation.
    - Visualizations provided clear insights into model performance.
- **Limitations and Future Work:**
    - LSTM requires more computational resources and training time compared to the other models.
    - Future work could include experimenting with additional deep learning architectures or hyperparameter tuning to further improve results.

This project highlights the importance of choosing the right algorithm based on the dataset and performance requirements while demonstrating the effectiveness of rigorous evaluation methods.