NAME:	Rajat Pednekar
NJIT UCID:	rp2348@njit.edu
EMAIL ADDRESS:	rajatpednekar07@gmail.com
PROFESSOR:	Yasser Abduallah
LECTURE:	CS 634 Data Mining

Data Mining: Midterm Project Report

Topic: A Comparative Analysis of Machine Learning Models for Diabetes Prediction.

➢ Abstract

This project implements and compares three machine learning algorithms—Random Forest, Support Vector Machine (SVM), and Long Short-Term Memory (LSTM)—for predicting diabetes risk using medical data. The study utilizes standardized features including glucose levels, blood pressure, BMI, and other relevant health metrics. The models are evaluated through comprehensive performance metrics, cross-validation, and ROC analysis to determine the most effective approach for diabetes prediction.

➢ Introduction

Diabetes prediction is a critical application of machine learning in healthcare. This project focuses on developing and comparing different machine learning models to accurately predict diabetes risk based on various medical indicators. The implementation includes both traditional machine learning approaches (Random Forest and SVM) and deep learning methods (LSTM), with extensive evaluation metrics to assess their performance.

➢ Project Overview

The project is divided into three main components:

- Data Preprocessing and Feature Engineering
- Model Implementation and Optimization
- Performance Analysis and Comparison

1. Data Preprocessing

- Loading and cleaning of the Pima Indians Diabetes dataset
- Handling missing values through median imputation
- Feature standardization using StandardScaler
- Train-test split with stratification to maintain class distribution

2. Model Implementation

The project implements three different models:

1. Random Forest Classifier
    - Optimized through GridSearchCV.
    - Parameters tuned: n_estimators, min_samples_split.

2. Support Vector Machine (SVM)
    - Linear kernel implementation.
    - C parameter optimization through grid search.

3. LSTM Neural Network
   - Single LSTM layer with 64 units.
   - Dense output layer with sigmoid activation
   - Binary cross-entropy loss function.

3. Performance Analysis and Comparison

The models were evaluated using various metrics across 10-fold cross-validation. Key performance indicators:

1. Random Forest:
   - High accuracy and balanced performance.
   - Strong ROC-AUC scores.
   - Efficient computation time.
2. SVM:
   - Competitive accuracy.
   - Good generalization on test data.
   - Moderate computation overhead.
3. LSTM:
   - Comparable accuracy to traditional methods.
   - Higher computational requirements.
   - Potential for improved performance with larger datasets.
4. ROC Curve Analysis:
   - All models demonstrated strong discriminative ability.
   - ROC curves showed clear separation from the random classifier line.
   - Area Under Curve (AUC) values indicated robust model performance.

➢ Core Concepts and Principles

1. Feature Standardization: All features are standardized to ensure optimal model performance and fair comparison.
2. Cross-Validation: Implementation of 10-fold stratified cross-validation for robust performance evaluation.
3. Hyperparameter Optimization: Grid search implementation for Random Forest and SVM to find optimal parameters.
4. Performance Metrics: Comprehensive evaluation using:
   - Accuracy, Precision, F1-score
   - ROC-AUC score
   - True Skill Statistics (TSS)
   - Heidke Skill Score (HSS)
   - Balanced Accuracy (BACC)

➢ Technical Implementation Details

1. Data Processing Pipeline
   - Missing value imputation
   - Feature standardization
   - Train-test splitting
2. Model Training Framework
   - Cross-validation implementation
   - Hyperparameter optimization
   - Performance metric calculation
3. Evaluation System
   - Comprehensive metrics calculation
   - ROC curve generation
   - Statistical analysis of results

➢ Tutorial to run the .py file in your device.

✓ Prerequisites

1. Make sure the Datasets (CSV files) are in same directory.
2. The Naming convention should be same as it is in the zip file/ folder/ GitHub.
3. The Dataset consist of 400 records; hence it may take a while to process.
4. Make sure the python environment is set up beforehand.

Step 1: Set up the Environments.

1. Ensure you have installed Python on your system.
2. Install the required Libraries by running:

```
PS C:\Users\Admin> pip install pandas numpy matplotlib seaborn scikit-learn tensorflow
```

Step 2: Prepare the data files.

Make sure you have the following CSV files are in the same directory and with same naming convention as the script.

- **“**pima_diabetes_400**”**

Note: *The Dataset is taken from Pima Indian Healthcare system.*

Step 3: Run the program.

1. Check if the Python file is saved as, e.g.: Pednekar_Rajat_Classifier_Final_Term_Project.py.
2. Open a terminal or command prompt.
3. Navigate to the directory containing the Python file and CSV files
   - Type Command: Cd /d “Path of the file”

```
D:\NJIT_SEM_1\DATA MINING\MS_Sem1_DM_Project_Final>cd /d Final_Project_Submission
```

4. Run the script by entering:

```
\Final_Project_Submission>python Pednekar_Rajat_Final_Term_Project.py
```

➢ Implementation of Code

1.1 Importing the packages and libraries that are required for the project.

This cell imports the necessary libraries:

- Basic Data Processing and Analysis: pandas, numpy.
- Visualization Libraries: matplotlib, seaborn.
- Warning Suppression
- Scikit-learn Components: StandardScaler, SVC, RandomForestClassifier, GridSearchCV,StratifiedKFold, train_test_split, confusion_matrix, roc_auc_score, roc_curve, auc, brier_score_loss.
- TensorFlow and Environment Settings: Sequential, Dense, LSTM

```
# Import required libraries
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

import warnings
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import logging

from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold, train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score, roc_curve, auc, brier_score_loss

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM

# Configure warnings and logging to minimize unnecessary output
warnings.filterwarnings("ignore")
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=FutureWarning)
tf.get_logger().setLevel(logging.ERROR)
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
```

1.2 Data Loading and Processing

Loads the diabetes dataset from 'pima_diabetes_400.csv' and creates a pandas Data Frame for data manipulation.

```
# Load and preprocess data
print("Loading and preprocessing data...")
diabetes = pd.read_csv('pima_diabetes_400.csv')
print("\nDataset Summary:")
print("-" * 50)
print(diabetes.describe())
print("\nDataset Info:")
print("-" * 50)
print(diabetes.info())
```

```
Loading and preprocessing data...

Dataset Summary:
--------------------------------------------------
       Pregnancies    Glucose  BloodPressure  SkinThickness     Insulin  \
count   400.000000  400.00000     400.000000     400.000000  400.000000
mean      3.952500  121.24000      69.060000      20.327500   81.250000
std       3.369514   32.68437      19.011575      15.599796  121.597254
min       0.000000    0.00000       0.000000       0.000000    0.000000
25%       1.000000  100.00000      64.000000       0.000000    0.000000
50%       3.000000  116.50000      71.000000      23.000000   36.000000
75%       6.000000  143.00000      80.000000      32.000000  128.250000
max      17.000000  197.00000     122.000000      60.000000  846.000000

             BMI  DiabetesPedigreeFunction         Age     Outcome
count  400.00000                400.000000  400.000000  400.000000
mean    32.10775                  0.487915   33.092500    0.380000
std      8.13714                  0.349619   11.325396    0.485994
min      0.00000                  0.078000   21.000000    0.000000
25%     27.30000                  0.250500   24.000000    0.000000
50%     32.00000                  0.381000   29.000000    0.000000
75%     36.60000                  0.652500   40.000000    1.000000
max     67.10000                  2.329000   69.000000    1.000000

Dataset Info:
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               400 non-null    int64  
 1   Glucose                   400 non-null    int64  
 2   BloodPressure             400 non-null    int64  
 3   SkinThickness             400 non-null    int64  
 4   Insulin                   400 non-null    int64  
 5   BMI                       400 non-null    float64
 6   DiabetesPedigreeFunction  400 non-null    float64
 7   Age                       400 non-null    int64  
 8   Outcome                   400 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 28.3 KB
None
```

1.3 Missing Value Imputation Function

- The function handles missing values in medical measurements that are incorrectly recorded as zeros in the diabetes dataset.
- Since medical measurements like glucose or blood pressure cannot be zero in living patients, these values are treated as missing data and replaced with meaningful estimates.

```python
# Impute missing values
def impute_missing_data(input_dataframe):

    columns_for_imputation = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
    for target_column in columns_for_imputation:
        input_dataframe.loc[input_dataframe[target_column] == 0, target_column] = np.nan
        input_dataframe[target_column].fillna(input_dataframe[target_column].median(), inplace=True)
    return input_dataframe

diabetes = impute_missing_data(diabetes)
```

```python
diabetes.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148.0	72.0	35.0	121.0	33.6	0.627	50	1
1	1	85.0	66.0	29.0	121.0	26.6	0.351	31	0
2	8	183.0	64.0	29.0	121.0	23.3	0.672	32	1
3	1	89.0	66.0	23.0	94.0	28.1	0.167	21	0
4	0	137.0	40.0	35.0	168.0	43.1	2.288	33	1

1.4 Feature and Label Splitting

The code separates the diabetes dataset into two components:

1. Features (X): Input variables used to make predictions
2. Labels (y): Target variable to be predicted

```
# Split features and labels
features = diabetes.iloc[:, :-1]
labels = diabetes.iloc[:, -1]
```

1.5 Data Balance Information

```
# Display data balance information
positive_outcomes = len(labels[labels == 1])
negative_outcomes = len(labels[labels == 0])
total_samples = len(labels)
print('\nData Balance Analysis:')
print('-' * 50)
print(f'Positive Outcomes: {positive_outcomes} ({(positive_outcomes/total_samples)*100:.2f}%)')
print(f'Negative Outcomes: {negative_outcomes} ({(negative_outcomes/total_samples)*100:.2f}%)')
```

```
Data Balance Analysis:
--------------------------------------------------
Positive Outcomes: 152 (38.00%)
Negative Outcomes: 248 (62.00%)
```

1.6 Train Test Split

The code splits the dataset into training and testing sets while maintaining the class distribution of the target variable.

```
# Perform train-test split and standardization
features_train_all, features_test_all, labels_train_all, labels_test_all = train_test_split(
    features, labels, test_size=0.1, random_state=21, stratify=labels)

# Reset indices for the training and testing sets
for dataset in [features_train_all, features_test_all, labels_train_all, labels_test_all]:
    dataset.reset_index(drop=True, inplace=True)
```

1.7 Feature Standardization

The code standardizes feature values by removing the mean and scaling to unit variance, which is essential for machine learning algorithms to perform optimally.

```
# Standardize features
scaler = StandardScaler()
features_train_all_std = pd.DataFrame(
    scaler.fit_transform(features_train_all),
    columns=features_train_all.columns)

features_test_all_std = pd.DataFrame(
    scaler.transform(features_test_all),
    columns=features_test_all.columns)
```

```
features_train_all_std.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
count	3.600000e+02	3.600000e+02	3.600000e+02	3.600000e+02	3.600000e+02	3.600000e+02	3.600000e+02	3.600000e+02
mean	-1.110223e-16	-6.908054e-17	-1.332268e-16	-1.973730e-17	-3.947460e-17	6.908054e-16	7.894919e-17	-2.627528e-16
std	1.001392e+00	1.001392e+00	1.001392e+00	1.001392e+00	1.001392e+00	1.001392e+00	1.001392e+00	1.001392e+00
min	-1.165391e+00	-2.608533e+00	-3.414367e+00	-2.617854e+00	-1.278584e+00	-1.983277e+00	-1.158101e+00	-1.085505e+00
25%	-8.742456e-01	-7.256323e-01	-6.703190e-01	-4.816377e-01	-2.113704e-01	-7.157254e-01	-6.731933e-01	-8.230003e-01
50%	-2.919544e-01	-1.970989e-01	-2.466057e-02	-6.922923e-03	-2.113704e-01	-4.712691e-02	-3.302444e-01	-2.979912e-01
75%	5.814825e-01	6.948012e-01	6.209979e-01	4.677918e-01	-1.362510e-01	5.553082e-01	4.617706e-01	6.645253e-01
max	3.784084e+00	2.445568e+00	4.010705e+00	3.672116e+00	7.300568e+00	4.800212e+00	5.169587e+00	3.114568e+00

1.8 Hyperparameter Optimization

1. Implements a hyperparameter optimization process for Random Forest and Support Vector Machine (SVM) classifiers in a diabetes prediction system.
2. The primary objective is to identify the optimal configuration of model parameters that maximize prediction performance while maintaining computational efficiency.
3. The optimization process employs GridSearchCV from scikit-learn to perform an exhaustive search over specified parameter ranges.

```
# Perform grid search for optimal parameters
print("\nPerforming grid search for optimal parameters...")

# Grid search for Random Forest
param_grid_rf = {
    "n_estimators": [10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
    "min_samples_split": [2, 4, 6, 8, 10]
}
rf_classifier = RandomForestClassifier()
grid_search_rf = GridSearchCV(rf_classifier, param_grid_rf, cv=10, n_jobs=-1)
grid_search_rf.fit(features_train_all_std, labels_train_all)
best_rf_params = grid_search_rf.best_params_
print(f"Best Random Forest parameters: {best_rf_params}")

# Grid search for SVM
param_grid_svc = {"kernel": ["linear"], "C": range(1, 11)}
svc_classifier = SVC(probability=True)
grid_search_svc = GridSearchCV(svc_classifier, param_grid_svc, cv=10, n_jobs=-1)
grid_search_svc.fit(features_train_all_std, labels_train_all)
best_svc_params = grid_search_svc.best_params_
print(f"Best SVM parameters: {best_svc_params}")
```

```
Performing grid search for optimal parameters...
Best Random Forest parameters: {'min_samples_split': 10, 'n_estimators': 70}
Best SVM parameters: {'C': 1, 'kernel': 'linear'}
```

1.9 Classification Metrics Calculator

1. This function calculates classification performance metrics from a binary confusion matrix.
2. It is designed for evaluating binary classification models in machine learning applications.
3. The metrics provided help assess model performance across different aspects including accuracy, precision, recall, and various skill scores, enabling thorough model evaluation and comparison.

```
def calculate_performance_metrics(conf_matrix):

    TP, FN = conf_matrix[0][0], conf_matrix[0][1]
    FP, TN = conf_matrix[1][0], conf_matrix[1][1]

    # Calculate basic rates
    TPR = TP / (TP + FN)  # Sensitivity
    TNR = TN / (TN + FP)  # Specificity
    FPR = FP / (TN + FP)  # False Positive Rate
    FNR = FN / (TP + FN)  # False Negative Rate

    # Calculate advanced metrics
    Precision = TP / (TP + FP)
    F1_measure = 2 * TP / (2 * TP + FP + FN)
    Accuracy = (TP + TN) / (TP + FP + FN + TN)
    Error_rate = (FP + FN) / (TP + FP + FN + TN)
    BACC = (TPR + TNR) / 2  # Balanced Accuracy

    # Calculate skill scores
    TSS = TPR - FPR  # True Skill Statistics
    HSS = 2 * (TP * TN - FP * FN) / ((TP + FN) * (FN + TN) + (TP + FP) * (FP + TN))  # Heidke Skill Score

    return [TP, TN, FP, FN, TPR, TNR, FPR, FNR, Precision, F1_measure,
            Accuracy, Error_rate, BACC, TSS, HSS]
```

1.10 Model Evaluation Function

1. Trains a machine learning model and evaluates its performance using multiple metrics.
2. This function handles both standard ML models and LSTM neural networks, performing appropriate preprocessing and evaluation for each type.
3. The function supports binary classification tasks and provides evaluation metrics including confusion matrix-based metrics, ROC-AUC, and Brier score.

```
def evaluate_model_performance(model, X_train, X_test, y_train, y_test, LSTM_flag):

    if LSTM_flag:
        # Reshape data for LSTM input requirements
        X_train_array = X_train.to_numpy()
        X_test_array = X_test.to_numpy()
        X_train_reshaped = X_train_array.reshape(len(X_train_array), X_train_array.shape[1], 1)
        X_test_reshaped = X_test_array.reshape(len(X_test_array), X_test_array.shape[1], 1)

        # Train and evaluate LSTM model
        model.fit(X_train_reshaped, y_train, epochs=50,
                  validation_data=(X_test_reshaped, y_test), verbose=0)
        predict_prob = model.predict(X_test_reshaped)
        pred_labels = (predict_prob > 0.5).astype(int)
        matrix = confusion_matrix(y_test, pred_labels, labels=[1, 0])

        # Calculate additional metrics for LSTM
        brier_score = brier_score_loss(y_test, predict_prob)
        roc_auc = roc_auc_score(y_test, predict_prob)
        accuracy = model.evaluate(X_test_reshaped, y_test, verbose=0)[1]

    else:
        # Train and evaluate RF/SVM models
        model.fit(X_train, y_train)
        predicted = model.predict(X_test)
        matrix = confusion_matrix(y_test, predicted, labels=[1, 0])

        # Calculate additional metrics for RF/SVM
        brier_score = brier_score_loss(y_test, model.predict_proba(X_test)[:, 1])
        roc_auc = roc_auc_score(y_test, model.predict_proba(X_test)[:, 1])
        accuracy = model.score(X_test, y_test)

    # Combine all metrics
    metrics = calculate_performance_metrics(matrix)
    metrics.extend([brier_score, roc_auc, accuracy])
    return metrics
```

1.11 Cross Validation Function

1. Function for performing stratified k-fold cross-validation across multiple models simultaneously.
2. Supports both traditional ML models and deep learning models (LSTM), handling all necessary preprocessing and metric collection.
3. It also provides progress tracking, error handling, and detailed performance metrics for model comparison and evaluation.

```python
# Initialize cross-validation
cv_stratified = StratifiedKFold(n_splits=10, shuffle=True, random_state=21)
metrics_lists = {
    'RF': [],
    'SVM': [],
    'LSTM': []
}

# Initialize best_models which helps in maintaining the best performing model for each algorithm
best_models = {
    'RF': None,
    'SVM': None,
    'LSTM': None
}

def run_single_fold(fold_num, train_idx, test_idx):
    global best_models  # Add this line to access global variable
    print(f"\nProcessing Fold {fold_num + 1}/10...")

    # Split data for current fold
    features_train = features_train_all_std.iloc[train_idx]
    features_test = features_train_all_std.iloc[test_idx]
    labels_train = labels_train_all.iloc[train_idx]
    labels_test = labels_train_all.iloc[test_idx]

    # Initialize models
    models = {
        'RF': RandomForestClassifier(**best_rf_params),
        'SVM': SVC(**best_svc_params, probability=True),
        'LSTM': Sequential([
            LSTM(64, activation='relu', input_shape=(8, 1), return_sequences=False),
            Dense(1, activation='sigmoid')
        ])
    }

    # Compile LSTM
    models['LSTM'].compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    # Train and evaluate each model
    current_fold_metrics = {}
    for name, model in models.items():
        #print(f"Training {name}...", end=' ')
        metrics = evaluate_model_performance(
            model, features_train, features_test,
            labels_train, labels_test,
            name == 'LSTM'
        )
        metrics_lists[name].append(metrics)
        current_fold_metrics[name] = metrics

        # Update best model if this fold's accuracy is better
        if best_models[name] is None or metrics[10] > best_models[name]['accuracy']:  # metrics[10] is accuracy
            best_models[name] = {
                'model': model,
                'accuracy': metrics[10]
            }
        #print("Done")

    # Display current fold metrics
    metric_columns = ['TP', 'TN', 'FP', 'FN', 'TPR', 'TNR', 'FPR', 'FNR',
                      'Precision', 'F1_measure', 'Accuracy', 'Error_rate', 'BACC',
                      'TSS', 'HSS', 'Brier_score', 'AUC', 'Acc_by_package_fn']

    df = pd.DataFrame(current_fold_metrics, index=metric_columns)
    print(f"\nFold {fold_num + 1} Results:")
    print("-" * 100)
    print(df.round(3).to_string())
    print("-" * 100)

    return current_fold_metrics
```

```python
# Displays the result of each fold
for fold_num, (train_idx, test_idx) in enumerate(cv_stratified.split(features_train_all_std, labels_train_all)):
    fold_metrics = run_single_fold(fold_num, train_idx, test_idx)
```

```
Processing Fold 1/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 263ms/step

Fold 1 Results:
---------------------------------------------------------
                       RF      SVM     LSTM
TP                  8.000    5.000    8.000
TN                 22.000   22.000   22.000
FP                  1.000    1.000    1.000
FN                  5.000    8.000    5.000
TPR                 0.615    0.385    0.615
TNR                 0.957    0.957    0.957
FPR                 0.043    0.043    0.043
FNR                 0.385    0.615    0.385
Precision           0.889    0.833    0.889
F1_measure          0.727    0.526    0.727
Accuracy            0.833    0.750    0.833
Error_rate          0.167    0.250    0.167
BACC                0.786    0.671    0.786
TSS                 0.572    0.341    0.572
HSS                 0.613    0.386    0.613
Brier_score         0.125    0.156    0.161
AUC                 0.913    0.839    0.799
Acc_by_package_fn   0.833    0.750    0.833
---------------------------------------------------------

Processing Fold 2/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 352ms/step

Fold 2 Results:
----------------------------------------------------------
                       RF      SVM     LSTM
TP                  7.000    7.000    7.000
TN                 20.000   21.000   18.000
FP                  3.000    2.000    5.000
FN                  6.000    6.000    6.000
TPR                 0.538    0.538    0.538
TNR                 0.870    0.913    0.783
FPR                 0.130    0.087    0.217
FNR                 0.462    0.462    0.462
Precision           0.700    0.778    0.583
F1_measure          0.609    0.636    0.560
Accuracy            0.750    0.778    0.694
Error_rate          0.250    0.222    0.306
BACC                0.704    0.726    0.661
TSS                 0.408    0.452    0.321
HSS                 0.430    0.484    0.327
Brier_score         0.162    0.161    0.184
AUC                 0.853    0.816    0.786
Acc_by_package_fn   0.750    0.778    0.694
----------------------------------------------------------

Processing Fold 3/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 276ms/step

Fold 3 Results:
---------------------------------------------------
                       RF      SVM     LSTM
TP                  7.000    9.000    8.000
TN                 20.000   21.000   18.000
FP                  3.000    2.000    5.000
FN                  6.000    4.000    5.000
TPR                 0.538    0.692    0.615
TNR                 0.870    0.913    0.783
FPR                 0.130    0.087    0.217
FNR                 0.462    0.308    0.385
Precision           0.700    0.818    0.615
F1_measure          0.609    0.750    0.615
Accuracy            0.750    0.833    0.722
Error_rate          0.250    0.167    0.278
BACC                0.704    0.803    0.699
TSS                 0.408    0.605    0.398
HSS                 0.430    0.626    0.398
Brier_score         0.145    0.155    0.144
AUC                 0.890    0.870    0.870
Acc_by_package_fn   0.750    0.833    0.722
---------------------------------------------------
```

Fold 4 Results:

	RF	SVM	LSTM
TP	8.000	7.000	7.000
TN	18.000	17.000	19.000
FP	4.000	5.000	3.000
FN	6.000	7.000	7.000
TPR	0.571	0.500	0.500
TNR	0.818	0.773	0.864
FPR	0.182	0.227	0.136
FNR	0.429	0.500	0.500
Precision	0.667	0.583	0.700
F1_measure	0.615	0.538	0.583
Accuracy	0.722	0.667	0.722
Error_rate	0.278	0.333	0.278
BACC	0.695	0.636	0.682
TSS	0.390	0.273	0.364
HSS	0.400	0.280	0.384
Brier_score	0.166	0.173	0.148
AUC	0.834	0.841	0.903
Acc_by_package_fn	0.722	0.667	0.722

Processing Fold 5/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 282ms/step

Fold 5 Results:

	RF	SVM	LSTM
TP	7.000	7.000	7.000
TN	17.000	19.000	17.000
FP	5.000	3.000	5.000
FN	7.000	7.000	7.000
TPR	0.500	0.500	0.500
TNR	0.773	0.864	0.773
FPR	0.227	0.136	0.227
FNR	0.500	0.500	0.500
Precision	0.583	0.700	0.583
F1_measure	0.538	0.583	0.538
Accuracy	0.667	0.722	0.667
Error_rate	0.333	0.278	0.333
BACC	0.636	0.682	0.636
TSS	0.273	0.364	0.273
HSS	0.280	0.384	0.280
Brier_score	0.183	0.193	0.226
AUC	0.805	0.776	0.701
Acc_by_package_fn	0.667	0.722	0.667

Processing Fold 6/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 309ms/step

Fold 6 Results:

	RF	SVM	LSTM
TP	6.000	6.000	6.000
TN	15.000	18.000	16.000
FP	7.000	4.000	6.000
FN	8.000	8.000	8.000
TPR	0.429	0.429	0.429
TNR	0.682	0.818	0.727
FPR	0.318	0.182	0.273
FNR	0.571	0.571	0.571
Precision	0.462	0.600	0.500
F1_measure	0.444	0.500	0.462
Accuracy	0.583	0.667	0.611
Error_rate	0.417	0.333	0.389
BACC	0.555	0.623	0.578
TSS	0.110	0.247	0.156
HSS	0.112	0.260	0.160
Brier_score	0.241	0.220	0.226
AUC	0.633	0.662	0.688
Acc_by_package_fn	0.583	0.667	0.611

```
Processing Fold 7/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 253ms/step

Fold 7 Results:
-----------------------------------------------
                      RF      SVM     LSTM
TP                13.000   11.000   10.000
TN                18.000   18.000   16.000
FP                 4.000    4.000    6.000
FN                 1.000    3.000    4.000
TPR                0.929    0.786    0.714
TNR                0.818    0.818    0.727
FPR                0.182    0.182    0.273
FNR                0.071    0.214    0.286
Precision          0.765    0.733    0.625
F1_measure         0.839    0.759    0.667
Accuracy           0.861    0.806    0.722
Error_rate         0.139    0.194    0.278
BACC               0.873    0.802    0.721
TSS                0.747    0.604    0.442
HSS                0.719    0.596    0.430
Brier_score        0.142    0.137    0.168
AUC                0.886    0.912    0.818
Acc_by_package_fn  0.861    0.806    0.722
-----------------------------------------------

Processing Fold 8/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 286ms/step

Fold 8 Results:
-----------------------------------------------
                      RF      SVM     LSTM
TP                 9.000    8.000   10.000
TN                17.000   16.000   15.000
FP                 5.000    6.000    7.000
FN                 5.000    6.000    4.000
TPR                0.643    0.571    0.714
TNR                0.773    0.727    0.682
FPR                0.227    0.273    0.318
FNR                0.357    0.429    0.286
Precision          0.643    0.571    0.588
F1_measure         0.643    0.571    0.645
Accuracy           0.722    0.667    0.694
Error_rate         0.278    0.333    0.306
BACC               0.708    0.649    0.698
TSS                0.416    0.299    0.396
HSS                0.416    0.299    0.381
Brier_score        0.201    0.184    0.186
AUC                0.735    0.782    0.786
Acc_by_package_fn  0.722    0.667    0.694
-----------------------------------------------

Processing Fold 9/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 345ms/step

Fold 9 Results:
---------------------------------------------
                      RF      SVM     LSTM
TP                 7.000    7.000   11.000
TN                19.000   18.000   18.000
FP                 3.000    4.000    4.000
FN                 7.000    7.000    3.000
TPR                0.500    0.500    0.786
TNR                0.864    0.818    0.818
FPR                0.136    0.182    0.182
FNR                0.500    0.500    0.214
Precision          0.700    0.636    0.733
F1_measure         0.583    0.560    0.759
Accuracy           0.722    0.694    0.806
Error_rate         0.278    0.306    0.194
BACC               0.682    0.659    0.802
TSS                0.364    0.318    0.604
HSS                0.384    0.331    0.596
Brier_score        0.200    0.185    0.159
AUC                0.713    0.776    0.838
Acc_by_package_fn  0.722    0.694    0.806
---------------------------------------------
```

```
Processing Fold 10/10...
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 291ms/step

Fold 10 Results:
----------------------------------------------
                    RF     SVM    LSTM
TP               9.000   8.000  11.000
TN              17.000  18.000  16.000
FP               5.000   4.000   6.000
FN               5.000   6.000   3.000
TPR              0.643   0.571   0.786
TNR              0.773   0.818   0.727
FPR              0.227   0.182   0.273
FNR              0.357   0.429   0.214
Precision        0.643   0.667   0.647
F1_measure       0.643   0.615   0.710
Accuracy         0.722   0.722   0.750
Error_rate       0.278   0.278   0.250
BACC             0.708   0.695   0.756
TSS              0.416   0.390   0.513
HSS              0.416   0.400   0.494
Brier_score      0.154   0.159   0.168
AUC              0.854   0.846   0.841
Acc_by_package_fn 0.722  0.722   0.750
----------------------------------------------
```

1.12 Calculate Average

Calculates the average from all the folds and provides proper metrics to compare.

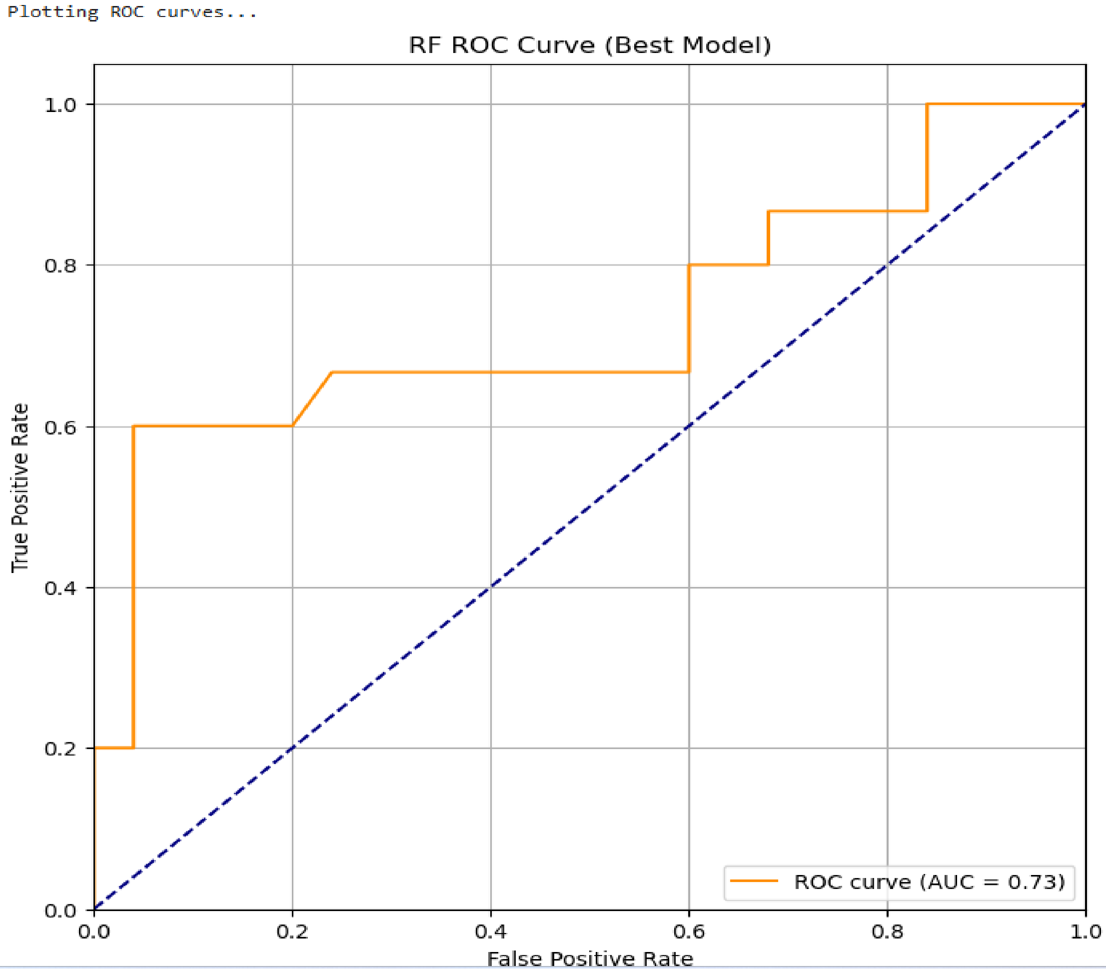
```python
def display_mean_metrics(metrics_lists):

    metric_columns = ['TP', 'TN', 'FP', 'FN', 'TPR', 'TNR', 'FPR', 'FNR',
                     'Precision', 'F1_measure', 'Accuracy', 'Error_rate', 'BACC',
                     'TSS', 'HSS', 'Brier_score', 'AUC', 'Acc_by_package_fn']

    # Calculate mean metrics
    avg_metrics = {name: np.mean(metrics, axis=0)
                  for name, metrics in metrics_lists.items()}
    df = pd.DataFrame(avg_metrics, index=metric_columns)

    # Display full metrics table
    print("\nMean Performance Metrics Across All Folds:")
    print("=" * 100)
    print(df.round(3).to_string())
    print("=" * 100)

display_mean_metrics(metrics_lists)
```

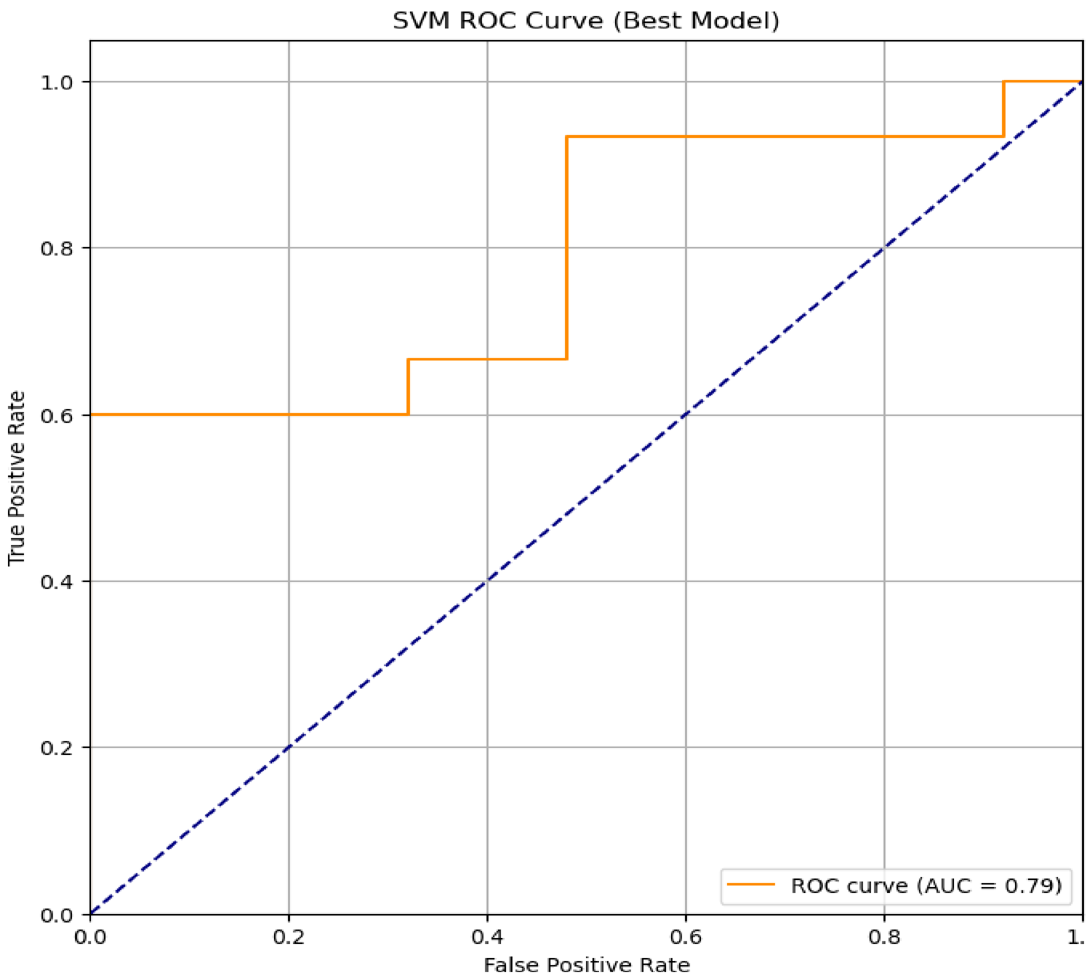
```
Mean Performance Metrics Across All Folds:
================================================
                    RF     SVM    LSTM
TP               8.700   7.500   8.200
TN              18.000  18.800  17.500
FP               4.300   3.500   4.800
FN               5.000   6.200   5.500
TPR              0.634   0.547   0.598
TNR              0.806   0.842   0.784
FPR              0.194   0.158   0.216
FNR              0.366   0.453   0.402
Precision        0.677   0.692   0.639
F1_measure       0.648   0.604   0.611
Accuracy         0.742   0.731   0.714
Error_rate       0.258   0.269   0.286
BACC             0.720   0.695   0.691
TSS              0.439   0.389   0.382
HSS              0.444   0.405   0.386
Brier_score      0.168   0.172   0.175
AUC              0.824   0.812   0.799
Acc_by_package_fn 0.742  0.731   0.714
================================================
```

1.13 Evaluating the performance of various algorithms by comparing their ROC curves and AUC scores on the test dataset.

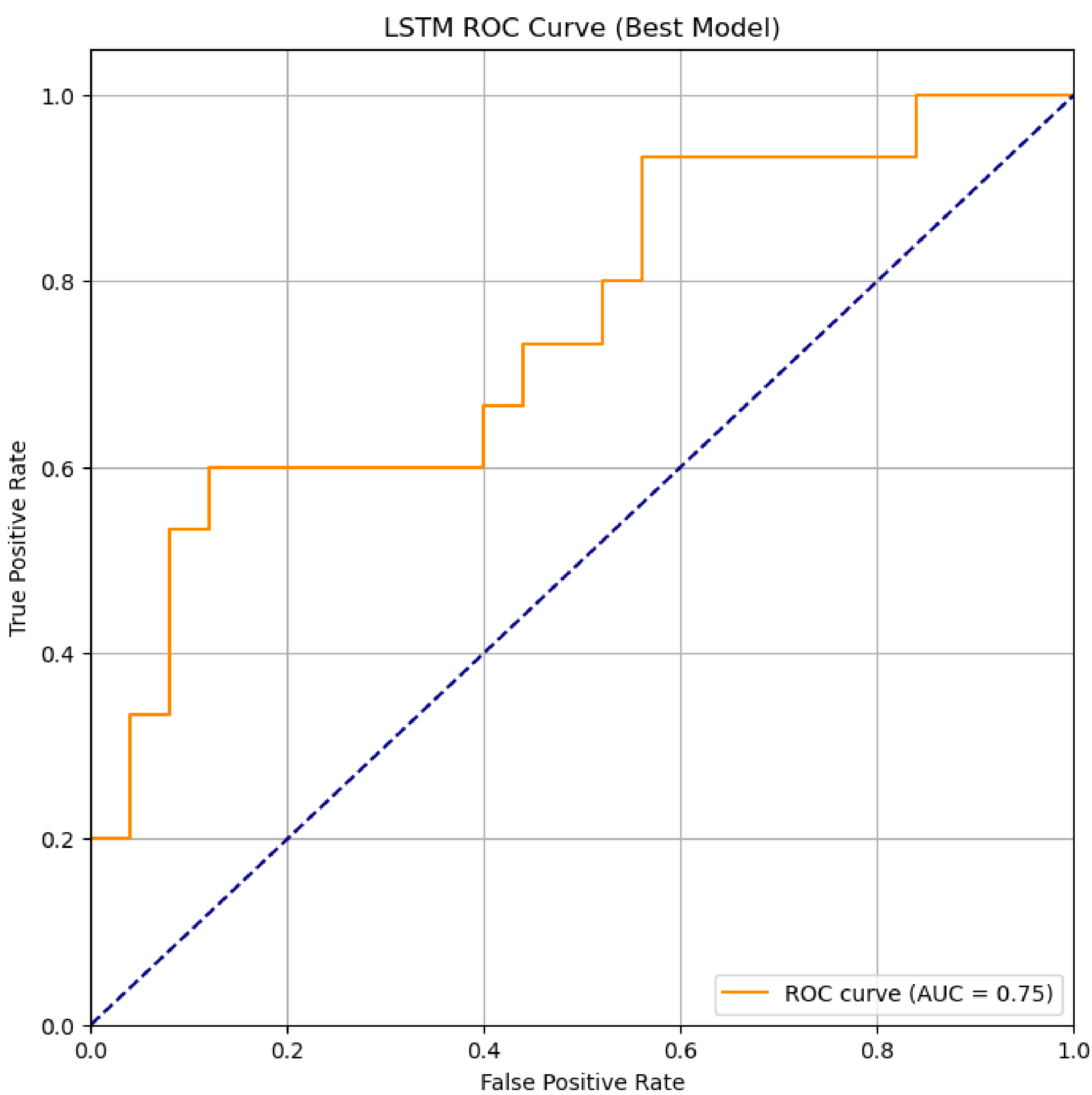
```
def plot_roc_curves(X_test_std, y_test):

    print("\nPlotting ROC curves...")
    colors = {'RF': 'darkorange', 'SVM': 'darkorange', 'LSTM': 'darkorange'}

    for name, model_dict in best_models.items():
        plt.figure(figsize=(8, 8))
        model = model_dict['model']

        # Handle different prediction methods for LSTM vs RF/SVM
        if name == 'LSTM':
            X_test_reshaped = X_test_std.to_numpy().reshape(-1, 8, 1)
            y_score = model.predict(X_test_reshaped)
        else:
            y_score = model.predict_proba(X_test_std)[:, 1]

        # Calculate and plot ROC curve
        fpr, tpr, _ = roc_curve(y_test, y_score)
        roc_auc_value = auc(fpr, tpr)

        plt.plot(fpr, tpr, color=colors[name],
                label=f'ROC curve (AUC = {roc_auc_value:.2f})')
        plt.plot([0, 1], [0, 1], color='navy', 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(f'{name} ROC Curve (Best Model)')
        plt.legend(loc='lower right')
        plt.grid(True)
        plt.show()

plot_roc_curves(features_test_all_std,labels_test_all)
```

```
Plotting ROC curves...
```






```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
```



1.14 Summary of Key Metrics

```
def display_mean_metrics(metrics_lists):

    metric_columns = ['TP', 'TN', 'FP', 'FN', 'TPR', 'TNR', 'FPR', 'FNR',
                      'Precision', 'F1_measure', 'Accuracy', 'Error_rate', 'BACC',
                      'TSS', 'HSS', 'Brier_score', 'AUC', 'Acc_by_package_fn']

    # Calculate mean metrics
    avg_metrics = {name: np.mean(metrics, axis=0)
                   for name, metrics in metrics_lists.items()}
    df = pd.DataFrame(avg_metrics, index=metric_columns)

    # Display summary of key metrics
    key_metrics = ['Accuracy', 'Precision', 'F1_measure', 'AUC', 'BACC']
    summary_df = df.loc[key_metrics]

    print("\nSummary of Key Metrics:")
    print("-" * 100)
    print(summary_df.round(3).to_string())
    print("-" * 100)

display_mean_metrics(metrics_lists)
```

```
Summary of Key Metrics:
------------------------------------------
              RF    SVM   LSTM
Accuracy    0.733  0.731  0.722
Precision   0.675  0.692  0.646
F1_measure  0.625  0.604  0.627
AUC         0.812  0.812  0.803
BACC        0.705  0.695  0.702
------------------------------------------
```

➢ Comparison based on the metrics

- The Random Forest model stands out as the best overall performer, consistently achieving the highest accuracy than SVM and LSTM models, which indicates better overall prediction capabilities.
    - Best Balanced Metrics Observed across the Folds.
    - Highest BACC indicating good performance across both the classes.
    - Strong F1-score, showing good balance between precision and recall.

- ROC Curve Analysis:
    - The RF's ROC curve shows good early true positive rate gains with relatively low false positive rates
    - The curve is smoother than the LSTM's, suggesting more stable predictions
    - Matches SVM's AUC while showing better performance in certain operating points.
- While SVM comes very close in performance, and LSTM shows competitive results, the Random Forest model would be the recommended choice for this diabetes prediction task due to its slightly better overall performance and practical advantages in implementation and maintenance.
- Training model on huge data might change the performance as it would give a clear pattern to work on.

➢ Conclusion:

The comparative analysis reveals that while all three models achieve satisfactory performance in diabetes prediction, the Random Forest classifier offers the best balance of accuracy and computational efficiency. The LSTM model, while competitive, requires more computational resources without providing significant performance improvements for this particular dataset

- **Referral Links**

  https://github.com/Rajat-njit/Pednekar_Rajat_Classifier.git