Name: Mahum Abid
NJIT UCID: ma2746
Email: ma2746@njit.edu
Option: 5

Final Term Project Report

Implementation of K-Nearest Neighbor, Random Forest, SVM and LSTM to Predict presence of Diabetes

Mahum Abid

GitHub Repository - https://github.com/mahumabid/Final_Project_DM

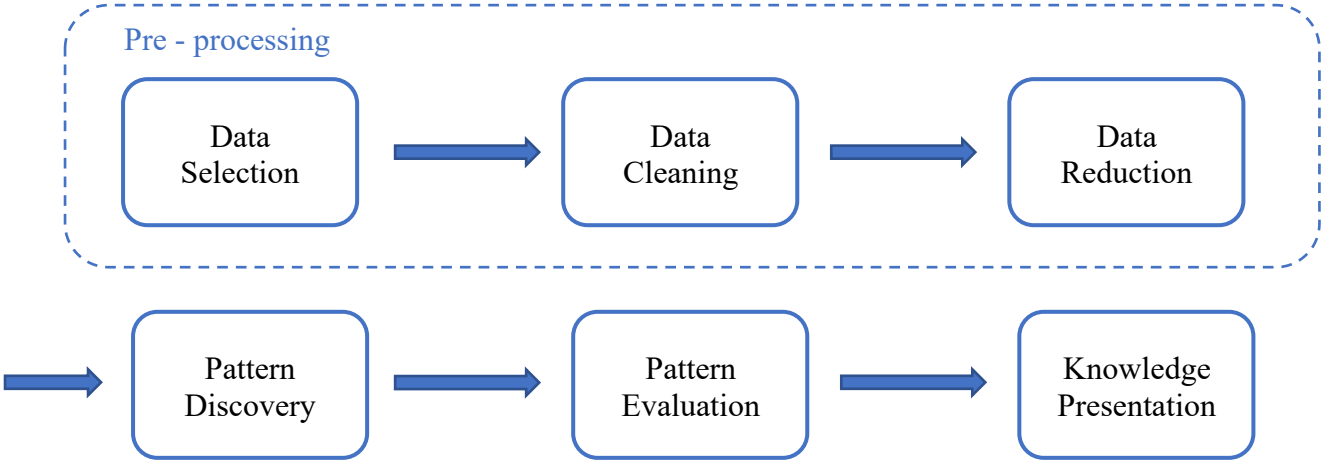
Introduction

Data Mining

Data mining is a process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems.
It involves extraction of interesting information or patterns from data in large databases.
Data mining is a multidisciplinary field that incorporates elements from different fields such as:

- Artificial Intelligence
- Machine Learning
- Data Visualization
- Statistics
- Pattern Recognition

Data Mining Process



Data Mining Main Functions

Data Analysis & Exploration:

- Statistical analysis.
- Visualization & Presentation

Pattern Discovery & Descriptive Modeling:

- Cluster analysis
- Association rules
- Sequential patterns

Predictive Analytics/Modeling (discover the future):

- Classification
- Regression
- Temporal analysis, timeseries

- Anomaly detection
- Deep Learning

Problem Statement

The goal of this project is to Implement 3 different supervised data mining (classification) algorithms and 1 deep learning algorithm in Python using data mining techniques to predict whether a patient has diabetes, based on certain diagnostic measurements available in the dataset. The project also aims to use 10-Fold cross validation to calculate various classifier performance metrics and then compare each classifier.

Selection of Algorithms

For this project, I have chosen total of 4 algorithms as this is my first time implementing deep learning algorithm, LSTM.
Below is the list of algorithms that I will be implementing in this project:

K-Nearest Neighbor (KNN) Classifier

K-nearest neighbor classifier is a well-known and classical algorithm in data mining. The algorithm classifies unlabeled objects based on the majority "class" of nearest neighbor. It learns from the training data sets and determines the K-nearest neighbors by calculating Euclidean distance of each neighbor. It then selects neighbors with minimum distance from unlabeled test object. In this algorithm unlabeled neighbors are classified according to the rule of majority.

Random Forests

Random forest, as the name implies, is a large number of collections of Classification and Regression Trees (CART, a binary decision tree). It is an ensemble algorithm used for classification and regression Random Forest works by growing large number of trees during training of the data and selecting the majority of classification label.

Support Vector Machines

Support vector machines (SVMs)) are supervised learning models with associated learning algorithms that analyze the data for classification and regression.
A support vector machine constructs a hyperplane or set of hyperplanes in a high-dimensional space, which can be used for classification or regression.

Long-Short Term Memory (LSTM)

LSTM is an artificial recurrent neural network (RNN) used in deep learning that is capable of learning order dependence in sequence prediction problems. LSTM has feedback connections and can process entire sequences of data.
LSTM has layers which consists of set of recurrently connected blocks, known as memory blocks. Each memory block contains one or more recurrently connected memory cells and three multiplicative units – the input, output and forget gates.

Dataset

The dataset used for this project is downloaded from Kaggle. This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The aim of this dataset is to predict whether a patient has diabetes, based on certain diagnostic measurements included in the dataset. This dataset has been shared on Kaggle by placing several constraints from a larger database. This data has information of patients that are females and at least 21 years old of Pima Indian heritage.

Dataset attributes:

- Number of times pregnant
- Plasma glucose concentration
- Blood pressure
- Skin fold thickness (mm)
- Insulin
- Body mass index
- Diabetes Pedigree Function
- Age in years
- Outcome variable [0, 1]

Source Code

How to Run the Source Code

- Ensure that the source code files, and the data set are in the same folder to avoid issues with path for loading the data.
- Ensure that the required packages and software are already installed and are updated to at least required versions.
- Open the file in PyCharm or any other IDE and run the source code.

How to Run the Source Code

- Ensure that the source code files, and the data set are in the same folder to avoid issues with path for loading the data
- Ensure that the required packages and software are already installed and are updated to at least required versions.
- Open the file in PyCharm or any other IDE and run the source code.

Using KNN, SVM, RF and LSTM To Predict Diabetes

Goal

"My project aims to implement a variety of machine learning classification algorithms, along with a deep learning model, to predict the likelihood of a patient having diabetes. This prediction is based on specific diagnostic measurements provided in the dataset."

Importing the packages and libraries that are required for the project

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")


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

from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
```

Implementation

Load Dataset from CSV file

Here, first we use the pandas 'read_csv' library to read the dataset from csv file.

We need to have the source code file and dataset csv file in the same folder for code to work. Otherwise, we need to give the specific path of the dataset to load the data.

Data Preprocessing

In next step, we need to check the data statistics to check the quality of data and to understand what kind of preprocessing will be required for the data.

Checking data statistics:
Checking data types of the attributes:

Loading Data And Preprocessing

```
diab = pd.read_csv('diabetes.csv')
diab.describe()
```

9]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

```
diab.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

Checking first 5 rows:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

This gives us idea about how our data looks.

Data Impurities

Here we can see that there are some attributes which have 0 values in the dataset. For following attributes zero value does not make sense,

- Glucose
- Blood Pressure
- Skin Thickness
- Insulin
- BMI

Any person cannot live with zero values for any of the above attributes. This shows us that we have some data impurities.

Now, to solve this type of data impurities we can treat the data with either of following;

Remove the rows with data impurities.

Replace such values with representative value for attribute such as mean or median values.

Here, for this project I have chosen to replace the zero values with the median values of the attributes after analyzing the data distribution. We cannot remove data rows as this would reduce our dataset which is already limited with only 768 rows.

Separating features and output label

Next, we split the dataset into dependent output label and the independent attributes so that we can train our algorithms accordingly.

Data Visualization

Data visualization is important step in the process of data mining as it enables us to visualize patterns in our dataset

Count plot for checking Data Imbalance

First, we plot the count plot to check the data imbalance between the output labels in our dataset

Separating The Dataset into Features and Output label

```
# Feature and label separation
features = diab.iloc[:, :-1]
labels = diab.iloc[:, -1]
```

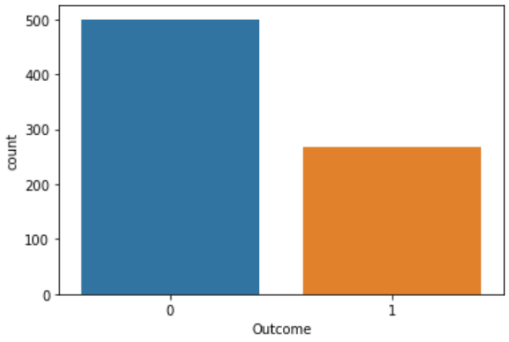
Data Visualization

In the dataset, an observable data imbalance exists in the target label, where the number of patients without diabetes is twice the number of patients with diabetes.

Now, we have two potential approaches: one involves addressing the data imbalance in the train dataset that we will generate, while the other entails employing stratified sampling in the train-test split. Additionally, we will use stratified k-fold cross-validation to maintain a similar label ratio in both the training and testing sets.

For this project, we will adopt the strategy of stratified sampling and apply stratified k-fold cross-validation.

```
# Visualizing the distribution of the target variable
sns.countplot(labels, label="Count")
plt.show()

# Checking for data imbalance
positive_outcomes, negative_outcomes = labels.value_counts()
total_samples = labels.count()

print('----------Checking for Data Imbalance------------')
print('Number of Positive Outcomes: ', positive_outcomes)
print('Percentage of Positive Outcomes: {}%'.format(round((positive_outcomes / total_samples) * 100, 2)))
print('Number of Negative Outcomes : ', negative_outcomes)
print('Percentage of Negative Outcomes: {}%'.format(round((negative_outcomes / total_samples) * 100, 2)))
print('\n')
```




```
----------Checking for Data Imbalance------------
Number of Positive Outcomes:  500
Percentage of Positive Outcomes: 65.1%
Number of Negative Outcomes :  268
Percentage of Negative Outcomes: 34.9%
```

Here, we can see that the target label has data imbalance as number of patients without diabetes is twice the number of patients with diabetes.

Now, we have two options, either fix the data imbalance in train dataset that we will create or ensure that we use stratified sampling in train test split and we use stratified cross validation to ensure the data in training and testing sets has similar ratio of labels.

In our dataset we have following distribution of outcome labels;

- Number of Positive Outcomes: 500
- Percentage of Positive Outcomes: 65.1% Number of Negative Outcomes: 268
- Percentage of Positive Outcomes: 34.9%

Heatmap for checking for Correlation between attributes.

The heatmap helps us to check the correlation between attributes of our dataset

Checking for Correlation between attributes

```
# Creating a correlation matrix and displaying it using a heatmap
fig, axis = plt.subplots(figsize=(8, 8))
correlation_matrix = features.corr()
sns.heatmap(correlation_matrix, annot=True, linewidths=.5, fmt='.2f', ax=axis)
plt.show()
```

Here, we can see that highest correlation is between two pairs

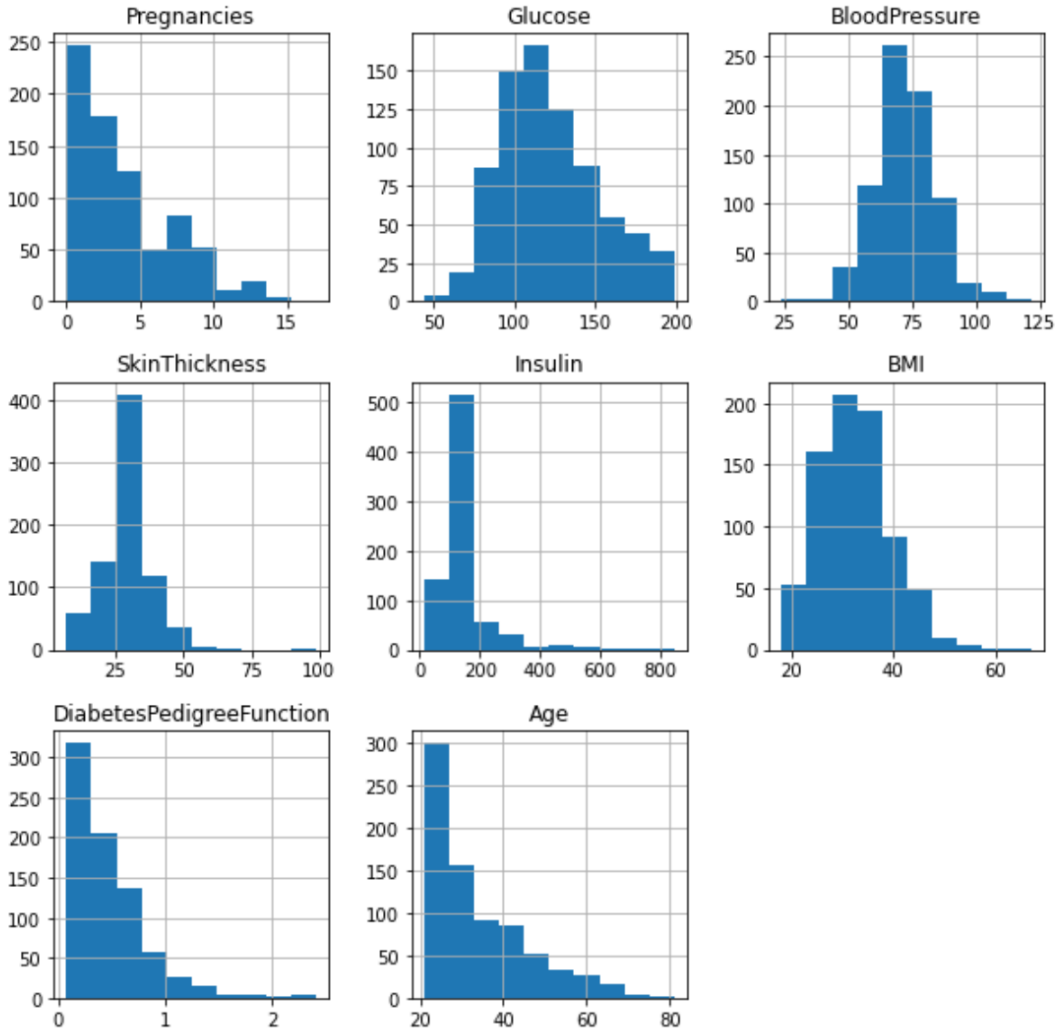
- Correlation between Age and Pregnancies is 0.54
- Correlation between BMI and Skin Thickness is 0.54

We can see that most of the attributes are not highly correlated with each other. Hence, we can use this set of attributes for our project

Histogram to see the distribution of values for each attribute.

Visualize the distribution of values for each attribute by plotting histograms.

```
features.hist(figsize=(10, 10))
plt.show()
```



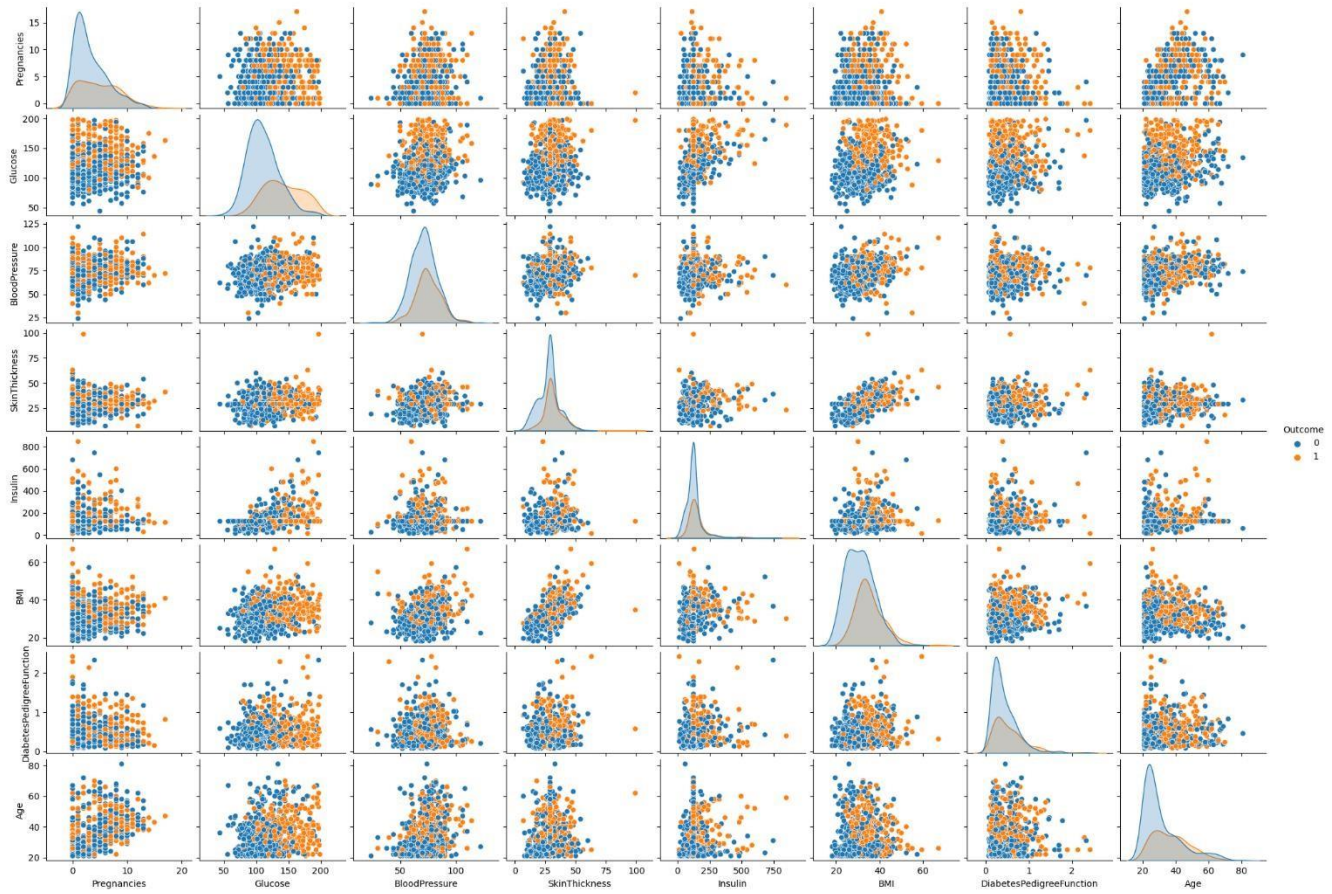
From the above plot we can make following observations:

- The distribution of Glucose and Blood Pressure is somewhat symmetric
- The distribution of remaining attributes is skewed towards one side

Pair plot to plot multiple pairwise bi-variate distributions

Generate a pairplot to visualize multiple pairwise bivariate distributions within our dataset.

```
# Creating a pair plot with a hue based on the 'Outcome' column
sns.pairplot(diab, hue='Outcome')
plt.show()
```



This pair plot shows us the pairwise bi-variate distribution of our dataset. This is helpful for us to understand the distribution of attributes and outcome labels in our dataset

Train-Test Data Split

The train-test data split is a technique for evaluating the performance of our supervised learning algorithms that involves taking our dataset and dividing it into two subsets.

The first subset of the data will be used to fit the algorithms we have chosen. The second subset of dataset will be kept separate for the purpose of testing our models on the dataset that is completely new to the algorithms. This second dataset is also referred to as the test dataset.

Here, to split the dataset in training and testing dataset we are using stratified sampling to ensure that the distribution of outcome labels remains same in both the datasets. This helps us with the issue of data imbalance.

In our project, we will be using the training set for 10-fold cross validation and further, we will use test set to plot ROC curve for each algorithm.

Train Test Data Split

```
# Perform train-test split with a 10% test size and stratification
features_train_all, features_test_all, labels_train_all, labels_test_all = train_test_split(features, labels, test_size=0.1,

# 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)
```

Standardization of Data

We have seen that our data attributes differ in statistics from each other. Here, we can use standardization so that the mean of our attributes will be zero and the standard deviation will be one.

In this project, I have used the below formula to standardize the data,

$$z = \frac{X - \mu}{\sigma}$$

Normalize the training dataset to enhance model performance.

Normalize the training dataset to enhance model performance.

The normalization process involves subtracting the mean from each value and then dividing by the standard deviation. This results in normalized attributes with a mean of 0 and a standard deviation of 1.

```
# Standardize features for training set
features_train_all_std = (features_train_all - features_train_all.mean()) / features_train_all.std()

# Standardize features for testing set
features_test_all_std = (features_test_all - features_test_all.mean()) / features_test_all.std()
```

```
features_train_all_std.describe()
```

91]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
count	6.910000e+02	6.910000e+02	6.910000e+02	6.910000e+02	6.910000e+02	6.910000e+02	6.910000e+02	6.910000e+02
mean	-5.141409e-18	9.897212e-17	-5.655550e-17	5.141409e-18	1.317486e-16	-6.169691e-17	-2.313634e-17	1.568130e-16
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
min	-1.171272e+00	-2.565904e+00	-3.968358e+00	-2.544585e+00	-1.444826e+00	-2.063325e+00	-1.174590e+00	-1.058467e+00
25%	-8.708459e-01	-7.237508e-01	-6.337132e-01	-4.843358e-01	-2.026905e-01	-7.104908e-01	-6.757402e-01	-8.065493e-01
50%	-2.699927e-01	-1.645256e-01	-6.438357e-02	-2.650264e-02	-2.026905e-01	-3.407390e-02	-3.147111e-01	-3.866867e-01
75%	6.312872e-01	6.249688e-01	5.862789e-01	3.168722e-01	-1.467385e-01	5.847756e-01	4.777553e-01	6.209834e-01
max	3.034700e+00	2.467122e+00	4.002257e+00	7.985577e+00	7.865593e+00	4.974289e+00	5.842258e+00	3.979884e+00

As observed in the table above, the mean of each attribute is approximately 0, and the standard deviation is 1.

We can see from above table that the mean of each attribute in standardized dataset is close to 0 and standard deviation is 1.

Define the necessary function for model fitting and metric calculation.

```
def calc_metrics(confusion_matrix):
    TP, FN = confusion_matrix[0][0], confusion_matrix[0][1]
    FP, TN = confusion_matrix[1][0], confusion_matrix[1][1]

    TPR = TP / (TP + FN)
    TNR = TN / (TN + FP)
    FPR = FP / (TN + FP)
    FNR = FN / (TP + FN)
    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
    TSS = TPR - FPR
    HSS = 2 * (TP * TN - FP * FN) / ((TP + FN) * (FN + TN) + (TP + FP) * (FP + TN))

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

```python
def get_metrics(model, X_train, X_test, y_train, y_test, LSTM_flag):
    def calc_metrics(conf_matrix):
        TP, FN = conf_matrix[0][0], conf_matrix[0][1]
        FP, TN = conf_matrix[1][0], conf_matrix[1][1]

        TPR = TP / (TP + FN)
        TNR = TN / (TN + FP)
        FPR = FP / (TN + FP)
        FNR = FN / (TP + FN)
        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
        TSS = TPR - FPR
        HSS = 2 * (TP * TN - FP * FN) / ((TP + FN) * (FN + TN) + (TP + FP) * (FP + TN))

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

    metrics = []

    if LSTM_flag == 1:
        # Convert data to numpy array
        Xtrain, Xtest, ytrain, ytest = map(np.array, [features_train, features_test, labels_train, labels_test])

        # Reshape data
        shape = Xtrain.shape
        Xtrain_reshaped = Xtrain.reshape(len(Xtrain), shape[1], 1)
        Xtest_reshaped = Xtest.reshape(len(Xtest), shape[1], 1)

        model.fit(Xtrain_reshaped, ytrain, epochs=50, validation_data=(Xtest_reshaped, ytest), verbose=0)
        lstm_scores = model.evaluate(Xtest_reshaped, ytest, verbose=0)
        predict_prob = model.predict(Xtest_reshaped)
        pred_labels = predict_prob > 0.5
        pred_labels_1 = pred_labels.astype(int)
        matrix = confusion_matrix(ytest, pred_labels_1, labels=[1, 0])
        lstm_brier_score = brier_score_loss(ytest, predict_prob)
        lstm_roc_auc = roc_auc_score(ytest, predict_prob)
        metrics.extend(calc_metrics(matrix))
        metrics.extend([lstm_brier_score, lstm_roc_auc, lstm_scores[1]])

    elif LSTM_flag == 0:
        model.fit(features_train, labels_train)
        predicted = model.predict(features_test)
        matrix = confusion_matrix(labels_test, predicted, labels=[1, 0])
        model_brier_score = brier_score_loss(labels_test, model.predict_proba(features_test)[:, 1])
        model_roc_auc = roc_auc_score(y_test, model.predict_proba(features_test)[:, 1])
        metrics.extend(calc_metrics(matrix))
        metrics.extend([model_brier_score, model_roc_auc, model.score(features_test, labels_test)])

    return metrics
```

Model Fitting and Parameter Tuning

Here, I have decided to select following Classification algorithms.

- K-Nearest Neighbor
- Random Forest
- Support Vector Machine

For Deep learning algorithm, I have decided to use LSTM

- Long Short-Term Memory

Since, this is the first time I am implementing any Deep Learning algorithm, I have decided to select one additional classifier for comparison.

KNN Model Fitting

Parameter Tuning for KNN

```
# Define KNN parameters for grid search
knn_parameters = {"n_neighbors": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]}

# Create KNN model
knn_model = KNeighborsClassifier()

# Perform grid search with cross-validation
knn_cv = GridSearchCV(knn_model, knn_parameters, cv=10, n_jobs=-1)
knn_cv.fit(features_train_all_std, labels_train_all)

# Print the best parameters found by GridSearchCV
print("\nBest Parameters for KNN based on GridSearchCV: ", knn_cv.best_params_)
print('\n')
```

```
Best Parameters for KNN based on GridSearchCV:  {'n_neighbors': 13}
```

```
# Extract the best value for 'n_neighbors' from the grid search results
best_n_neighbors = knn_cv.best_params_['n_neighbors']
```

Here, I have used Grid Search CV library from sklearn package to get best parameters for KNN algorithm.

I have used the values ranging from 1 to 15 for the parameter number of nearest neighbors.

Based on Grid Search best parameter value for number of nearest neighbors is 13 for our dataset.

This value can change if we change the data split.

Random Forest Model Fitting

Here, I have used Grid Search CV library from sklearn package to get best parameters for Random Forest algorithm. Also, we are tuning the parameters number of estimators and minimum sample splits for the algorithm.

RF Parameter Tuning

```
# Define Random Forest parameters for grid search
param_grid_rf = {
    "n_estimators": [10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
    "min_samples_split": [2, 4, 6, 8, 10]
}

# Create Random Forest model
rf_classifier = RandomForestClassifier()

# Perform grid search with cross-validation
grid_search_rf = GridSearchCV(estimator=rf_classifier, param_grid=param_grid_rf, cv=10, n_jobs=-1)
grid_search_rf.fit(features_train_all_std, labels_train_all)

# Display the best parameters from the grid search
best_rf_params = grid_search_rf.best_params_
print("\nBest Parameters for Random Forest based on GridSearchCV: ", best_rf_params)
print('\n')

# Extract the best values for 'min_samples_split' and 'n_estimators'
min_samples_split = best_rf_params['min_samples_split']
n_estimators = best_rf_params['n_estimators']
```

```
Best Parameters for Random Forest based on GridSearchCV:  {'min_samples_split': 4, 'n_estimators': 80}
```

Based on Grid Search best parameter value for number of estimators and minimum sample splits for the algorithm are 4 and 80 respectively.

These values can change if we change the data split. In the project, I am using values that are generated in that particular run.

SVM Model Fitting

Similar to other algorithms, I have used Grid Search CV library from sklearn package to get best parameters for SVM algorithm.

For SVM, I have decided to use linear kernel for our project. Hence, in parameter tuning I am tuning the value of C for the algorithm.

SVM Parameter Tuning

Here, I am using 'linear' kernel for SVC for this project

```
# Define Support Vector Machine parameters for grid search
param_grid_svc = {"kernel": ["linear"], "C": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}

# Create Support Vector Machine model
svc_classifier = SVC()

# Perform grid search with cross-validation
grid_search_svc = GridSearchCV(estimator=svc_classifier, param_grid=param_grid_svc, cv=10, n_jobs=-1)
grid_search_svc.fit(features_train_all_std, labels_train_all)

# Display the best parameters from the grid search
best_svc_params = grid_search_svc.best_params_
print("\nBest Parameters for Support Vector Machine based on GridSearchCV: ", best_svc_params)
print('\n')

# Extract the best value for 'C'
C_value = best_svc_params['C']
```

```
Best Parameters for Support Vector Machine based on GridSearchCV:  {'C': 3, 'kernel': 'linear'}
```

Based on Grid Search best parameter value for C is 3.

These values can change if we change the data split. In the project, I am using values that are generated in that particular run.

LSTM Model Fitting

For LSTM, I am using Keras and Tensorflow packages. For creating LSTM model, I have added 2 layers, first layer is LSTM layer with activation as relu and then for output layer I am using dense layer with activation function as sigmoid function. The sigmoid function helps us with generating output for binary classification.

Further, I am converting predicted probabilities into predicted labels by classifying probabilities greater than 0.5 as 1 label and rest as 0 label.

Performance metrics

In this step we evaluate the performance of our algorithms by calculating various performance parameters for each algorithm in each iteration of 10-fold cross validation.

In this project, I have calculated performance metrics that were discussed in the class. The list of performance parameters calculated is given below

True Positive (TP)

It is the number of positive examples correctly predicted by the classification model.

True Negative (TN)

It is the number of negative examples correctly predicted by the classification model.

False positive (FP)

It is the number of negative examples wrongly predicted as positive by the classification model.

False negative (FN)

It is the number of positive examples wrongly predicted as negative by the classification model.

True positive rate (TPR) or sensitivity

It is the fraction of positive examples predicted correctly by the model.

$$TPR = \frac{TP}{TP + FN}$$

True negative rate (TNR) or specificity

It is the fraction of negative examples predicted correctly by the model.

$$TNR = \frac{TN}{+ FP} TN$$

False positive rate (FPR)

It is the fraction of negative examples predicted as positive.

$$FPR = \frac{FP}{TN + FP}$$

False negative rate (FNR)

It is the fraction of positive examples predicted as negative.

$$FNR = \frac{FN}{TP + FN}$$

Precision (p)

$$p = \frac{TP}{TP + FP}$$

F1 Measure (F1)

$$F1 = \frac{(2 \times TP)}{(2 \times TP + FP + FN)}$$

Accuracy (Acc)

It is the measure of how many of the outcomes were correctly classified by the classifier

$$Acc = \frac{(TP + TN)}{(TP + FP + FN + TN)}$$

Error Rate (Err)

It is the ratio of wrongly classified labels to total number of labels

$$Err = \frac{(FP + FN)}{(TP + FP + FN + TN)}$$

Confusion Matrix

It is also known as error matrix.

	Actual Positive	**Actual Negative**
Predicted Positive	True Positive	False Positive
Predicted Negative	False Negative	True Negative

Balanced Accuracy (BACC)

It measures the average sensitive (known as the true positive rate) and the specificity (known as false positive rate)

$$BACC = \frac{TPR + TNR}{2}$$

True Skill Statistics (TSS)

It measures the difference between the recall minus the probability of false detection.

$$TSS = \frac{TP}{TP + FN} - \frac{FP}{FP + TN}$$

Heidke Skill Score (HSS)

It measures the fractional prediction over random prediction.

$$HSS = \frac{2 \times (TP \times TN - FP \times FN)}{(TP + FN) \times (FN + TN) + (TP + FP) \times (FP + TN)}$$

Brier Score (BS)

It is the mean squared error between the expected probabilities and the predicted probabilities

$$BS = \frac{1}{m} \sum_{n=1}^{m} (y_n - \hat{y}_n)^2$$

ROC Curve

Receiver Operating Characteristic (ROC) curve is a plot that shows the performance our algorithms at all classification thresholds. This curve plots the parameters:

- True Positive Rate
- False Positive Rate

AUC: Area Under the ROC

It is the measure of performance across all possible classification thresholds. It measures the area under the ROC curve.

10-Fold Stratified Cross-Validation

First, we split the dataset into 10 parts. By using stratified Cross Validation, we ensure similar distribution of outcome labels in all 10 splits.

Next, we use one part for testing and remaining 9 parts for training the model. We repeat this process for 10 iterations to get performance of our algorithms over 10 different datasets in 10 iterations.

i	Fold1	Fold2	Fold3	Fold4	Fold5	Fold6	Fold7	Fold8	Fold9	Fold10	Metrics
1	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M1
2	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M2
3	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M3
4	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M4
5	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M5
6	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M6
7	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M7
8	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M8
9	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M9
10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	N/10	M10

Legend

Testing Data

Training Data

We can calculate performance metrics for the algorithms over each iteration and then take the average to get overall performance of the algorithm.

Comparing the classifiers with selected parameters by using 10-Fold Stratified Cross-Validation to calculate all metrics

Implementing 10-Fold Stratified Cross-Validation

In this project, I will be using the training data set for validation as well using Startefied 10-Fold Cross Validation

```
# Define Stratified K-Fold cross-validator
cv_stratified = StratifiedKFold(n_splits=10, shuffle=True, random_state=21)
```

```
# Initialize metric columns
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']

# Initialize metrics lists for each algorithm
knn_metrics_list, rf_metrics_list, svm_metrics_list, lstm_metrics_list = [], [], [], []

C = 1.0

# 10 Iterations of 10-fold cross-validation
for iter_num, (train_index, test_index) in enumerate(cv_stratified.split(features_train_all_std, labels_train_all), start=1):
    # KNN Model
    knn_model = KNeighborsClassifier(n_neighbors=best_n_neighbors)
    # Random Forest Model
    rf_model = RandomForestClassifier(min_samples_split=min_samples_split, n_estimators=n_estimators)
    # SVM Classifier Model
    svm_model = SVC(C=C, kernel='linear', probability=True)
    # LSTM model
    lstm_model = Sequential()
    lstm_model.add(LSTM(64, activation='relu', batch_input_shape=(None, 8, 1), return_sequences=False))
    lstm_model.add(Dense(1, activation='sigmoid'))
    # Compile model
    lstm_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    # Split data into training and testing sets
    features_train, features_test = features_train_all_std.iloc[train_index, :], features_train_all_std.iloc[test_index, :]
    labels_train, labels_test = labels_train_all[train_index], labels_train_all[test_index]

    # Get metrics for each algorithm
    knn_metrics = get_metrics(knn_model, features_train, features_test, labels_train, labels_test, 0)
    rf_metrics = get_metrics(rf_model, features_train, features_test, labels_train, labels_test, 0)
    svm_metrics = get_metrics(svm_model, features_train, features_test, labels_train, labels_test, 0)
    lstm_metrics = get_metrics(lstm_model, features_train, features_test, labels_train, labels_test, 1)

    # Append metrics to respective lists
    knn_metrics_list.append(knn_metrics)
    rf_metrics_list.append(rf_metrics)
    svm_metrics_list.append(svm_metrics)
    lstm_metrics_list.append(lstm_metrics)

    # Create a DataFrame for all metrics
    metrics_all_df = pd.DataFrame([knn_metrics, rf_metrics, svm_metrics, lstm_metrics],
                                  columns=metric_columns, index=['KNN', 'RF', 'SVM', 'LSTM'])

    # Display metrics for all algorithms in each iteration
    print('\nIteration {}: \n'.format(iter_num))
    print('\n----- Metrics for all Algorithms in Iteration {} -----\n'.format(iter_num))
    print(metrics_all_df.round(decimals=2).T)
    print('\n')
```

10-Fold cross validation ensures that we check the performance of our model over whole training dataset. This resolves issues of 1 test set being different than rest of the dataset.

Iteration 1 results

-

----- Metrics for all Algorithms in Iteration 1 -----

	KNN	RF	SVM	LSTM
TP	14.00	15.00	13.00	13.00
TN	39.00	39.00	37.00	35.00
FP	6.00	6.00	8.00	10.00
FN	11.00	10.00	12.00	12.00
TPR	0.56	0.60	0.52	0.52
TNR	0.87	0.87	0.82	0.78
FPR	0.13	0.13	0.18	0.22
FNR	0.44	0.40	0.48	0.48
Precision	0.70	0.71	0.62	0.57
F1_measure	0.62	0.65	0.57	0.54
Accuracy	0.76	0.77	0.71	0.69
Error_rate	0.24	0.23	0.29	0.31
BACC	0.71	0.73	0.67	0.65
TSS	0.43	0.47	0.34	0.30
HSS	0.45	0.48	0.35	0.30
Brier_score	0.17	0.17	0.17	0.19
AUC	0.80	0.81	0.79	0.78
Acc_by_package_fn	0.76	0.77	0.71	0.69

Iteration 2 results

Iteration 2:

----- Metrics for all Algorithms in Iteration 2 -----

	KNN	RF	SVM	LSTM
TP	11.00	11.00	14.00	13.00
TN	37.00	38.00	42.00	33.00
FP	8.00	7.00	3.00	12.00
FN	13.00	13.00	10.00	11.00
TPR	0.46	0.46	0.58	0.54
TNR	0.82	0.84	0.93	0.73
FPR	0.18	0.16	0.07	0.27
FNR	0.54	0.54	0.42	0.46
Precision	0.58	0.61	0.82	0.52
F1_measure	0.51	0.52	0.68	0.53
Accuracy	0.70	0.71	0.81	0.67
Error_rate	0.30	0.29	0.19	0.33
BACC	0.64	0.65	0.76	0.64
TSS	0.28	0.30	0.52	0.27
HSS	0.29	0.32	0.55	0.27
Brier_score	0.17	0.17	0.14	0.19
AUC	0.79	0.81	0.86	0.75
Acc_by_package_fn	0.70	0.71	0.81	0.67

Iteration 3 results

```
Iteration 3:


----- Metrics for all Algorithms in Iteration 3 -----
```

	KNN	RF	SVM	LSTM
TP	11.00	11.00	10.00	9.00
TN	34.00	32.00	34.00	37.00
FP	11.00	13.00	11.00	8.00
FN	13.00	13.00	14.00	15.00
TPR	0.46	0.46	0.42	0.38
TNR	0.76	0.71	0.76	0.82
FPR	0.24	0.29	0.24	0.18
FNR	0.54	0.54	0.58	0.62
Precision	0.50	0.46	0.48	0.53
F1_measure	0.48	0.46	0.44	0.44
Accuracy	0.65	0.62	0.64	0.67
Error_rate	0.35	0.38	0.36	0.33
BACC	0.61	0.58	0.59	0.60
TSS	0.21	0.17	0.17	0.20
HSS	0.22	0.17	0.18	0.21
Brier_score	0.21	0.23	0.24	0.22
AUC	0.72	0.68	0.68	0.73
Acc_by_package_fn	0.65	0.62	0.64	0.67

Iteration 4 results

```
Iteration 4:


----- Metrics for all Algorithms in Iteration 4 -----
```

	KNN	RF	SVM	LSTM
TP	13.00	17.00	13.00	13.00
TN	37.00	40.00	41.00	39.00
FP	8.00	5.00	4.00	6.00
FN	11.00	7.00	11.00	11.00
TPR	0.54	0.71	0.54	0.54
TNR	0.82	0.89	0.91	0.87
FPR	0.18	0.11	0.09	0.13
FNR	0.46	0.29	0.46	0.46
Precision	0.62	0.77	0.76	0.68
F1_measure	0.58	0.74	0.63	0.60
Accuracy	0.72	0.83	0.78	0.75
Error_rate	0.28	0.17	0.22	0.25
BACC	0.68	0.80	0.73	0.70
TSS	0.36	0.60	0.45	0.41
HSS	0.37	0.61	0.49	0.43
Brier_score	0.16	0.14	0.14	0.17
AUC	0.81	0.86	0.87	0.82
Acc_by_package_fn	0.72	0.83	0.78	0.75

Iteration 5 results

Iteration 5:

----- Metrics for all Algorithms in Iteration 5 -----

	KNN	RF	SVM	LSTM
TP	10.00	9.00	6.00	9.00
TN	41.00	40.00	43.00	40.00
FP	4.00	5.00	2.00	5.00
FN	14.00	15.00	18.00	15.00
TPR	0.42	0.38	0.25	0.38
TNR	0.91	0.89	0.96	0.89
FPR	0.09	0.11	0.04	0.11
FNR	0.58	0.62	0.75	0.62
Precision	0.71	0.64	0.75	0.64
F1_measure	0.53	0.47	0.38	0.47
Accuracy	0.74	0.71	0.71	0.71
Error_rate	0.26	0.29	0.29	0.29
BACC	0.66	0.63	0.60	0.63
TSS	0.33	0.26	0.21	0.26
HSS	0.36	0.29	0.24	0.29
Brier_score	0.18	0.19	0.18	0.18
AUC	0.77	0.75	0.81	0.78
Acc_by_package_fn	0.74	0.71	0.71	0.71

Iteration 6 results

Iteration 6:

----- Metrics for all Algorithms in Iteration 6 -----

	KNN	RF	SVM	LSTM
TP	17.00	17.00	18.00	19.00
TN	41.00	38.00	40.00	36.00
FP	4.00	7.00	5.00	9.00
FN	7.00	7.00	6.00	5.00
TPR	0.71	0.71	0.75	0.79
TNR	0.91	0.84	0.89	0.80
FPR	0.09	0.16	0.11	0.20
FNR	0.29	0.29	0.25	0.21
Precision	0.81	0.71	0.78	0.68
F1_measure	0.76	0.71	0.77	0.73
Accuracy	0.84	0.80	0.84	0.80
Error_rate	0.16	0.20	0.16	0.20
BACC	0.81	0.78	0.82	0.80
TSS	0.62	0.55	0.64	0.59
HSS	0.64	0.55	0.65	0.57
Brier_score	0.13	0.13	0.12	0.17
AUC	0.91	0.88	0.94	0.84
Acc_by_package_fn	0.84	0.80	0.84	0.80

Iteration 7 results

Iteration 7:

----- Metrics for all Algorithms in Iteration 7 -----

	KNN	RF	SVM	LSTM
TP	16.00	17.00	13.00	16.00
TN	39.00	39.00	37.00	34.00
FP	6.00	6.00	8.00	11.00
FN	8.00	7.00	11.00	8.00
TPR	0.67	0.71	0.54	0.67
TNR	0.87	0.87	0.82	0.76
FPR	0.13	0.13	0.18	0.24
FNR	0.33	0.29	0.46	0.33
Precision	0.73	0.74	0.62	0.59
F1_measure	0.70	0.72	0.58	0.63
Accuracy	0.80	0.81	0.72	0.72
Error_rate	0.20	0.19	0.28	0.28
BACC	0.77	0.79	0.68	0.71
TSS	0.53	0.58	0.36	0.42
HSS	0.54	0.58	0.37	0.41
Brier_score	0.15	0.15	0.15	0.17
AUC	0.86	0.87	0.86	0.81
Acc_by_package_fn	0.80	0.81	0.72	0.72

Iteration 8 results

Iteration 8:

----- Metrics for all Algorithms in Iteration 8 -----

	KNN	RF	SVM	LSTM
TP	13.00	13.00	10.00	10.00
TN	41.00	41.00	42.00	42.00
FP	4.00	4.00	3.00	3.00
FN	11.00	11.00	14.00	14.00
TPR	0.54	0.54	0.42	0.42
TNR	0.91	0.91	0.93	0.93
FPR	0.09	0.09	0.07	0.07
FNR	0.46	0.46	0.58	0.58
Precision	0.76	0.76	0.77	0.77
F1_measure	0.63	0.63	0.54	0.54
Accuracy	0.78	0.78	0.75	0.75
Error_rate	0.22	0.22	0.25	0.25
BACC	0.73	0.73	0.68	0.68
TSS	0.45	0.45	0.35	0.35
HSS	0.49	0.49	0.39	0.39
Brier_score	0.16	0.17	0.17	0.18
AUC	0.83	0.81	0.82	0.78
Acc_by_package_fn	0.78	0.78	0.75	0.75

Iteration 9 results

```
Iteration 9:


----- Metrics for all Algorithms in Iteration 9 -----

                      KNN     RF    SVM   LSTM
TP                  12.00  17.00  16.00  18.00
TN                  38.00  34.00  38.00  36.00
FP                   7.00  11.00   7.00   9.00
FN                  12.00   7.00   8.00   6.00
TPR                  0.50   0.71   0.67   0.75
TNR                  0.84   0.76   0.84   0.80
FPR                  0.16   0.24   0.16   0.20
FNR                  0.50   0.29   0.33   0.25
Precision            0.63   0.61   0.70   0.67
F1_measure           0.56   0.65   0.68   0.71
Accuracy             0.72   0.74   0.78   0.78
Error_rate           0.28   0.26   0.22   0.22
BACC                 0.67   0.73   0.76   0.78
TSS                  0.34   0.46   0.51   0.55
HSS                  0.36   0.45   0.52   0.53
Brier_score          0.17   0.16   0.15   0.15
AUC                  0.80   0.84   0.85   0.85
Acc_by_package_fn    0.72   0.74   0.78   0.78
```

Iteration 10 results

```
Iteration 10:


----- Metrics for all Algorithms in Iteration 10 -----

                      KNN     RF    SVM   LSTM
TP                  16.00  18.00  17.00  17.00
TN                  38.00  38.00  39.00  39.00
FP                   7.00   7.00   6.00   6.00
FN                   8.00   6.00   7.00   7.00
TPR                  0.67   0.75   0.71   0.71
TNR                  0.84   0.84   0.87   0.87
FPR                  0.16   0.16   0.13   0.13
FNR                  0.33   0.25   0.29   0.29
Precision            0.70   0.72   0.74   0.74
F1_measure           0.68   0.73   0.72   0.72
Accuracy             0.78   0.81   0.81   0.81
Error_rate           0.22   0.19   0.19   0.19
BACC                 0.76   0.80   0.79   0.79
TSS                  0.51   0.59   0.58   0.58
HSS                  0.52   0.59   0.58   0.58
Brier_score          0.14   0.12   0.13   0.12
AUC                  0.87   0.91   0.90   0.92
Acc_by_package_fn    0.78   0.81   0.81   0.81
```

All iterations result table

KNN

Metrics for Algorithm 1:

	iter1	iter2	iter3	iter4	iter5	iter6	iter7	iter8	\
TP	14.00	11.00	11.00	13.00	10.00	17.00	16.00	13.00	
TN	39.00	37.00	34.00	37.00	41.00	41.00	39.00	41.00	
FP	6.00	8.00	11.00	8.00	4.00	4.00	6.00	4.00	
FN	11.00	13.00	13.00	11.00	14.00	7.00	8.00	11.00	
TPR	0.56	0.46	0.46	0.54	0.42	0.71	0.67	0.54	
TNR	0.87	0.82	0.76	0.82	0.91	0.91	0.87	0.91	
FPR	0.13	0.18	0.24	0.18	0.09	0.09	0.13	0.09	
FNR	0.44	0.54	0.54	0.46	0.58	0.29	0.33	0.46	
Precision	0.70	0.58	0.50	0.62	0.71	0.81	0.73	0.76	
F1_measure	0.62	0.51	0.48	0.58	0.53	0.76	0.70	0.63	
Accuracy	0.76	0.70	0.65	0.72	0.74	0.84	0.80	0.78	
Error_rate	0.24	0.30	0.35	0.28	0.26	0.16	0.20	0.22	
BACC	0.71	0.64	0.61	0.68	0.66	0.81	0.77	0.73	
TSS	0.43	0.28	0.21	0.36	0.33	0.62	0.53	0.45	
HSS	0.45	0.29	0.22	0.37	0.36	0.64	0.54	0.49	
Brier_score	0.17	0.17	0.21	0.16	0.18	0.13	0.15	0.16	
AUC	0.80	0.79	0.72	0.81	0.77	0.91	0.86	0.83	
Acc_by_package_fn	0.76	0.70	0.65	0.72	0.74	0.84	0.80	0.78	

	iter9	iter10
TP	12.00	16.00
TN	38.00	38.00
FP	7.00	7.00
FN	12.00	8.00
TPR	0.50	0.67
TNR	0.84	0.84
FPR	0.16	0.16
FNR	0.50	0.33
Precision	0.63	0.70
F1_measure	0.56	0.68
Accuracy	0.72	0.78
Error_rate	0.28	0.22
BACC	0.67	0.76
TSS	0.34	0.51
HSS	0.36	0.52
Brier_score	0.17	0.14
AUC	0.80	0.87
Acc_by_package_fn	0.72	0.78

Random Forest – All iterations result table

Metrics for Algorithm 2:

	iter1	iter2	iter3	iter4	iter5	iter6	iter7	iter8	\
TP	15.00	11.00	11.00	17.00	9.00	17.00	17.00	13.00	
TN	39.00	38.00	32.00	40.00	40.00	38.00	39.00	41.00	
FP	6.00	7.00	13.00	5.00	5.00	7.00	6.00	4.00	
FN	10.00	13.00	13.00	7.00	15.00	7.00	7.00	11.00	
TPR	0.60	0.46	0.46	0.71	0.38	0.71	0.71	0.54	
TNR	0.87	0.84	0.71	0.89	0.89	0.84	0.87	0.91	
FPR	0.13	0.16	0.29	0.11	0.11	0.16	0.13	0.09	
FNR	0.40	0.54	0.54	0.29	0.62	0.29	0.29	0.46	
Precision	0.71	0.61	0.46	0.77	0.64	0.71	0.74	0.76	
F1_measure	0.65	0.52	0.46	0.74	0.47	0.71	0.72	0.63	
Accuracy	0.77	0.71	0.62	0.83	0.71	0.80	0.81	0.78	
Error_rate	0.23	0.29	0.38	0.17	0.29	0.20	0.19	0.22	
BACC	0.73	0.65	0.58	0.80	0.63	0.78	0.79	0.73	
TSS	0.47	0.30	0.17	0.60	0.26	0.55	0.58	0.45	
HSS	0.48	0.32	0.17	0.61	0.29	0.55	0.58	0.49	
Brier_score	0.17	0.17	0.23	0.14	0.19	0.13	0.15	0.17	
AUC	0.81	0.81	0.68	0.86	0.75	0.88	0.87	0.81	
Acc_by_package_fn	0.77	0.71	0.62	0.83	0.71	0.80	0.81	0.78	

	iter9	iter10
TP	17.00	18.00
TN	34.00	38.00
FP	11.00	7.00
FN	7.00	6.00
TPR	0.71	0.75
TNR	0.76	0.84
FPR	0.24	0.16
FNR	0.29	0.25
Precision	0.61	0.72
F1_measure	0.65	0.73
Accuracy	0.74	0.81
Error_rate	0.26	0.19
BACC	0.73	0.80
TSS	0.46	0.59
HSS	0.45	0.59
Brier_score	0.16	0.12
AUC	0.84	0.91
Acc_by_package_fn	0.74	0.81

SVM – All iterations result table

Metrics for Algorithm 3:

	iter1	iter2	iter3	iter4	iter5	iter6	iter7	iter8	\
TP	13.00	14.00	10.00	13.00	6.00	18.00	13.00	10.00	
TN	37.00	42.00	34.00	41.00	43.00	40.00	37.00	42.00	
FP	8.00	3.00	11.00	4.00	2.00	5.00	8.00	3.00	
FN	12.00	10.00	14.00	11.00	18.00	6.00	11.00	14.00	
TPR	0.52	0.58	0.42	0.54	0.25	0.75	0.54	0.42	
TNR	0.82	0.93	0.76	0.91	0.96	0.89	0.82	0.93	
FPR	0.18	0.07	0.24	0.09	0.04	0.11	0.18	0.07	
FNR	0.48	0.42	0.58	0.46	0.75	0.25	0.46	0.58	
Precision	0.62	0.82	0.48	0.76	0.75	0.78	0.62	0.77	
F1_measure	0.57	0.68	0.44	0.63	0.38	0.77	0.58	0.54	
Accuracy	0.71	0.81	0.64	0.78	0.71	0.84	0.72	0.75	
Error_rate	0.29	0.19	0.36	0.22	0.29	0.16	0.28	0.25	
BACC	0.67	0.76	0.59	0.73	0.60	0.82	0.68	0.68	
TSS	0.34	0.52	0.17	0.45	0.21	0.64	0.36	0.35	
HSS	0.35	0.55	0.18	0.49	0.24	0.65	0.37	0.39	
Brier_score	0.17	0.14	0.24	0.14	0.18	0.12	0.15	0.17	
AUC	0.79	0.86	0.68	0.87	0.81	0.94	0.86	0.82	
Acc_by_package_fn	0.71	0.81	0.64	0.78	0.71	0.84	0.72	0.75	

	iter9	iter10
TP	16.00	17.00
TN	38.00	39.00
FP	7.00	6.00
FN	8.00	7.00
TPR	0.67	0.71
TNR	0.84	0.87
FPR	0.16	0.13
FNR	0.33	0.29
Precision	0.70	0.74
F1_measure	0.68	0.72
Accuracy	0.78	0.81
Error_rate	0.22	0.19
BACC	0.76	0.79
TSS	0.51	0.58
HSS	0.52	0.58
Brier_score	0.15	0.13
AUC	0.85	0.90
Acc_by_package_fn	0.78	0.81

LSTM – All iterations result table

Metrics for Algorithm 4:

	iter1	iter2	iter3	iter4	iter5	iter6	iter7	iter8	\
TP	13.00	13.00	9.00	13.00	9.00	19.00	16.00	10.00	
TN	35.00	33.00	37.00	39.00	40.00	36.00	34.00	42.00	
FP	10.00	12.00	8.00	6.00	5.00	9.00	11.00	3.00	
FN	12.00	11.00	15.00	11.00	15.00	5.00	8.00	14.00	
TPR	0.52	0.54	0.38	0.54	0.38	0.79	0.67	0.42	
TNR	0.78	0.73	0.82	0.87	0.89	0.80	0.76	0.93	
FPR	0.22	0.27	0.18	0.13	0.11	0.20	0.24	0.07	
FNR	0.48	0.46	0.62	0.46	0.62	0.21	0.33	0.58	
Precision	0.57	0.52	0.53	0.68	0.64	0.68	0.59	0.77	
F1_measure	0.54	0.53	0.44	0.60	0.47	0.73	0.63	0.54	
Accuracy	0.69	0.67	0.67	0.75	0.71	0.80	0.72	0.75	
Error_rate	0.31	0.33	0.33	0.25	0.29	0.20	0.28	0.25	
BACC	0.65	0.64	0.60	0.70	0.63	0.80	0.71	0.68	
TSS	0.30	0.27	0.20	0.41	0.26	0.59	0.42	0.35	
HSS	0.30	0.27	0.21	0.43	0.29	0.57	0.41	0.39	
Brier_score	0.19	0.19	0.22	0.17	0.18	0.17	0.17	0.18	
AUC	0.78	0.75	0.73	0.82	0.78	0.84	0.81	0.78	
Acc_by_package_fn	0.69	0.67	0.67	0.75	0.71	0.80	0.72	0.75	

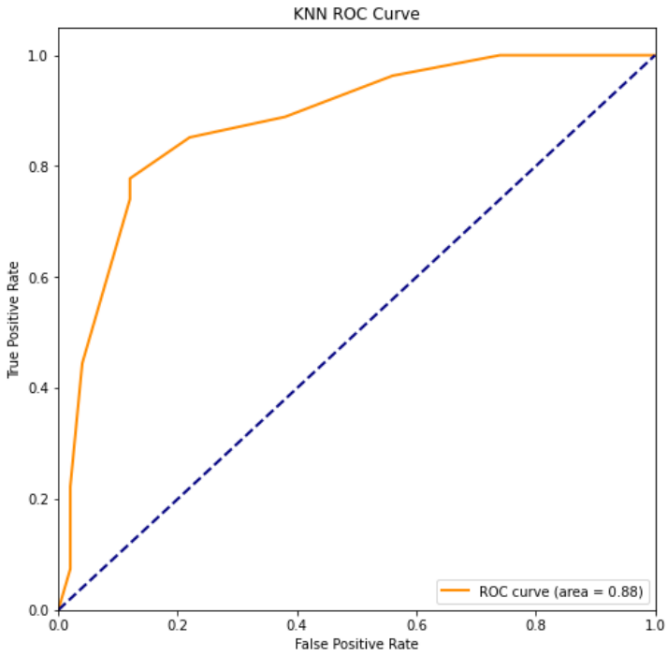
	iter9	iter10
TP	18.00	17.00
TN	36.00	39.00
FP	9.00	6.00
FN	6.00	7.00
TPR	0.75	0.71
TNR	0.80	0.87
FPR	0.20	0.13
FNR	0.25	0.29
Precision	0.67	0.74
F1_measure	0.71	0.72
Accuracy	0.78	0.81
Error_rate	0.22	0.19
BACC	0.78	0.79
TSS	0.55	0.58
HSS	0.53	0.58
Brier_score	0.15	0.12
AUC	0.85	0.92
Acc_by_package_fn	0.78	0.81

ROC Plots

KNN

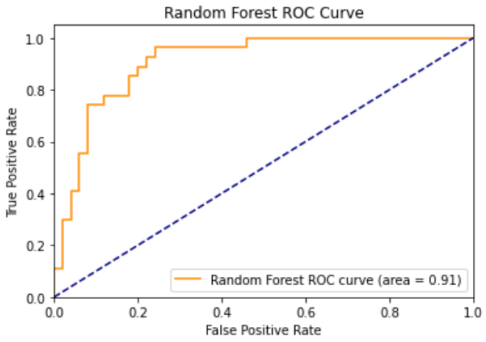
Evaluating the performance of various algorithms by comparing their ROC curves and AUC scores on the test dataset.

```
# KNN Model
knn_model = KNeighborsClassifier(n_neighbors=best_n_neighbors)
knn_model.fit(features_train_all_std, labels_train_all)

# Obtain predicted probabilities
y_score = knn_model.predict_proba(features_test_all_std)[:, 1]

# Compute ROC curve and ROC area
fpr, tpr, _ = roc_curve(labels_test_all, y_score)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc))
plt.plot([0, 1], [0, 1], color='navy', 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('KNN ROC Curve')
plt.legend(loc='lower right')
plt.show()
```



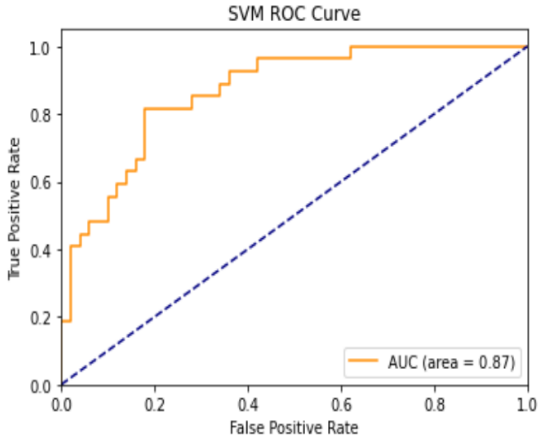
Random Forest

```
# Random Forest Model
rf_model = RandomForestClassifier(min_samples_split=min_samples_split, n_estimators=n_estimators)
rf_model.fit(features_train_all_std, labels_train_all)

# Obtain predicted probabilities
y_score_rf = rf_model.predict_proba(features_test_all_std)[:, 1]

# Compute ROC curve and ROC area
fpr_rf, tpr_rf, _ = roc_curve(labels_test_all, y_score_rf)
roc_auc_rf = auc(fpr_rf, tpr_rf)

# PLot Random Forest ROC curve
plt.figure()
plt.plot(fpr_rf, tpr_rf, color="darkorange", label="Random Forest ROC curve (area = {:.2f})".format(roc_auc_rf))
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("Random Forest ROC Curve")
plt.legend(loc="lower right")
plt.show()
```



SVM

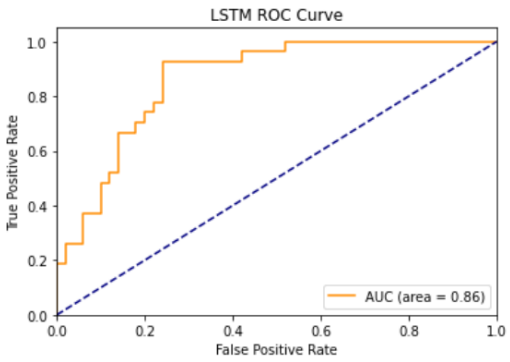
```
# SVM Classifier Model
svm_model = SVC(C=C, kernel='linear', probability=True)
svm_model.fit(features_train_all_std, labels_train_all)

# Obtain predicted probabilities
y_score_svm = svm_model.predict_proba(features_test_all_std)[:, 1]

# Compute ROC curve and ROC area
fpr_svm, tpr_svm, _ = roc_curve(labels_test_all, y_score_svm)
roc_auc_svm = auc(fpr_svm, tpr_svm)

# Plot SVM ROC curve
plt.figure()
plt.plot(fpr_svm, tpr_svm, color="darkorange", label="AUC (area = {:.2f})".format(roc_auc_svm))
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("SVM ROC Curve")
plt.legend(loc="lower right")
plt.show()
```



LSTM

```
# LSTM model
lstm_model = Sequential()
lstm_model.add(LSTM(64, activation='relu', batch_input_shape=(None, 8, 1), return_sequences=False))
lstm_model.add(Dense(1, activation='sigmoid'))
# Compile model
lstm_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Convert data to numpy array
X_train_array = features_train_all_std.to_numpy()
X_test_array = features_test_all_std.to_numpy()
y_train_array = labels_train_all.to_numpy()
y_test_array = labels_test_all.to_numpy()

# Reshape data
input_shape = X_train_array.shape
input_train = X_train_array.reshape(len(X_train_array), input_shape[1], 1)
input_test = X_test_array.reshape(len(X_test_array), input_shape[1], 1)
output_train = y_train_array
output_test = y_test_array

# Train the LSTM model
lstm_model.fit(input_train, output_train, epochs=50, validation_data=(input_test, output_test), verbose=0)

# Predict probabilities for the test set
predict_lstm = lstm_model.predict(input_test)

# Compute ROC curve and ROC area
fpr_lstm, tpr_lstm, _ = roc_curve(labels_test_all, predict_lstm)
roc_auc_lstm = auc(fpr_lstm, tpr_lstm)

# Plot LSTM ROC curve
plt.figure()
plt.plot(fpr_lstm, tpr_lstm, color="darkorange", label="AUC (area = {:.2f})".format(roc_auc_lstm))
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("LSTM ROC Curve")
plt.legend(loc="lower right")
plt.show()
```

```
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make_predict_function.<locals>.predict_function at 0x000001
1DF5E52EE0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) c
reating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of
tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental_relax_shapes=T
rue option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflo
w.org/tutorials/customization/performance#python_or_tensor_args and https://www.tensorflow.org/api_docs/python/tf/function f
or  more details.
```



Average performance metrics for all algorithms

```
# Calculate the average metrics for each algorithm
knn_avg_df = knn_metrics_df.mean()
rf_avg_df = rf_metrics_df.mean()
svm_avg_df = svm_metrics_df.mean()
lstm_avg_df = lstm_metrics_df.mean()

# Create a DataFrame with the average performance for each algorithm
avg_performance_df = pd.DataFrame({'KNN': knn_avg_df, 'RF': rf_avg_df, 'SVM': svm_avg_df, 'LSTM': lstm_avg_df}, index=metric_

# Display the average performance for each algorithm
print(avg_performance_df.round(decimals=2))
print('\n')
```

	KNN	RF	SVM	LSTM
TP	13.30	14.50	13.00	13.70
TN	38.50	37.90	39.30	37.10
FP	6.50	7.10	5.70	7.90
FN	10.80	9.60	11.10	10.40
TPR	0.55	0.60	0.54	0.57
TNR	0.86	0.84	0.87	0.82
FPR	0.14	0.16	0.13	0.18
FNR	0.45	0.40	0.46	0.43
Precision	0.67	0.67	0.70	0.64
F1_measure	0.60	0.63	0.60	0.59
Accuracy	0.75	0.76	0.76	0.74
Error_rate	0.25	0.24	0.24	0.26
BACC	0.70	0.72	0.71	0.70
TSS	0.41	0.44	0.41	0.39
HSS	0.42	0.45	0.43	0.40
Brier_score	0.16	0.16	0.16	0.17
AUC	0.82	0.82	0.84	0.81
Acc_by_package_fn	0.75	0.76	0.76	0.74

Conclusion

In this project, I have compared the performance of four different classification algorithms to predict whether a patient has diabetes, based on certain diagnostic measurements available in the dataset. Based on the results of all the algorithms we can conclude following observations:

- All four algorithms, KNN, Random Forest, SVM and LSTM, have performed similarly in the prediction of diabetes based on the dataset when we consider their average performance over 10 folds of the cross validation.
- The performances of algorithms differed in each individual iterations of the 10-fold cross validation but on average performed similar. This shows each algorithm performs differently to different datasets but on average have similar performance.
- The average accuracy of all 4 algorithms is similar to each other which is around 75%. This shows that all the algorithms have consistence performance in predicting the labels.
- When we look at the true positive rate and true negative rate, we can see that all algorithms have performed poorly in predicting positive outcomes with Random Forest with highest TPR of 61%. Whereas TNR for all algorithms is above 85%. This shows that the algorithms are performing better predicting negative outcome than positive outcome. This may be due to data label imbalance in our training set as the number of negative outcomes is almost twice the

number of positive outcomes. We should try and balance the outcome labels in the training set in future to avoid such issues.

- F1 measure of all algorithms is similar on average over 10 folds of cross validation with Random Forest having highest F1 measure of 64%
- BACC, which is the average of TPR and TNR, is between 70% and 73% for all algorithms. This is due to high TNR and low TPR values.
- The TSS and HSS scores for all algorithms are in the ranges of 40% to 46% which shows the algorithms have not performed well as per these metrics.
- The Brier scores for all algorithms is similar and low which shows all algorithms have performed decently in predicting probabilities of the outcomes
- The AUC of more than 80% for all four algorithms shows that the probabilities prediction of outcomes has been decent for all algorithms at all possible classification thresholds.
- Comparison of the accuracy calculated, and the accuracy derived using ready packages is used to validate that our performance metrics calculations are correct

Overall, the above performance of the algorithms can be attributed to following limitations that can be improved in the future.

- Our dataset has total of 768 records, which is low for training some of the advanced algorithms such as LSTM, Random Forest, and SVM. We need to increase the size of dataset to improve performance of the algorithms.
- The data label imbalance has affected the algorithms with all algorithms performing poorly to predict positive outcomes. We need to ensure training data has balanced data labels.