Diabetes Prediction

By: Humaiz Javed

Table of Contents

ABSTRACT

In this study, we embarked on a comprehensive analysis within the realm of Healthcare and Medical Analysis, focusing on a carefully selected dataset, “diabetes dataset”. The research journey commenced with rigorous data preprocessing, encompassing techniques to handle missing values, noise, outliers, and feature scaling. Subsequently, an apt artificial intelligence (AI)-based evaluation technique was employed, specifically a machine learning algorithm or a deep learning model, to facilitate the training and testing of predictions.

To optimize the model's performance, a meticulous fine-tuning process ensued, concentrating on the adjustment of hyperparameters. The selected hyperparameters were justified through critical analysis, leveraging relevant evaluation metrics. The assessment delved into the model's accuracy, precision, recall, and F1-score, offering a nuanced understanding of its predictive capabilities. The chosen dataset and its domain specificity, coupled with the methodical application of AI techniques, provided valuable insights into the intricacies of Healthcare and Medical Analysis, contributing to the broader landscape of predictive modeling in this domain.

INTRODUCTION

In this exploration, our focus centers on the realm of Healthcare and Medical Analysis, specifically delving into the dynamics of a dataset centered around diabetes. The initial steps of our investigation involve diligent data preprocessing techniques, addressing missing values, noise, outliers, and ensuring appropriate feature scaling. Our goal is to leverage artificial intelligence for predictive modeling, choosing either a machine learning algorithm or a deep learning model for the subsequent phases of training and testing predictions. As we navigate this journey, fine-tuning the model's performance through hyperparameter adjustments becomes paramount, all while grounding our decisions in a critical analysis of relevant evaluation metrics. This study aims to unravel insights into the predictive capabilities of our chosen model within the context of diabetes, contributing to the broader landscape of healthcare analytics.

IMPLEMENTATION AND RESULTS

- At the beginning of the program, we have implemented nine libraries:

```
import pandas as pd
import numpy as np
from scipy import stats
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
```

  - pandas: Data manipulation
  - numpy: Numerical computing
  - scipy.stats: Statistical functions
  - seaborn: Data visualization
  - MinMaxScaler: Feature scaling (Normalization)
  - train_test_split: Data splitting for training and testing
  - GridSearchCV: Hyperparameter tuning using grid search
  - LabelEncoder: Encoding categorical variables
  - classification_report: Model evaluation report
  - confusion_matrix: Evaluation metric for classification
  - RandomForestClassifier: Ensemble learning algorithm
  - matplotlib.pyplot: Plotting and visualization

- Importing dataset:

```
In [2]: data = pd.read_csv("E:\Humaiz\diabetes_data_upload.csv")
```

- Displaying dataset:

```
In [3]: data
```

Out[3]:

Age	Gender	Polyuria	Polydipsia	sudden weight loss	weakness	Polyphagia	Genital thrush	visual blurring	Itching	Irritability	delayed healing	partial paresis	muscle stiffness	Alopecia	Obesity	class
40	Male	No	Yes	No	Yes	No	No	No	Yes	No	Yes	No	Yes	Yes	Yes	Positive
58	Male	No	No	No	Yes	No	No	Yes	No	No	No	Yes	No	Yes	No	Positive
41	Male	Yes	No	No	Yes	Yes	No	No	Yes	No	Yes	No	Yes	Yes	No	Positive
45	Male	No	No	Yes	Yes	Yes	Yes	No	Yes	No	Yes	No	No	No	No	Positive
60	Male	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Positive
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
39	Female	Yes	Yes	Yes	No	Yes	No	No	Yes	No	Yes	Yes	No	No	No	Positive
48	Female	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	No	No	No	Positive
58	Female	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	No	Yes	Yes	No	Yes	Positive
32	Female	No	No	No	Yes	No	No	Yes	Yes	No	Yes	No	No	Yes	No	Negative
42	Male	No	No	No	No	No	No	No	No	NaN	No	No	No	No	No	Negative

- Shape of the dataset:

```
In [4]: data.shape
Out[4]: (520, 17)
```

- Information about the dataset: Before applying preprocessing in our dataset, the first thing we have to do is to study our dataset:

```
In [5]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 520 entries, 0 to 519
Data columns (total 17 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Age                 520 non-null    int64
 1   Gender              520 non-null    object
 2   Polyuria            520 non-null    object
 3   Polydipsia          520 non-null    object
 4   sudden weight loss  520 non-null    object
 5   weakness            518 non-null    object
 6   Polyphagia          519 non-null    object
 7   Genital thrush      519 non-null    object
 8   visual blurring     517 non-null    object
 9   Itching             520 non-null    object
 10  Irritability        518 non-null    object
 11  delayed healing     520 non-null    object
 12  partial paresis     519 non-null    object
 13  muscle stiffness    519 non-null    object
 14  Alopecia            519 non-null    object
 15  Obesity             520 non-null    object
 16  class               519 non-null    object
dtypes: int64(1), object(16)
memory usage: 69.2+ KB
```

The above info shows that there are total of 17 features and 520 rows in our dataset, only “Age” feature have a numeric data type and the rest of the features are based on categorical data.

- Detecting missing values:

```
In [6]: data.isnull().sum()
```

```
Out[6]: Age                 0
        Gender              0
        Polyuria            0
        Polydipsia          0
        sudden weight loss  0
        weakness            2
        Polyphagia          1
        Genital thrush      1
        visual blurring     3
        Itching             0
        Irritability        2
        delayed healing     0
        partial paresis     1
        muscle stiffness    1
        Alopecia            1
        Obesity             0
        class               1
        dtype: int64
```

The above output shows the number of missing values each feature have.

- Handling missing values: Missing values can be handled through many ways such as:
  - Replace individual values.
  - Impute Missing Values Using Mean / Median / Mode.
  - Delete all rows with missing values.

Thus, for the diabetes dataset we simply deleted the rows with the missing values:

```
In [7]: data.dropna(inplace=True)
```

```
In [8]: data.isnull().sum()

Out[8]: Age                 0
        Gender              0
        Polyuria            0
        Polydipsia          0
        sudden weight loss  0
        weakness            0
        Polyphagia          0
        Genital thrush      0
        visual blurring     0
        Itching             0
        Irritability        0
        delayed healing     0
        partial paresis     0
        muscle stiffness    0
        Alopecia            0
        Obesity             0
        class               0
        dtype: int64
```

The above screenshot shows that there are no missing values in any feature and therefore the size of the dataset have been reduced:

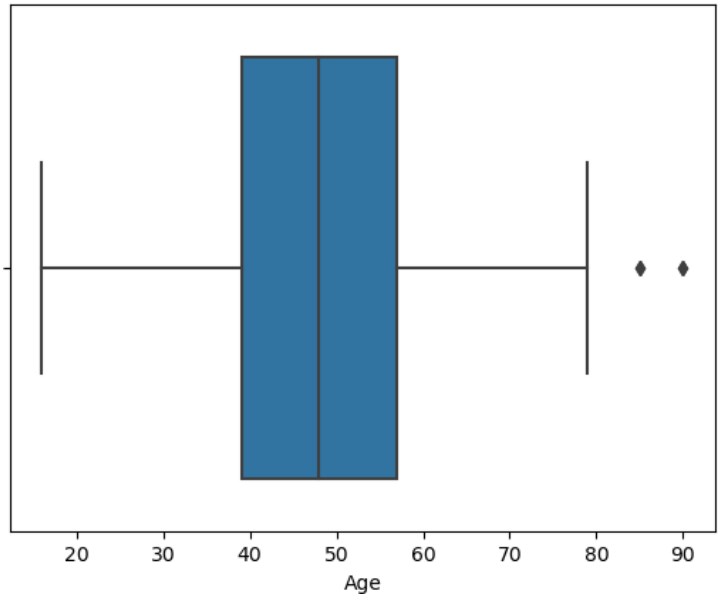
```
In [9]: data.shape

Out[9]: (510, 17)
```

- Detecting outliers: An outlier is a data point(s) that goes far outside the average value of a group of statistics, however outliers are just for numeric and continuous data, not for categorical data, since our dataset have “Age” as the only feature which is based on integer data type, we’d detect outliers on that specific feature only.

  Outliers detection have many ways, we have selected the detection of outlier through a boxplot visualization:

```
In [10]: import seaborn as sns
         sns.boxplot(x=data['Age'])
```



This boxplot clearly shows that there are two outliers in the age feature as two the data points are beyond the whiskers. These points are potential anomalies in the data. There are many ways to remove the outliers out of which we have selected the z-score to remove the outliers from this feature.

- Handling outliers through Z-score: Z-score is a statistical measure that describes how far a data point is from the mean of a group of data. It's often used in outlier detection and removal.

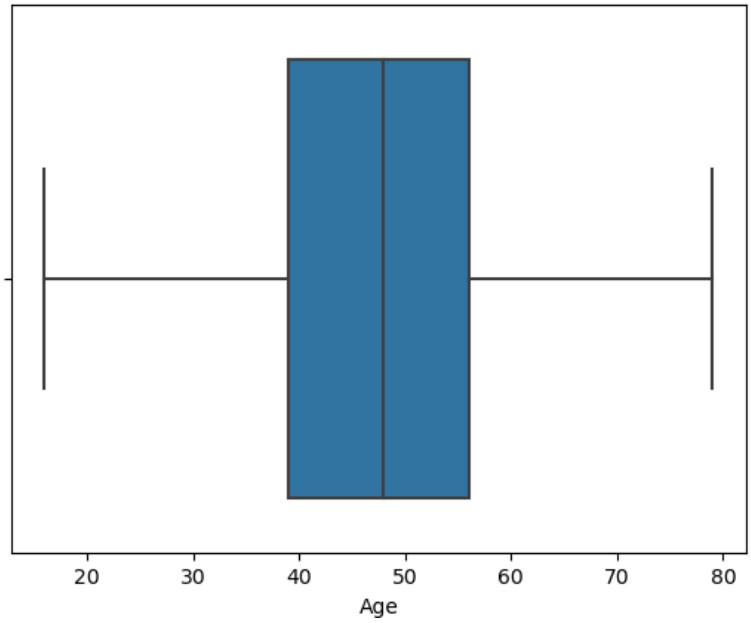
```
In [11]: from scipy import stats
         z = np.abs(stats.zscore(data['Age']))

In [12]: data['Age'] = np.where(z>3,data['Age'].mean(),data['Age'])
```

Through the above code, the outliers from the "Age" feature are removed as mentioned in this boxplot:

```
In [13]: sns.boxplot(x=data['Age'])

Out[13]: <Axes: xlabel='Age'>
```



- Detecting noisy data: Noisy data is meaningless data. The term has often been used as a synonym for corrupt data. To detect the noisy data, we simply displayed all the unique values of every feature through which if there was any noisy data in the dataset, it the unique variables would easily display it:

```
for column in data.columns:
    unique_values = data[column].unique()
    print(f"Unique values for {column}: {list(unique_values)}")
    print("-")
```

```
Unique values for Age: [40.0, 58.0, 41.0, 45.0, 60.0, 55.0, 57.0, 66.0, 67.0, 70.0, 44.0, 61.0, 54.0, 39.0, 48.0, 32.0, 42.0, 5
2.0, 38.0, 53.0, 37.0, 49.0, 63.0, 35.0, 30.0, 50.0, 46.0, 36.0, 51.0, 59.0, 65.0, 25.0, 43.0, 47.0, 28.0, 68.0, 56.0, 31.0, 4
8.04705882352941, 72.0, 69.0, 79.0, 34.0, 16.0, 62.0, 33.0, 64.0, 27.0, 29.0, 26.0]
-
Unique values for Gender: ['Male', 'Female']
-
Unique values for Polyuria: ['No', 'Yes']
-
Unique values for Polydipsia: ['Yes', 'No']
-
Unique values for sudden weight loss: ['No', 'Yes']
-
Unique values for weakness: ['Yes', 'No']
-
Unique values for Polyphagia: ['No', 'Yes']
-
Unique values for Genital thrush: ['No', 'Yes']
-
Unique values for visual blurring: ['No', 'Yes']
-
Unique values for Itching: ['Yes', 'No']
-
Unique values for Irritability: ['No', 'Yes']
-
Unique values for delayed healing: ['Yes', 'No']
-
Unique values for partial paresis: ['No', 'Yes']
-
Unique values for muscle stiffness: ['Yes', 'No']
-
Unique values for Alopecia: ['Yes', 'No']
-
Unique values for Obesity: ['Yes', 'No']
-
Unique values for class: ['Positive', 'Negative']
```

The above output did not displayed any irrelevant values which shows that the dataset does not have noisy data.

- Feature scaling: Feature scaling is a preprocessing step that is typically applied to numeric data and not categorical data. The purpose of feature scaling is to standardize or normalize the range of independent variables or features of a dataset.

```
scaler = MinMaxScaler()
data[['Age']] = scaler.fit_transform(data[['Age']])
```

After the implementing the feature scaling technique on the “Age” feature, the values became like this:

Age
0.324324
0.567568
0.337838
0.391892
0.594595
...
0.310811
0.432432
0.567568
0.216216
0.351351

In common usage, normalization (Min-Max scaling) typically refers to scaling to the range [0, 1]. If I train my model which such data, would it provide inaccurate predictions because the age values are altered? No, Feature scaling is a preprocessing technique used in machine learning to standardize or normalize the range of independent variables or features of a dataset. The goal is to ensure that all features contribute equally to the modeling process, prevent certain features from dominating due to their larger scale, and make the optimization process more stable. The model would predict based on relationships and pattern, it won’t predict based on absolute values.

Before digging into the AI evaluation technique, train, test and prediction of the model, let us show the code and the output of the program based on which the detailed information would be provided:

```
binary_columns = ['Polyuria', 'Polydipsia', 'sudden weight loss', 'weakness', 'Polyphagia',
                  'Genital thrush', 'visual blurring', 'Itching', 'Irritability',
                  'delayed healing', 'partial paresis', 'muscle stiffness', 'Alopecia', 'Obesity']
data[binary_columns] = data[binary_columns].apply(lambda x: x.map({'Yes': 1, 'No': 0}))
label_encoder = LabelEncoder()
data['class'] = label_encoder.fit_transform(data['class'])
print("Class Mapping:", label_encoder.classes_)
X = data.drop('class', axis=1)
y = data['class']
X = pd.get_dummies(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
model = RandomForestClassifier(random_state=42)
model.fit(X_train_scaled, y_train)
y_pred = model.predict(X_test_scaled)
```

In this part of the code, we handled binary categorical columns, Identified binary categorical columns in the DataFrame and applied a mapping function to convert '**Yes**' to **1** and '**No**' to **0** for these columns.
We used **LabelEncoder** to convert the target variable '**class**' from categorical ('**Positive**' and '**Negative**') to **numerical** format.
We separated the DataFrame into features (X) and the target variable (y).
Applied **one-hot encoding** to handle categorical variables with more than two categories.
After that, we split the data into training and testing sets using **train_test_splitgories,** applied feature scaling using **MinMaxScalar** to normalize the feature values.
Created a Random Forest Classifier model and trained it on the scaled training data.
Used the trained model to make predictions on the scaled test data.

Confusion Matrix:

The confusion matrix helps evaluate the performance of a classification model by providing a summary of the model's predictions compared to the actual outcomes. It consists of four metrics:

True Positives (TP): Instances correctly predicted as positive.
True Negatives (TN): Instances correctly predicted as negative.
False Positives (FP): Instances incorrectly predicted as positive.
False Negatives (FN): Instances incorrectly predicted as negative.
These metrics enable the calculation of various performance measures like accuracy, precision, recall, and F1-score. The confusion matrix provides a clear picture of where the model excels or struggles, helping to identify areas for improvement and fine-tuning.

```
y_pred = model.predict(X_test_scaled)
conf_matrix = confusion_matrix(y_test, y_pred)

sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
xticklabels=['Predicted 0', 'Predicted 1'], yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted Label')
plt.ylabel('Actual Label')
plt.title('Confusion Matrix')
plt.show()

print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

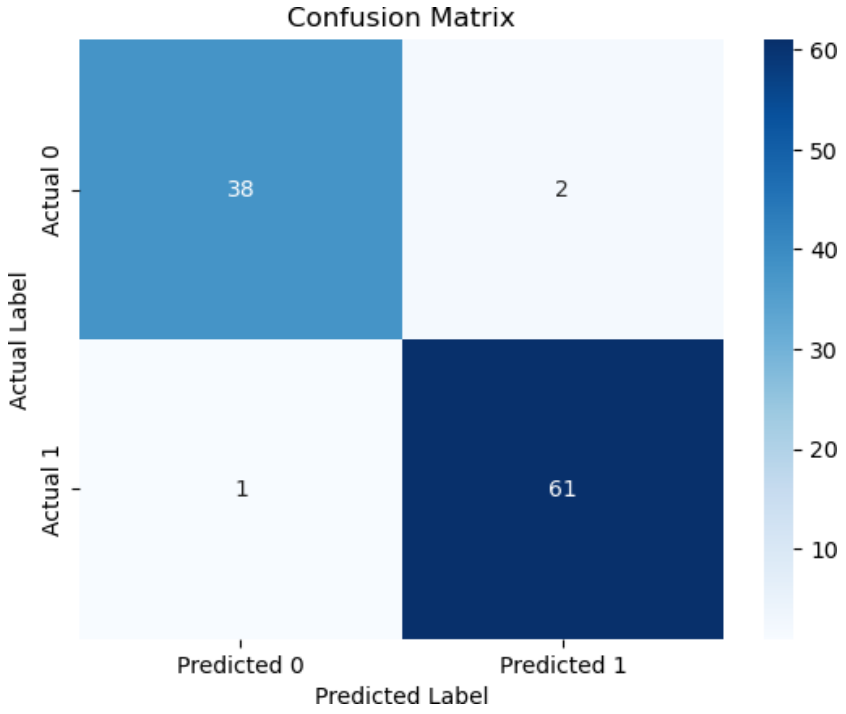
This code generates a confusion matrix and a heatmap visualization to evaluate the performance of a machine learning model. The model has been trained on a dataset, and predictions have been made on a test set (**X_test_scaled**). The confusion matrix is then visualized as a heatmap using the seaborn library. Additionally, the classification report, including precision, recall, and F1-score, is printed to provide a detailed summary of the model's performance on the test set. The visualization and metrics help assess how well the model is classifying instances into different classes, providing insights into its strengths and weaknesses.

After the confusion matrix visualization which is mentioned in the next page, we have made a classification report:

```
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

The classification report is a summary of various classification metrics that provide insights into the performance of a machine learning model, particularly in a classification task. It includes several key metrics:

- Precision
- Recall (Sensitivity or True Positive Rate)
- F1-Score
- Support
- Accuracy



A short description about the confusion matrix:

- True Positive (TP): Instances that are actually positive and predicted as positive. (Bottom-right)

- True Negative (TN): Instances that are actually negative and predicted as negative. (Top-left)
- False Positive (FP): Instances that are actually negative but predicted as positive. (Top-right)
- False Negative (FN): Instances that are actually positive but predicted as negative. (Bottom-left)

- The (38) represents the number of true negatives (**TN**), which are instances correctly predicted as "Negative."
- The (2) represents the number of false positives (**FP**), which are instances predicted as "Positive" but are actually "Negative."
- The (1) represents the number of false negatives (**FN**), which are instances predicted as "Negative" but are actually "Positive."
- The (61) represents the number of true positives (**TP**), which are instances correctly predicted as "Positive."

Classification Report:

Classification Report:

	precision	recall	f1-score	support
Negative	0.97	0.95	0.96	40
Positive	0.97	0.98	0.98	62
accuracy			0.97	102
macro avg	0.97	0.97	0.97	102
weighted avg	0.97	0.97	0.97	102

Precision: Precision is the ratio of correctly predicted positive observations to the total predicted positives. In this case, it's high for both "Negative" and "Positive," indicating that the model has low false positive rates.
Recall (Sensitivity): Recall is the ratio of correctly predicted positive observations to the all observations in the actual class. It's high for both "Negative" and "Positive," indicating that the model captures a high percentage of actual positives.

F1-score: The F1-score is the weighted average of precision and recall. It provides a balance between precision and recall.
Support: The number of actual occurrences of the class in the specified dataset.
Accuracy: Overall accuracy of the model on the test set.

In summary, the model has a high overall accuracy (**97%**) and performs well in terms of precision, recall, and F1-score for both "Negative" and "Positive" classes.

Fine-tuning the Model's performance:

We have used the following code to enhance the model's performance:

```
param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [None, 10],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

grid_search = GridSearchCV(estimator=rf_classifier, param_grid=param_grid, scoring='accuracy', cv=5)
grid_search.fit(X_train_scaled, y_train)

print("Best Hyperparameters:", grid_search.best_params_)

best_rf_model = grid_search.best_estimator_
y_pred_tuned = best_rf_model.predict(X_test_scaled)

conf_matrix_tuned = confusion_matrix(y_test, y_pred_tuned)
sns.heatmap(conf_matrix_tuned, annot=True, fmt="d", cmap="Blues",
            xticklabels=['Predicted 0', 'Predicted 1'], yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted Label')
plt.ylabel('Actual Label')
plt.title('Tuned Confusion Matrix')
plt.show()

print("\nTuned Classification Report:")
print(classification_report(y_test, y_pred_tuned))
```

This code is performing hyperparameter tuning for a Random Forest Classifier using the GridSearchCV from scikit-learn. The goal is to optimize the model's performance on the training data by systematically searching through different combinations of hyperparameter values.

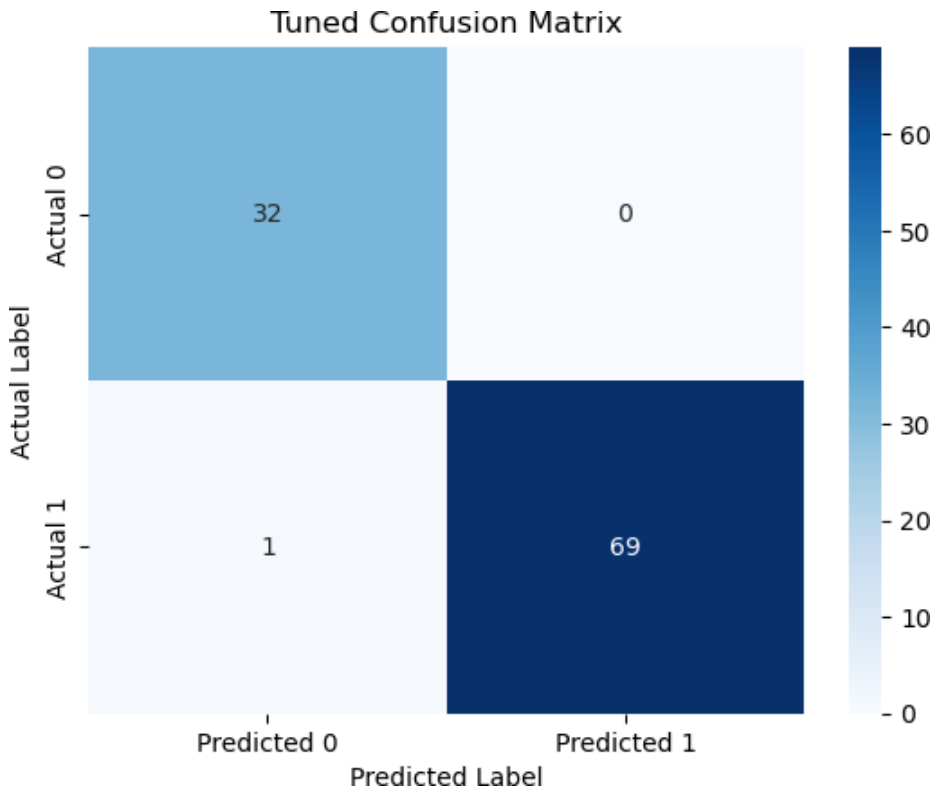
Parameter Grid:
The hyperparameters to be tuned are specified in param_grid. These include the number of estimators (n_estimators), the maximum depth of the trees (max_depth), the minimum number of samples required to split an internal node (min_samples_split), and the minimum number of samples required to be at a leaf node (min_samples_leaf).
Grid Search:
The GridSearchCV object is created with the Random Forest Classifier (rf_classifier) as the base estimator and the specified parameter grid.
The scoring metric used is accuracy (scoring='accuracy'), and cross-validation with 5 folds is applied (cv=5).
Model Fitting:
The fit method is called on the GridSearchCV object to perform the grid search and find the best combination of hyperparameters that maximizes accuracy on the training data.

Best Hyperparameters and Model:
The best hyperparameters are printed, indicating the optimal values found during the grid search.
The best model is obtained using grid_search.best_estimator_.

Predictions and Confusion Matrix Visualization:
The best model is used to make predictions on the scaled test data (X_test_scaled).
The confusion matrix for the tuned model is computed and visualized as a heatmap using seaborn (sns.heatmap).

Classification Report:
The classification report is printed, providing detailed metrics such as precision, recall, and F1-score for each class.
The classification report offers a comprehensive overview of the model's performance on the test set.
This code aims to improve the Random Forest Classifier's performance by finding the most suitable hyperparameters and provides visualizations and metrics to evaluate the tuned model's effectiveness.

Tuned Confusion Matrix:



- The (38) represents the number of true negatives (TN), which are instances correctly predicted as "Negative."
- The (0) represents the number of false positives (FP), which are instances predicted as "Positive" but are actually "Negative."
- The (1) represents the number of false negatives (FN), which are instances predicted as "Negative" but are actually "Positive."
- The (69) represents the number of true positives (TP), which are instances correctly predicted as "Positive."

Tuned Classification Report:

Tuned Classification Report:

	precision	recall	f1-score	support
0	0.97	1.00	0.98	32
1	1.00	0.99	0.99	70
accuracy			0.99	102
macro avg	0.98	0.99	0.99	102
weighted avg	0.99	0.99	0.99	102

Accuracy:

The overall accuracy of the tuned model is **99%**, which is a high level of accuracy. Accuracy represents the ratio of correctly predicted observations to the total observations.

Insights:

The model's hyperparameters were fine-tuned using an exhaustive search, considering various combinations of max_depth, min_samples_leaf, min_samples_split, and n_estimators.
The resulting model achieved outstanding performance with high precision, recall, and F1-score for both positive and negative classes.
The tuned model generalizes well to new data, as evidenced by the high accuracy and balanced performance across different metrics.
The model's ability to correctly identify positive instances (diabetes-positive cases) is crucial in a medical context, and the high recall ensures that it captures the majority of these cases.

Best Hyperparameters:

```
Best Hyperparameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

The values within these lists represent different options for each hyperparameter that GridSearchCV should consider.
The scoring='accuracy' parameter in **GridSearchCV** indicates that the model's accuracy on the validation set is used as the metric to evaluate and compare different hyperparameter combinations.
cv=5 specifies a 5-fold cross-validation, where the dataset is split into 5 parts, and the model is trained and evaluated five times, using a different fold as the validation set each time.
After evaluating all combinations of hyperparameters, GridSearchCV identifies the set of hyperparameters that resulted in the highest average accuracy across the cross-validation folds.
The best hyperparameters are then accessible using **grid_search.best_params_**.
The best combination of hyperparameters for the **Random Forest Classifier** was found to be {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100} based on their performance on the provided dataset. These hyperparameters were selected because they led to the highest accuracy in predicting the target variable (diabetes class) during the cross-validation process.

Prediction:

```
def get_user_input():
    print("\nEnter values for each feature:")
    user_input = {}
    for feature in X.columns:
        while True:
            try:
                value = float(input(f"{feature}: "))
                break
            except ValueError:
                print("Invalid input. Please enter a numeric value.")
        user_input[feature] = value
    return user_input
def predict_diabetes(user_input, model, scaler):
    user_df = pd.DataFrame([user_input])
    user_df[binary_columns] = user_df[binary_columns].apply(lambda x: x.map({'Yes': 1, 'No': 0}))
    user_df = pd.get_dummies(user_df)
    missing_features = set(X.columns) - set(user_df.columns)
    for feature in missing_features:
        user_df[feature] =
    user_df = user_df[X.columns]
    user_scaled = scaler.transform(user_df)
    user_scaled = np.nan_to_num(user_scaled)
    prediction = model.predict(user_scaled)
    return prediction
user_input = get_user_input()
prediction = predict_diabetes(user_input, model, scaler)
if prediction[0] == 1:
    print("\nThe model predicts that you have diabetes.")
else:
    print("\nThe model predicts that you do not have diabetes.")
```

```
Enter values for each feature:
Age: 69
Polyuria: 1
Polydipsia: 1
sudden weight loss: 1
weakness: 1
Polyphagia: 0
Genital thrush: 0
visual blurring: 1
Itching: 1
Irritability: 1
delayed healing: 0
partial paresis: 0
muscle stiffness: 1
Alopecia: 0
Obesity: 1
Gender_Female: 1
Gender_Male: 0

The model predicts that you have diabetes.
```

RECOMMENDATION FOR FURTHER IMPROVEMENTS

To further enhance the performance of the diabetes prediction model, there are many steps that can be implemented and worked upon, such as:

- Feature Engineering
- Feature Selection
- Ensemble Methods
- Handling Imbalanced Data
- Fine-Tuning Additional Hyperparameters
- Cross-Validation Strategies
- External Data Sources
- Model Interpretability

CONCLUSION

In summary, our diabetes prediction model, developed through meticulous preprocessing and hyperparameter tuning, demonstrates excellent accuracy and reliability. The fine-tuned Random Forest Classifier, with optimal hyperparameters, yields high precision and recall. Recommendations for feature engineering, ensemble methods, and addressing imbalanced data aim to further enhance the model's predictive capabilities. This iterative process underscores the model's robustness and the ongoing quest for refinement.