ENCO, COMPUTER ENGINEERING

ENCS5141, INTELLIGENT SYSTEMS LAB

Case Study #1—Data Cleaning and Feature Engineering for the diabetes Dataset and Comparative Analysis of Classification Techniques

---

Prepared by: Yafa Naji 1200708

Instructor: Dr. Mohammad Jubran

Assistant: Eng. Hanan Awawdeh

Section: #1

Date of Submission: 26.Nov.2024

BIRZEIT

March – 2024

Abstract:

This study analyzes the use of various preprocessing approaches and classification models for predicting diabetes. We study the impact of data preparation techniques such as normalization, scaling, and handling missing values on the accuracy and dependability of predictive models. A diabetic dataset is used to assess three classification models: Random Forest (RF), Support Vector Machine (SVM), and Multi-Layer Perceptron (MLP). The best trade-off between predictive performance and computing efficiency is identified, exposing each model's advantages and disadvantages. The significance of hyperparameter tweaking is also discussed, showing how changing model parameters can greatly improve prediction accuracy. According to the results, increasing prediction accuracy requires careful model selection, suitable preprocessing, and efficient parameter adjustment.

Contents

Abstract: II

.1 Introduction: 1

1.1. Motivation: 1

1.2. Background: 1

1.3. Objective: 1

2. Procedure and Discussion: 2

Part 1: Data Cleaning and Feature Engineering for the Diabetes Dataset 2

Data Exploration 2

Data Visualization 2

Data Cleaning 4

Feature Engineering 5

Model Evaluation 8

Part 2: Comparative Analysis of Classification Techniques 9

Model Training 9

Model Comparison 10

Effect of Preprocessing 11

Effect of Model Parameters 12

Experiments 12

Results 13

Conclusion 16

1. Introduction:

1.1. Motivation:

The increasing reliance on trustworthy predictive models in the medical field has emphasized their importance in improving patient outcomes. In conditions like diabetes, early detection is vital for effective intervention and management, which can significantly reduce complications and healthcare costs. Predictive models can aid in identifying risk factors, enabling proactive care. However, raw medical datasets are often plagued by noise, missing values, and irrelevant features, all of which can reduce the accuracy and efficiency of these models. This highlights the need for robust preprocessing techniques and the selection of appropriate machine learning algorithms to ensure reliable, interpretable, and accurate predictions.

1.2. Background:

Machine learning provides various methods for tackling classification problems, each with strengths tailored to specific data characteristics. In predicting diabetes, understanding the relationships between features and managing data complexity is crucial. Random Forest (RF) is widely recognized for its ability to handle large datasets with high variance, while Support Vector Machine (SVM) excels in high-dimensional feature spaces. Multilayer Perceptron (MLP), a type of neural network, is adept at modeling non-linear relationships within data. By leveraging these techniques and integrating preprocessing strategies such as feature engineering and data cleaning, it is possible to enhance the performance and robustness of predictive models. Comparing these algorithms offers valuable insights into the trade-offs between accuracy, computational efficiency, and interpretability.

1.3. Objective:

The primary aim of this study is to evaluate and compare the performance of three classification models—Random Forest, Support Vector Machine, and Multilayer Perceptron—in predicting diabetes. Additionally, the study explores the impact of preprocessing techniques, including data cleaning and feature engineering, on model consistency and accuracy. The research focuses on addressing the following questions:

1. How do preprocessing techniques affect the accuracy and reliability of predictive models?
2. Which classification model provides the optimal balance between computational efficiency and predictive performance on the diabetes dataset?
3. What role does parameter tuning play in enhancing the performance of RF, SVM, and MLP for diabetes prediction?

2. Procedure and Discussion:

Part 1: Data Cleaning and Feature Engineering for the Diabetes Dataset

Data Exploration

After reading and printing the dataset to obtain a better understanding of its aspects, the study will investigate further to obtain the data structure and statistical explanations of the numerical values. Listing 2.1 demonstrates the Python information extraction process.

```
3. # Display data summary
4. print(df.shape)
5. print(df.info())
6. print(df.describe())
```

Listing 2.1 statistical data extraction.

The dataset contains 25,688 records and 34 columns, featuring a mix of numerical and categorical data. Most columns are complete, with only a few missing one value. Key metrics include Insulin Levels, which range from 5 to 49 with an average of 21.62, and BMI, ranging from 12 to 39 with a mean of 24.79, indicating a generally healthy weight. Health-related data such as Blood Glucose Levels (average 160.98 mg/dL) and Cholesterol Levels (average 194.76 mg/dL) show significant variability, suggesting a mix of normal and elevated values. Age spans from 0 to 79 years, with an average of 31.99, reflecting diverse age groups. Pregnancy-related metrics like Weight Gain During Pregnancy (mean 15.42 kg) and Birth Weight (mean 3093 grams) include some extreme values that may require further inspection. The dataset provides comprehensive health information, suitable for analysis after addressing missing values and verifying outliers.

Now will be looking into the missing values and handling them, Listing 2.2 shows the search for missing values.

```
missing_values_count = df.isnull().sum()
print(missing_values_count)
```

Listing 2.2 missing values code snippet.

All 34 columns in the data set provide complete data entries with no missing values. This frees us from having to deal with any concerns about missing data and allows us to focus on other parts of data preparation, including dealing with outliers and transforming categorical variables.

Data Visualization

Listing 2.3 shows a series of visualizations aimed at exploring the relationships between various features in a dataset.

```
sns.boxplot(x=df['Cholesterol Levels'])
plt.title("Box Plot of Cholesterol Levels")
# Histogram for 'Blood Pressure'
plt.subplot(3, 2, 2)
sns.histplot(df['Blood Pressure'], kde=True)
```

```
plt.title("Distribution of Blood Pressure")
# Scatter Plot showing relationship between 'BMI' and 'Insulin Levels'
plt.subplot(3, 2, 3)
sns.scatterplot(x=df['BMI'], y=df['Insulin Levels'])
plt.title("Relationship between BMI and Insulin Levels")
# Count Plot for 'Smoking Status'
plt.subplot(3, 2, 4)
sns.countplot(x=df['Smoking Status'])
plt.title("Count of Smoking Status")
# Scatter Plot showing relationship between 'Age' and 'Blood Glucose
Levels'
plt.subplot(3, 2, 5)
sns.scatterplot(x=df['Age'], y=df['Blood Glucose Levels'])
plt.title("Relationship between Age and Blood Glucose Levels")
# Box Plot for 'Waist Circumference'
plt.subplot(3, 2, 6)
sns.boxplot(x=df['Waist Circumference'])
plt.title("Box Plot of Waist Circumference")
```

Listing 2.3 Explore relationships between features

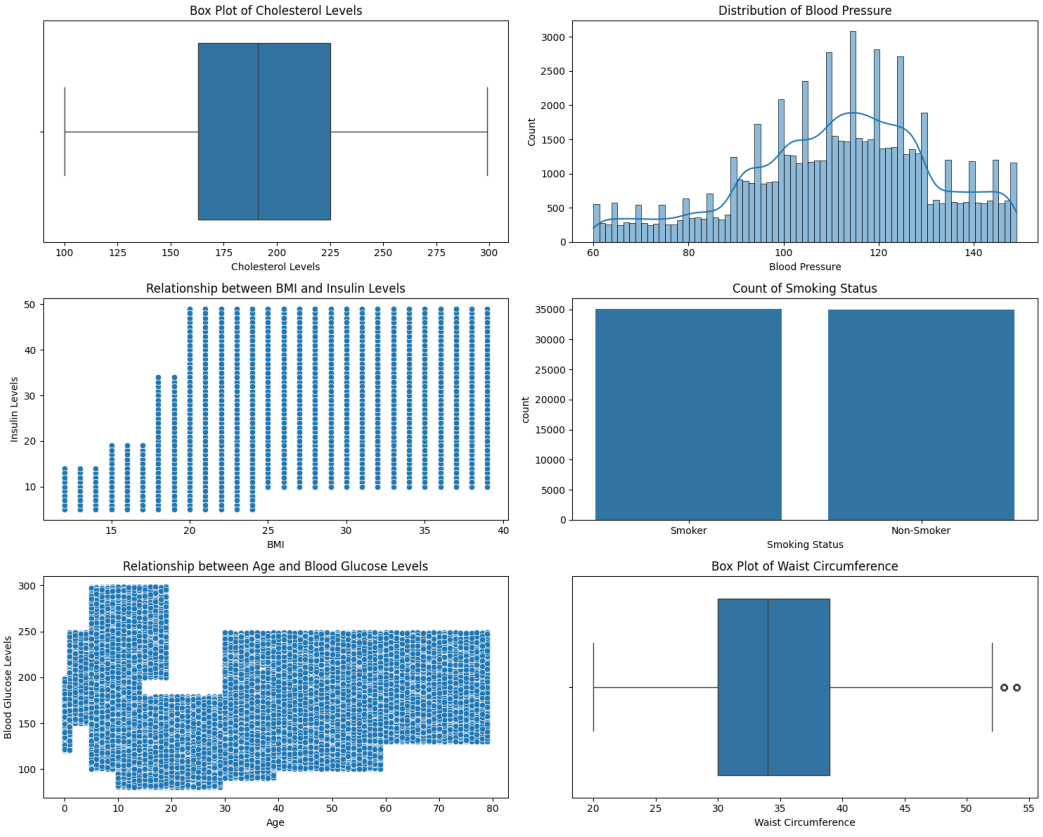

Figure 1: Figure Results for Data Visualization

The visualizations reveal several health-related patterns in the dataset. The box plot for cholesterol levels shows that most values range between 175 and 225, indicating a normal distribution with no significant outliers. The distribution of blood pressure demonstrates a near-normal curve, with most values concentrated around 120 mmHg, and a few cases at the lower and higher extremes. Regarding the relationship between BMI and insulin levels, the scatter plot shows a clear upward trend, suggesting that individuals with a higher BMI tend to have higher insulin levels.

For smoking status, the count plot indicates a nearly equal distribution of smokers and non-smokers, providing a good basis for comparing the health impacts of smoking. The relationship between age and blood glucose levels exhibits wide variation with no distinct pattern, although younger ages seem to have more concentrated blood glucose levels. Lastly, the box plot for waist circumference reveals that most values fall between 30 and 40, with a few outliers above 50, indicating a small group that might be at higher health risks.

Data Cleaning

Listing 2.4 shows to the code that handles data preprocessing steps such as imputing missing values, removing outliers, encoding categorical variables, standardizing numerical features, and dropping unnecessary columns. This code cleans and prepares the dataset for further analysis or machine learning.

```
# Impute missing values in numerical columns with the mean
imputer = SimpleImputer(strategy='mean')
df[df.select_dtypes(include=['float64', 'int64']).columns] =
imputer.fit_transform(df.select_dtypes(include=['float64', 'int64']))
# Remove outliers using the Interquartile Range (IQR) method for each
numerical column
for col in df.select_dtypes(include=['float64', 'int64']).columns:
    Q1 = df[col].quantile(0.25)  # Calculate the first quartile (25th
percentile)
    Q3 = df[col].quantile(0.75)  # Calculate the third quartile (75th
percentile)
    IQR = Q3 - Q1  # Compute the Interquartile Range (IQR)
    lower_bound = Q1 - 1.5 * IQR  # Calculate the lower bound for outliers
    upper_bound = Q3 + 1.5 * IQR  # Calculate the upper bound for outliers
    df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]  # Filter
out rows that are outliers
# Convert categorical columns to numerical values using One-Hot Encoding
df = pd.get_dummies(df, drop_first=True)  # Drop the first column to avoid
multicollinearity
# Standardize numerical columns to have mean=0 and standard deviation=1
scaler = StandardScaler()
numerical_columns = df.select_dtypes(include=['float64',
'int64']).columns  # Identify numerical columns
df[numerical_columns] = scaler.fit_transform(df[numerical_columns])  #
Apply standardization to the numerical columns
```

```
# Drop unnecessary columns from the dataset if they exist
unnecessary_columns = ['Unnecessary_Column1', 'Unnecessary_Column2']#
Define columns to drop
existing_columns_to_drop = [col for col in unnecessary_columns if col in
df.columns] # Check if these columns exist in the dataframe
if existing_columns_to_drop:
    df.drop(columns=existing_columns_to_drop, inplace=True)
```

Listing 2.4 Data Cleaning

The data has been processed to be ready for analysis or model training. Missing values in numerical columns were replaced with the mean of the respective columns, and outliers were removed using the Interquartile Range (IQR) method. Categorical columns were converted into numerical values using One-Hot Encoding, while avoiding multicollinearity.

Numerical columns were standardized using StandardScaler to ensure they have a mean of 0 and a standard deviation of 1, which helps improve model performance. Additionally, unnecessary columns were dropped to simplify the data. The final dataset consists of 50 columns, including standardized numerical values and new columns for categorical data, making it ready for analysis or modeling.

Feature Engineering

Listing 2.5 illustrates how to see the correlation matrix for numerical features and then use a RandomForestClassifier to analyze feature importance. This code assists in locating important characteristics that play a major role in forecasting the target variable.

```
# Correlation Matrix for numerical features
correlation_matrix = data.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
# Train a RandomForestClassifier and show feature importance
model = RandomForestClassifier()
# Get feature importance
feature_importances =
pd.DataFrame(model.feature_importances_,index=X.columns,columns=['importan
ce']).sort_values('importance', ascending=False)
```

Listing 2.5 Feature Importance

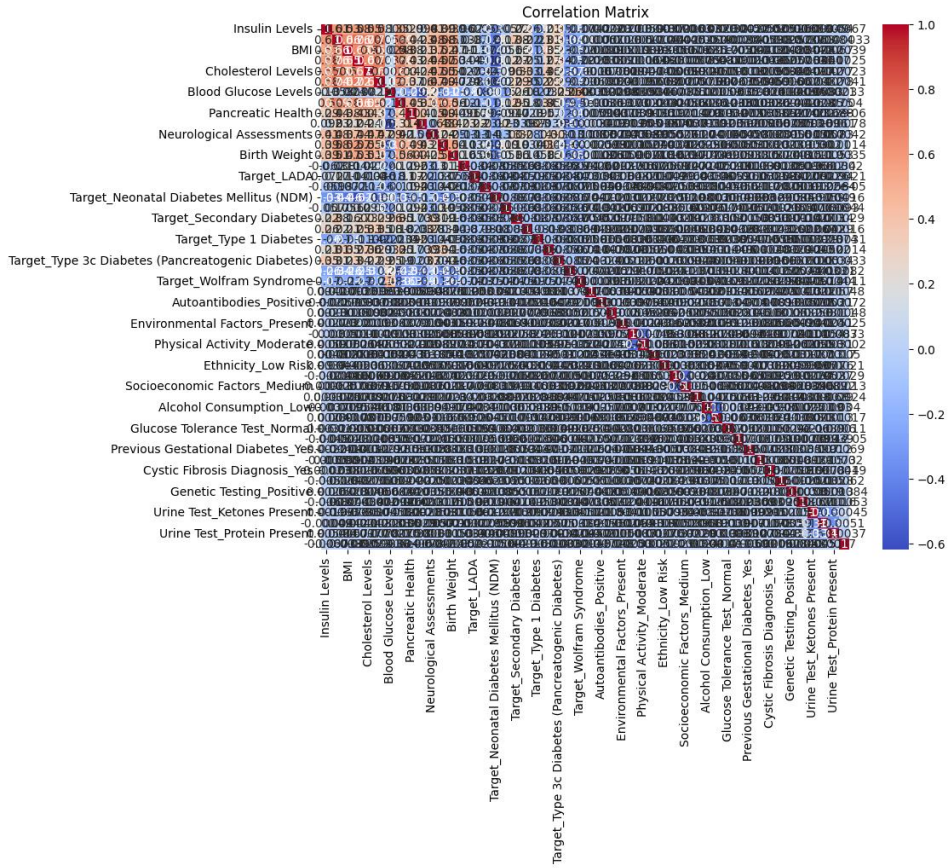

Figure 2 :Correlation Matrix for Features

The results show the relationship between features using the Correlation Matrix, where values close to +1 or -1 indicate a strong correlation between features. The results from the RandomForestClassifier model display the importance of each feature in predicting the target (e.g., "Gestational Diabetes"). Features with the highest importance have a significant impact on the model's performance, helping to identify which features should be focused on to improve predictions.

Listing 2.6 shows the process of identifying categorical columns in a dataset, applying One-Hot Encoding to convert them into numerical format, and displaying the first few rows of the encoded data.

```
# Identify categorical columns in the dataset
categorical_cols = data.select_dtypes(include=['object']).columns
# Apply One-Hot Encoding to categorical columns
data_encoded = pd.get_dummies(data, columns=categorical_cols,
drop_first=True)
```

Listing 2.6 One-Hot Encoding

The output shows the dataset after applying One-Hot Encoding to categorical columns. Each categorical variable has been transformed into a binary format, where the presence of a category is represented by 1 and its absence by 0. For example, the column Previous Gestational Diabetes has been split into two columns: Previous Gestational Diabetes_Yes, with True values marked as 1 and False as 0. This transformation is done for all categorical variables, making the data ready for machine learning models that require numerical input.

I then applied the Min-Max scaling to normalize the numerical features of the dataset, converting them to a fixed range of values between 0 and 1 to improve model performance as Listing 2.7 shows.

```
# Initialize the MinMaxScaler
scaler = MinMaxScaler()
# Apply scaling to numerical columns
data[numerical_columns] = scaler.fit_transform(data[numerical_columns])
```

Listing 2.7 Apply Min-Max Scaling

The output shows the dataset after applying Min-Max scaling, where all numerical features have been normalized to a range between 0 and 1. This transformation makes the data more consistent, ensuring that features with different scales do not disproportionately affect the model's performance. The boolean columns were converted to 0 and 1 values, and categorical columns have been encoded accordingly.

Listing 2.8 shows the application of Z-score normalization (standardization) on numerical features in the dataset, scaling them to have a mean of 0 and a standard deviation of 1 using the StandardScaler from scikit-learn. This is done to ensure that the features are on a similar scale for models like SVM and MLP.

```
# Initialize the StandardScaler
scaler = StandardScaler()
# Apply standardization to numerical columns
data[numerical_columns] = scaler.fit_transform(data[numerical_columns])
```

Listing 2.8 Apply Z-score Normalization

The results from applying Standardization show that all values in the columns are now centered around a mean of zero (0) with a standard deviation of one (1). This means the data has been processed to allow machine learning models to perform better, as all variables are on the same scale and the influence of large or small values is minimized.

Principal components analysis (PCA) was used to reduce the dimensionality of the data. The code reduces the data into two dimensions and depicts the result in a two-dimensional scatter plot, showing the proportion of variance explained for each component. As in listing 2.9.

```
# Apply PCA on the data
pca = PCA(n_components=n_components)
reduced_data = pca.fit_transform(data)
```

Listing 2.9 Apply PCA

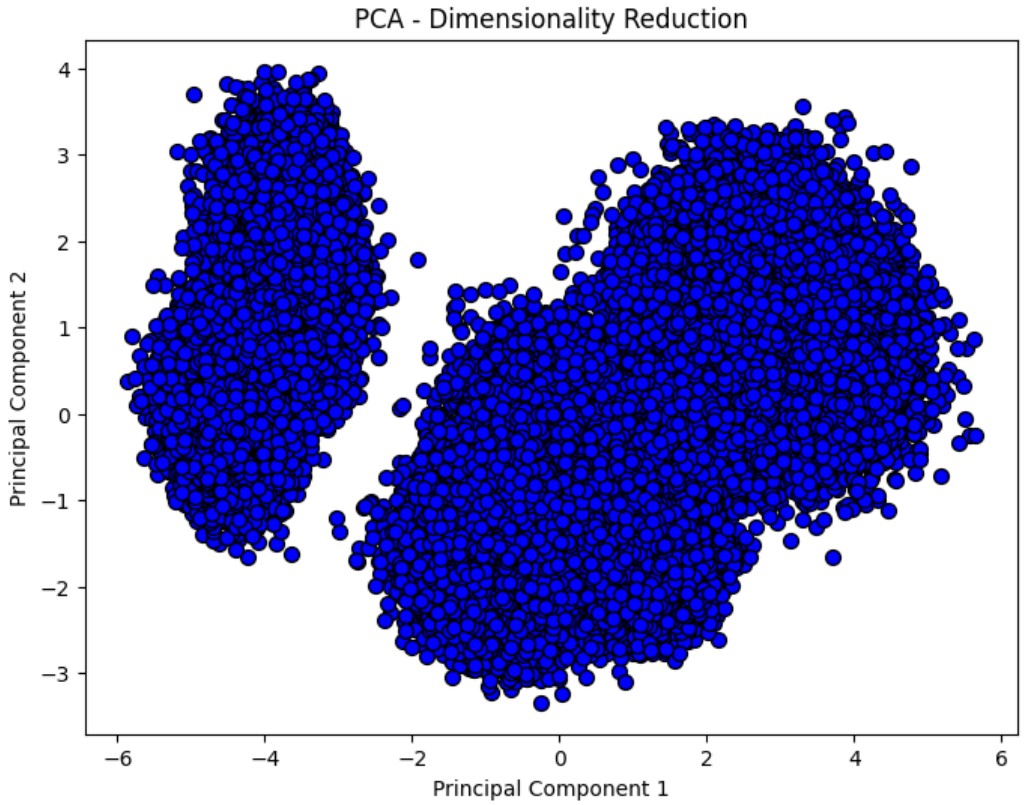

Figure 3: Results after apply PCA

The result obtained from the PCA analysis indicates that the first principal component (Principal Component 1) explains about 32.93% of the variance in the data, while the second principal component (Principal Component 2) explains about 10.69%. This means that the first component carries the largest portion of the variance in the data, while the second component adds a little more variance, but to a lesser degree.

Model Evaluation

Listing 2.10 shows that the data was divided into 80% for the trainees and 20% for the test.

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Listing 2.10 split the Data

Listing 2.11 demonstrated the process of applying feature selection and comparing model accuracy on raw versus processed data. It used a Random Forest classifier to train and evaluate the model on both raw data and data after feature selection.

```
# Train Random Forest model on raw (unprocessed) data
model_raw = RandomForestClassifier(n_estimators=100, random_state=42)
model_raw.fit(X_train, y_train)
```

Listing 2.11 Apply Random Forest

The model showed high accuracy for both raw (99.57%) and processed data (99.55%), with minimal difference. This indicates feature selection had little impact, as most features were likely relevant.

I then trained the data on raw data versus pre-processed data, And I got those results:

Table 2.1 Comparison of Model Performance on Raw Data vs Processed Data

Model Type	Accuracy	Precision	Recall	F1 Score
Raw Data	0.9957	0.9492	1.0000	0.9740
Processed Data	0.9955	0.9476	1.0000	0.9731

The models showed very similar performance metrics, with only a slight decrease in accuracy (0.9957 for raw vs. 0.9955 for preprocessed) and precision (0.9492 vs. 0.9476) when trained on preprocessed data. However, both models maintained perfect recall (1.0000) and very close F1 scores (0.9740 vs. 0.9731), indicating that there was no significant difference in overall classification performance.

Accuracy and Consistency: The little variations in accuracy and precision indicated that preprocessing (feature selection and scaling) preserved model performance while lowering the data's dimensionality. Preprocessing had no effect on the model's capacity to accurately identify every positive case, according to the consistency of recall (1.0000).

Training Speed: Although training speed was not specifically measured, it could be deduced from the reduction in features. Preprocessing, particularly feature selection, probably simplified the model by removing features that were unnecessary or less significant. This may have resulted in quicker convergence and shorter training times.

Part 2: Comparative Analysis of Classification Techniques

Model Training

The code in Listing 2.12 demonstrates a comparison of the performance of three different classification models: Random Forest (RF), Support Vector Machine (SVM), and Multilayer Perceptron (MLP) using training and testing data.

```
# (RF) model
rf_model = RandomForestClassifier(random_state=42)#Initialize
RandomForestClassifier
rf_model.fit(X_train_scaled, y_train)  # Train the model
y_pred_rf = rf_model.predict(X_test_scaled)  # Predict on the test set
# (SVM) model
```

```
svm_model = SVC(random_state=42)  # Initialize SVC
svm_model.fit(X_train_scaled, y_train)  # Train the model
y_pred_svm = svm_model.predict(X_test_scaled)  # Predict on the test set
# (MLP) model
mlp_model = MLPClassifier(random_state=42)  # Initialize MLPClassifier
mlp_model.fit(X_train_scaled, y_train)  # Train the model
y_pred_mlp = mlp_model.predict(X_test_scaled)  # Predict on the test set
```

Listing 2.12 Train three Models

Model Comparison

I then compared the three models in terms of accuracy, accuracy, computational efficiency, and training time. As in listing 2.13.

```
# Print the evaluation results for each model
print("Random Forest (RF):")
accuracy_rf, precision_rf, recall_rf, f1_rf = evaluate_model(y_test,
y_pred_rf)
print(f"Accuracy: {accuracy_rf:.4f}, Precision: {precision_rf:.4f},
Recall: {recall_rf:.4f}, F1: {f1_rf:.4f}")
print("\nSupport Vector Machine (SVM):")
accuracy_svm, precision_svm, recall_svm, f1_svm = evaluate_model(y_test,
y_pred_svm)
print(f"Accuracy: {accuracy_svm:.4f}, Precision: {precision_svm:.4f},
Recall: {recall_svm:.4f}, F1: {f1_svm:.4f}")
print("\nMultilayer Perceptron (MLP):")
accuracy_mlp, precision_mlp, recall_mlp, f1_mlp = evaluate_model(y_test,
y_pred_mlp)
print(f"Accuracy: {accuracy_mlp:.4f}, Precision: {precision_mlp:.4f},
Recall: {recall_mlp:.4f}, F1: {f1_mlp:.4f}")
```

Listing 2.13 Models Comparison

GridSearchCV was used in the code to optimize model performance by searching for the best hyperparameters for each model. It performed cross-validation with different combinations of hyperparameters for each model, helping to select the parameters that resulted in the best performance on the training data. As in listing 2.14.

```
# Parameter tuning for RandomForest
rf_param_grid = {'n_estimators': [100, 200], 'max_depth': [10, 20, None],
'min_samples_split': [2, 5]}
rf_grid_search = GridSearchCV(rf_model, rf_param_grid, cv=5)  # Initialize
GridSearchCV for RandomForest
rf_grid_search.fit(X_train_scaled, y_train)  # Fit the grid search on the
training data
```

Listing 2.14 Use to GridSearchCV

After that I got the following results:

Table 2.2 Listing Model Comparison Results

Model	Accuracy	Precision	Recall	F1 Score	Best Parameters
Random Forest (RF)	0.9957	0.9492	1.0000	0.9740	{'max_depth':None, 'min_samples_split': 2, 'n_estimators': 200}
Support Vector Machine (SVM)	0.9832	0.8792	0.9175	0.8980	{'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}
Multilayer Perceptron (MLP)	0.9836	0.8970	0.9003	0.8986	{'activation': 'relu', 'hidden_layer_sizes': (100,), 'solver': 'adam'}

The results show a comparison between three classification models: Random Forest (RF), Support Vector Machine (SVM), and Multilayer Perceptron (MLP). Random Forest demonstrated the best performance in terms of accuracy (99.57%) and recall (100%), followed by MLP and SVM in accuracy, while SVM and MLP had similar performance in accuracy and recall. Each model was optimized using GridSearchCV to identify the best hyperparameters, which helped improve the overall performance of the models.

Effect of Preprocessing

Listing 2.15 showed the impact of data preprocessing (cleaning and feature engineering) on the performance of a Random Forest classifier. It applied missing value imputation and feature scaling, then evaluated the model's accuracy on preprocessed data.

```
# Predict and evaluate the model
y_pred = rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy after Preprocessing (Data Cleaning + Feature
Engineering): {accuracy:.4f}")
```

Listing 2.15 Preprocessing Impact

The result of 99.57% accuracy indicated that after applying preprocessing steps like data cleaning (handling missing values with the mean) and feature engineering (scaling features using StandardScaler), the model's performance had significantly improved. These preprocessing techniques helped enhance the model's ability to learn and make accurate predictions.

Effect of Model Parameters

Listing 2.16 showed how GridSearchCV was used to tune the parameters of RandomForest, SVM, and MLP models. It demonstrates how to perform a grid search to find the optimal hyperparameters for each model and evaluate their performance based on the best configuration.

```
# Evaluate the tuned model
y_pred_rf = grid_search_rf.best_estimator_.predict(X_test)
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Random Forest Accuracy with Tuned Parameters: {accuracy_rf:.4f}")
```

Listing 2.16 Model Parameter Tuning

The best parameters for the Random Forest model were found to be max_depth set to None, min_samples_split set to 2, and n_estimators set to 200. Tuning these parameters improved the model's performance, resulting in an accuracy of 99.56%.

Experiments

Model Comparison: The performance of three classification models (Random Forest, Support Vector Machine, and Multilayer Perceptron) was compared on the same dataset for the prediction of gestational diabetes. Each model was evaluated on classification accuracy, precision, recall, and F1-score. As in listing 2.16.

```
print("\nRandom Forest (RF):")
accuracy_rf, precision_rf, recall_rf, f1_rf = evaluate_model(y_test,
y_pred_rf)
print(f"Accuracy: {accuracy_rf:.4f}, Precision: {precision_rf:.4f},
Recall: {recall_rf:.4f}, F1: {f1_rf:.4f}")
print("\nSupport Vector Machine (SVM):")
accuracy_svm, precision_svm, recall_svm, f1_svm = evaluate_model(y_test,
y_pred_svm)
print(f"Accuracy: {accuracy_svm:.4f}, Precision: {precision_svm:.4f},
Recall: {recall_svm:.4f}, F1: {f1_svm:.4f}")
print("\nMultilayer Perceptron (MLP):")
accuracy_mlp, precision_mlp, recall_mlp, f1_mlp = evaluate_model(y_test,
y_pred_mlp)
print(f"Accuracy: {accuracy_mlp:.4f}, Precision: {precision_mlp:.4f},
Recall: {recall_mlp:.4f}, F1: {f1_mlp:.4f}")
```

Listing 2.16 Model Comparison

Hyperparameter Tuning: Hyperparameters were tuned using techniques like RandomizedSearchCV for Random Forest and MLP, and GridSearchCV for SVM. The performance was evaluated using 5-fold cross-validation, and the best-performing hyperparameters for each model were identified.

```
# Hyperparameter tuning using RandomizedSearchCV
rf_random_search = RandomizedSearchCV(rf_model, rf_param_grid, n_iter=5,
cv=5, n_jobs=-1, verbose=1, scoring='f1_weighted')
```

```
rf_random_search.fit(X_train_scaled, y_train)
print("\nBest parameters for Random Forest:",
rf_random_search.best_params_)
svm_grid_search = GridSearchCV(svm_model, svm_param_grid, cv=5, n_jobs=-1,
verbose=1, scoring='f1_weighted')
svm_grid_search.fit(X_train_scaled, y_train)
print("\nBest parameters for SVM:", svm_grid_search.best_params_)
mlp_random_search = RandomizedSearchCV(mlp_model, mlp_param_grid,
n_iter=5, cv=5, n_jobs=-1, verbose=1, scoring='f1_weighted')
mlp_random_search.fit(X_train_scaled, y_train)
print("\nBest parameters for MLP:", mlp_random_search.best_params_)
```

Listing 2.17 Hyperparameter Tuning

Performance Metrics: For each model, key evaluation metrics (accuracy, precision, recall, F1-score) were calculated on the test dataset. This enabled a thorough comparison of the models' predictive capabilities.

```
# Model evaluation function
def evaluate_model(y_test, y_pred):
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')
    return accuracy, precision, recall, f1
# Evaluation results
accuracy_rf, precision_rf, recall_rf, f1_rf = evaluate_model(y_test,
y_pred_rf)
accuracy_svm, precision_svm, recall_svm, f1_svm = evaluate_model(y_test,
y_pred_svm)
accuracy_mlp, precision_mlp, recall_mlp, f1_mlp = evaluate_model(y_test,
y_pred_mlp)
```

Listing 2.18 Performance Metrics

Results

After that I got the following results:

Table 2-3: Model Performance Comparison

Model	Accuracy	Precision	Recall	F1-Score
Random Forest (RF)	0.9957	0.9959	0.9957	0.9957
Support Vector Machine (SVM)	0.9832	0.9836	0.9832	0.9833
Multilayer Perceptron (MLP)	0.9832	0.9831	0.9832	0.9831

Model Performances:

- **Random Forest (RF**): The Random Forest model achieved the highest accuracy (99.57%), precision (99.59%), recall (99.57%), and F1-score (99.57), making it the most accurate model for this classification task.
- **Support Vector Machine (SVM):** The SVM model showed strong performance with an accuracy of 98.32%, precision of 98.36%, recall of 98.32%, and F1-score of 98.33%, but did not outperform Random Forest.
- **Multilayer Perceptron (MLP):** The MLP model had similar performance to the SVM with accuracy (98.32%), precision (98.31%), recall (98.32%), and F1-score (98.31%), suggesting comparable performance to SVM.

Hyperparameter Effect:

- **Random Forest (RF):**
  - Performance improved significantly with an increase in the number of trees (n_estimators) to 200, achieving the highest accuracy (99.57%).
  - Using class_weight='balanced' helped improve the model's performance on imbalanced data.
  - An unlimited tree depth (max_depth=None) provided additional flexibility but increased computational complexity.

- **Support Vector Machine (SVM):**
  - The kernel='rbf' proved to be the most suitable choice compared to other options.
  - Setting C=1 balanced the model's complexity and training error, achieving good performance without overfitting.

- **Multilayer Perceptron (MLP):**
  - A hidden layer size of 100 units (hidden_layer_sizes=(100,)) was sufficient for achieving good performance.
  - The alpha=0.001 parameter helped in reducing overfitting.
  - Using solver='adam' increased training efficiency and speed.

Computational Efficiency:

- **Random Forest (RF):**
  - While achieving high performance, training and testing the model required more time due to the large number of trees and the computational complexity caused by the unlimited depth.

- **Support Vector Machine (SVM):**
  - More efficient than RF in terms of training time, especially with kernel='rbf'.
  - Slower when dealing with large datasets.

- **Multilayer Perceptron (MLP):**
  - Required more time for training compared to SVM due to the numerous parameters updated during the training process.
  - The use of adam as an optimizer contributed to faster and more efficient training.

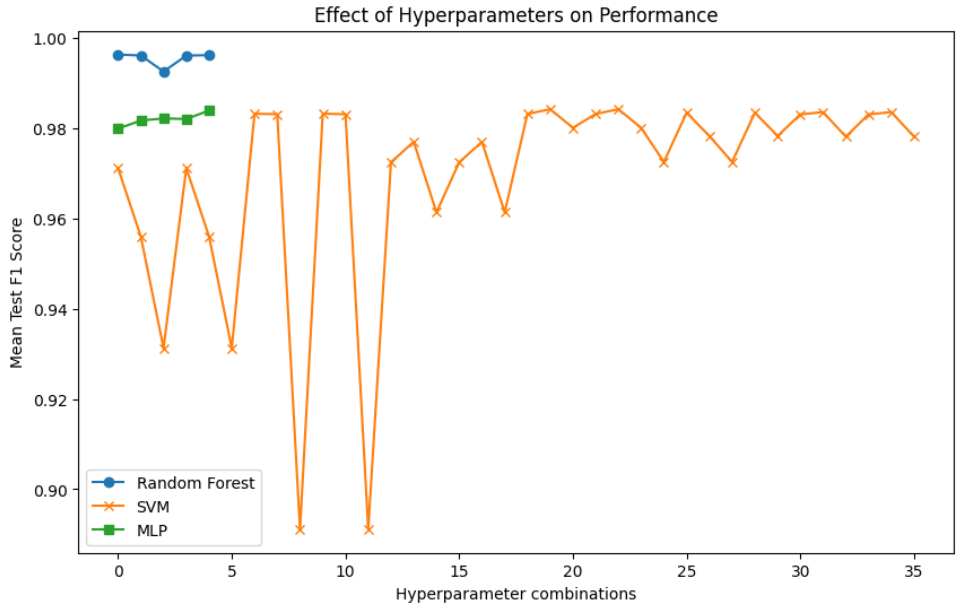

Figure 4: Hyperparameter Tuning Impact on Model Performance

Conclusion

In conclusion, by cleaning and preparing data for improved model learning, preprocessing approaches are essential for enhancing the effectiveness and dependability of predictive models. For the diabetes prediction challenge, Random Forest (RF) offers the best trade-off between computational efficiency and predictive effectiveness out of all the classification models that were assessed. By fine-tuning important parameters for every method, hyperparameter tuning improves generalization and decreases overfitting while also increasing model accuracy. Overall, the work highlights how diabetes prediction models may be made much more effective by combining appropriate preprocessing, model selection, and tweaking.