Mini Project

Data Science Fundamentals

DISEASE RECOGNITION USING SYMPTOMS

Table of Contents

1. Introduction

1.1 Overview

The process of disease diagnosis is often time-sensitive and complex, especially when multiple diseases share similar symptoms. "Disease Recognition Using Symptoms" is a machine learning-based project aimed at developing an efficient system for predicting diseases based on observed symptoms. The project leverages a structured binary dataset of symptoms mapped to various diseases (prognoses). By analyzing this data, the system seeks to assist healthcare professionals in recognizing potential diagnoses, reducing diagnostic time and error rates.
This project explores the application of data preprocessing, imputation techniques, and machine learning classifiers to build a framework that is both reliable and interpretable for medical use cases.

Objectives

The primary goal of this project is to develop a machine learning-based disease prediction model that accurately identifies potential diagnoses based on a set of symptoms. This involves understanding patterns in symptom-disease associations and optimizing machine learning methods to create a reliable, user-friendly prediction system. The specific objectives of the project include:

1. **Understanding Symptom-Disease Associations**: Analyze patterns within the dataset to identify how individual symptoms or combinations of symptoms contribute to different disease outcomes.
2. **Data Imputation and Preprocessing**: Handle missing or incomplete data entries in the dataset by applying imputation strategies (minimum, maximum, or mode imputation). Ensure that the dataset is clean and ready for model training.
3. **Model Development**: Build and train machine learning models to predict diseases based on input symptoms. Prioritize models that balance accuracy with interpretability, ensuring that medical practitioners and stakeholders can trust the predictions.
4. **Evaluation and Comparison**: Evaluate the performance of different classifiers on the original and imputed datasets. Compare models based on key metrics such as accuracy, precision, recall, and F1-score to select the most suitable approach.
5. **Practical Application**: Design a framework that can be extended for use in real-world diagnostic systems or healthcare applications, enabling faster and more efficient disease recognition.
6. **Addressing Challenges**: Tackle challenges such as overlapping symptoms, high-dimensional symptom data, and handling categorical target variables (prognosis) effectively to ensure robustness of the system.

1.2 Problem Definition and Scope

Accurate and timely disease diagnosis remains a critical challenge in the healthcare sector, especially when symptoms overlap or patient information is incomplete. Misdiagnosis or delays can lead to improper treatment, worsening health outcomes, and increased costs. Manual diagnostic methods are often time-consuming, prone to human error, and difficult to scale.

Traditional approaches rely heavily on healthcare professionals, but these methods may not fully capture the intricate relationships between symptoms and diseases. Overlapping symptoms, like fever and fatigue, complicate diagnoses without extensive testing. Additionally, issues such as incomplete records and high-dimensional symptom-disease data further hinder the diagnostic process.

This project introduces a **machine learning-based approach** to disease recognition using historical symptom-disease datasets. Predictive models analyze symptoms to provide accurate and interpretable diagnoses, assisting healthcare providers in reducing diagnostic time and improving accuracy.

Scope of the Project

1. **Scalable Diagnosis Framework**: A machine learning system capable of predicting diseases from diverse symptoms, addressing high-dimensional and overlapping symptom challenges.
2. **Data Imputation**: Handles missing symptom data through techniques like minimum, maximum, and mode-based imputations for better predictive modeling.
3. **Healthcare Efficiency**: Automates disease recognition to support practitioners, especially in resource-limited settings.
4. **Real-World Applications**: Potential to integrate into healthcare tools like mobile diagnostic apps or clinical decision-making systems.

Challenges Addressed

1. **Overlapping Symptoms**: Many diseases share similar symptoms, making differentiation difficult. The model identifies patterns to improve diagnostic accuracy.
2. **Missing Data**: Tackles incomplete datasets using robust imputation techniques for consistent model performance.
3. **High Dimensionality**: Addresses the complexity of datasets with feature importance analysis for efficient predictions.
4. **Categorical Prognosis**: Uses appropriate classification techniques to handle the categorical target variable effectively.
5. **Model Optimization**: Compares and optimizes classifiers to identify the most accurate and reliable model.
6. **Scalability and Usability**: Ensures the system is scalable for large datasets and provides user-friendly, interpretable predictions.

By addressing these challenges, this project contributes to advancing healthcare diagnostics through an efficient, scalable, and data-driven solution.

2. Data Preparation

2.1 Dataset Used

The **Disease Symptoms Dataset** is a collection of disease symptoms and their corresponding diagnoses, used for predicting diseases based on symptom presence or absence.

- **Dataset Name**: Disease Symptoms Dataset
- **Source**: University of Columbia at New York Presbyterian Hospital (2004)
- **Owner**: Open-access for academic and research purposes

Dataset Characteristics:

- **Rows**: 4,962 (each representing a patient)
- **Columns**: 133 (132 for symptoms, 1 for prognosis)
- **File Size**: ~5 MB
- **Memory Usage**: Efficient for most data analysis tools

Column Descriptions:

1. **Symptoms (132 columns)**:
    - Binary (0 = absence, 1 = presence)
    - Example symptoms: Itching, Skin Rash, Shivering, Stomach Pain, etc.
2. **Prognosis (1 column)**:
    - Categorical (target disease)
    - Example diseases: Dimorphic Hemorrhoids, Urinary Tract Infection, Drug Reaction, etc.

Relevance:

This dataset is useful for building classification models to predict diseases based on symptoms, with a broad range of symptoms making it ideal for training disease-prediction models.

Additional Insights:

- **Target Variable**: Prognosis (disease).
- **Feature Composition**: 132 numerical features (symptoms) and 1 categorical feature (disease).
- **Class Imbalances**: Some diseases may occur more frequently than others.

3. Exploratory Data Analysis

3.1 Getting Insights About the Dataset

The first step in the EDA process is to explore and understand the dataset's structure and relationships. The following operations were applied during data manipulation:

Dataset Overview: Displayed the first few rows to understand the structure and basic composition of the data.

Descriptive Statistics: Calculated summary statistics like mean, standard deviation, min, max, and median values for the symptoms columns to understand their distribution.

Class Distribution: Checked the frequency of diseases in the Prognosis column to identify any class imbalances.

Unique Values: Counted the unique values in each symptom column to verify their binary nature (0 or 1).

3.2 Handling Missing Values

1. **Injecting Missing Values**:
    - To simulate real-world scenarios, we intentionally introduced missing values into the dataset. Approximately 30% of the data entries in the columns were randomly replaced with NaN (Not a Number) values.
2. **Imputation Strategies**:
    - After introducing missing values, we applied three different imputation strategies to fill the missing data. These strategies resulted in the creation of three separate datasets, each imputed differently:
        - **Minimum Imputation**: In the first dataset, missing values were replaced with the **minimum value** from the respective columns.
        - **Maximum Imputation**: In the second dataset, missing values were replaced with the **maximum value** from the respective columns.
        - **Mode Imputation**: In the third dataset, missing values were replaced with the **mode** (most frequent value) of each respective column.
3. **Resulting Datasets**:
    - Each of the three imputation strategies produced a distinct dataset. These three imputed datasets along with one original were then used for further analysis and model training to evaluate the impact of different imputation methods on the overall results.

3.3 Data Encoding

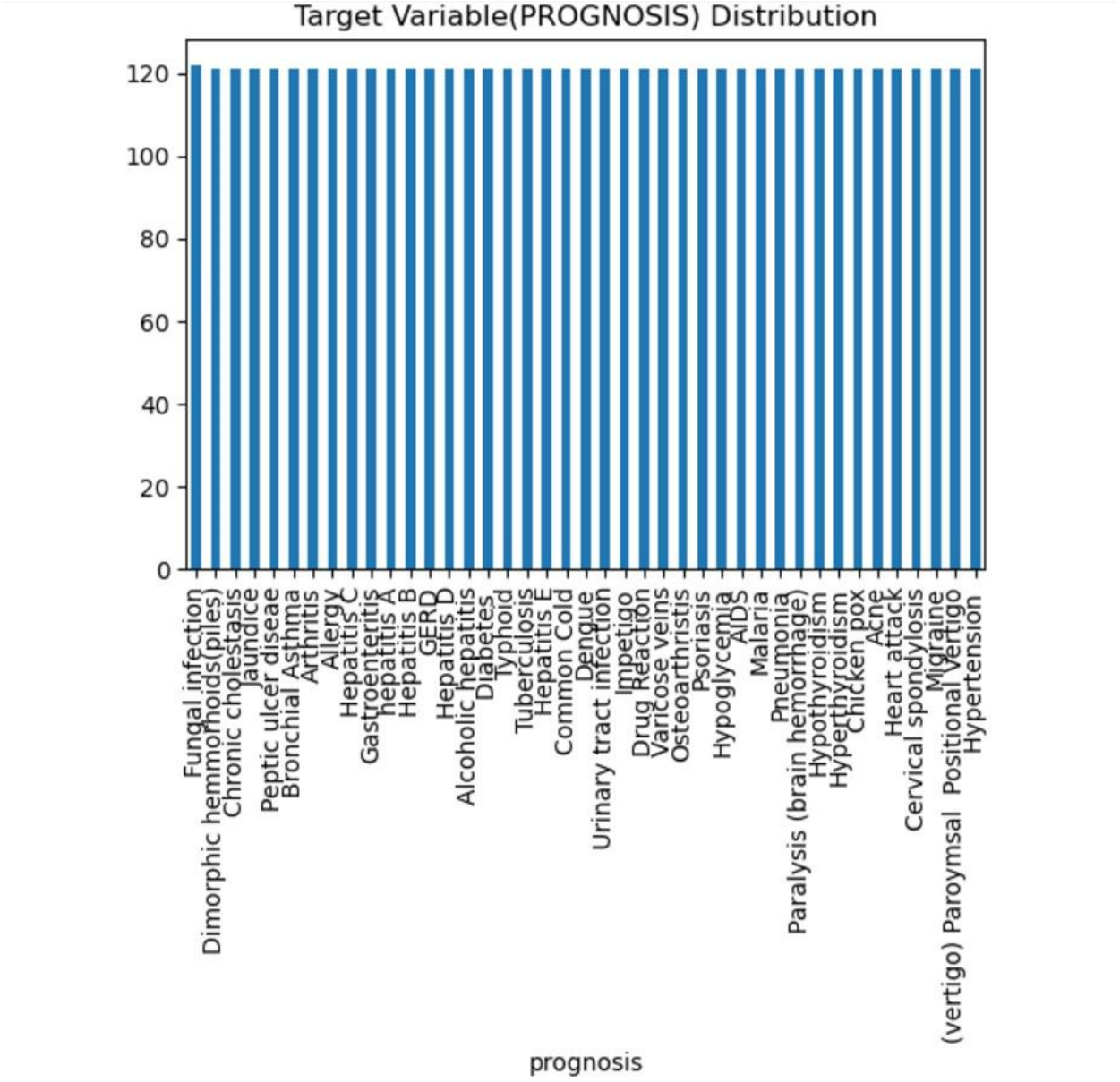
Target Encoding (Prognosis):

Prognosis column, which contains categorical disease names, was converted to numerical labels using **Label Encoding**. Each unique disease was assigned a unique integer value.

3.4 Data visualization

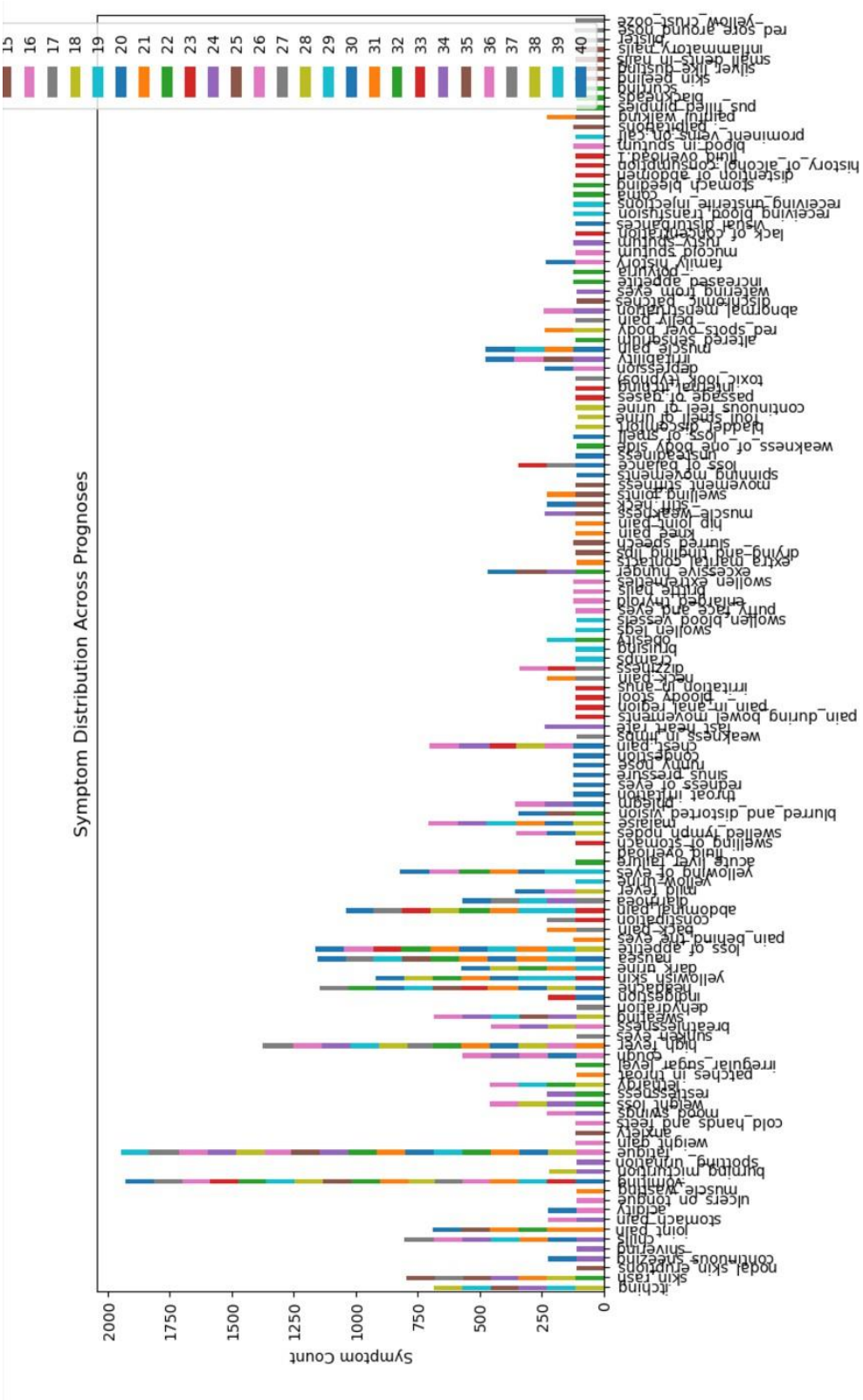
To gain insights into the dataset, two key visualizations were created:

1. **Target Variable Distribution**:
    - A bar chart was used to analyze the distribution of the target variable (*Prognosis*). This helped identify any imbalances in the dataset and determine whether certain diseases were over- or underrepresented.



2. **Symptom Distribution Across Prognoses:**

   - A stacked bar chart was plotted to show how symptoms are distributed across different prognoses. This visualization provided a clear view of the relationship between symptoms and their corresponding diseases.



4. Methods Used

1. Support Vector Machine (SVM)

SVM identifies the optimal hyperplane that separates different diseases in the symptom feature space. It uses kernels, like the radial basis function (RBF), to map data points into higher dimensions, enabling it to handle non-linear relationships between symptoms and diseases. In disease classification, SVM is effective in finding complex patterns, making it well-suited for scenarios where diseases exhibit overlapping or subtle symptom variations.
Limitation: SVM can be computationally expensive and slow for large datasets, as it requires solving complex optimization problems. Additionally, it may struggle with noisy datasets or when the classes are not well-separated, requiring careful tuning of hyperparameters.

2. K-Nearest Neighbors (KNN)

KNN classifies diseases by comparing the symptoms of a test sample to its k nearest neighbors in the dataset. It determines the class (disease) based on a majority vote of the neighbors. KNN works effectively when diseases share overlapping symptoms, as it uses proximity to make predictions. It is simple and intuitive, requiring no explicit training phase, making it a good baseline classifier.
Limitation: KNN becomes computationally expensive as the dataset grows, as it requires storing the entire training set and calculating distances for each prediction. It is also sensitive to the choice of k and struggles with high-dimensional data, where the concept of "distance" becomes less meaningful (curse of dimensionality).

3. Decision Tree

Decision Trees split the dataset based on symptom values, forming a tree-like structure where each internal node represents a symptom-based decision, and each leaf represents a disease class. For example, the tree might first split based on whether a patient has "fever," then branch further based on "cough" or "rash." Decision Trees are highly interpretable, making it easy to understand the symptom patterns leading to disease predictions.
Limitation: Decision Trees are prone to overfitting, especially when they grow too deep or are built without pruning. They can also be sensitive to small variations in the data, leading to different tree structures. When irrelevant features are present, the model's performance may decline.

4. Naïve Bayes

Naïve Bayes calculates the probability of a disease given the observed symptoms using Bayes' Theorem. It assumes conditional independence between symptoms, meaning each symptom contributes independently to the likelihood of a disease. Despite its simplicity, Naïve Bayes performs well in many real-world scenarios, especially when symptoms are somewhat independent or the dataset is small. It is computationally efficient and provides quick predictions.
Limitation: The assumption of conditional independence between symptoms can be

unrealistic in many cases, as symptoms for diseases are often correlated. This limitation can lead to less accurate predictions when the dataset exhibits strong interdependencies between features.

5. Neural Networks

Neural Networks consist of layers of interconnected nodes (neurons), where each neuron processes input features (symptoms) through weights and biases, followed by a non-linear activation function. The network learns complex, non-linear relationships between symptoms and diseases through backpropagation. Neural Networks are especially powerful in capturing intricate patterns in datasets with large numbers of features, making them suitable for disease prediction tasks where symptoms interact in complex ways.
Limitation: Neural Networks require a large amount of data and computational resources to train effectively. They are also less interpretable compared to other models, making it difficult to understand how specific symptoms contribute to the prediction. This can be a drawback in critical fields like healthcare.

6. Random Forest Classifier

Random Forest is an ensemble method that builds multiple Decision Trees during training, using random subsets of the dataset and features. This randomness reduces overfitting and increases robustness. For predictions, Random Forest takes the majority vote across all trees. In disease prediction, Random Forest handles noisy data and complex symptom interactions effectively, providing stable and accurate results.
Limitation: While Random Forest reduces overfitting compared to a single Decision Tree, it can still overfit if the number of trees or depth is not properly tuned. It also requires more computational resources and memory, and the ensemble nature can make predictions slower compared to simpler models.

5. Results and Evaluation

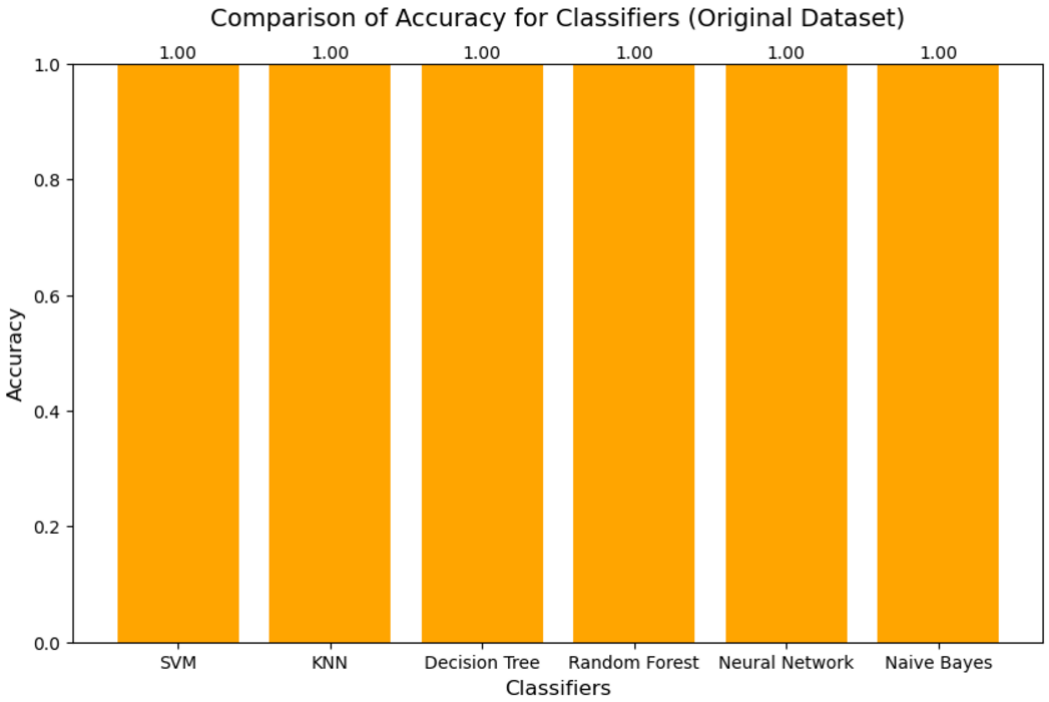
- **Results :**

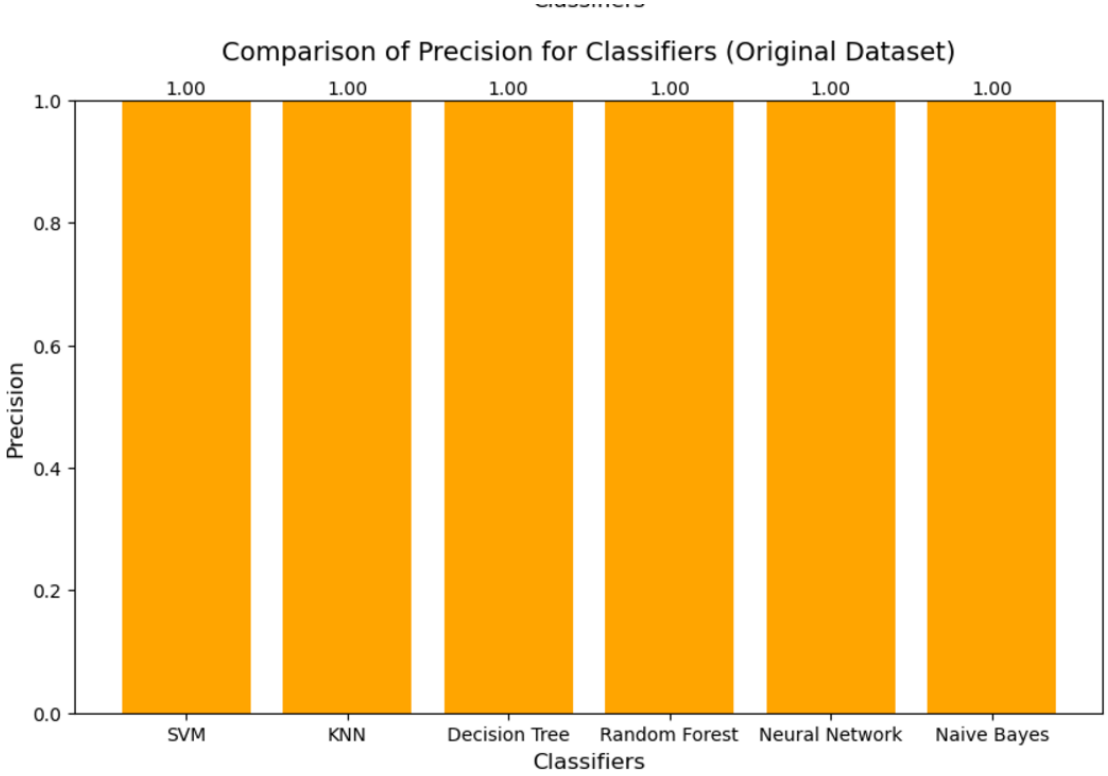
  The datasets were divided into training (80%) and testing (20%) sets. The performance of the following classification models—Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Decision Tree (DT), Naïve Bayes (NB), Neural Networks (NN), and Random Forest—was evaluated on the test set. Below are the results:

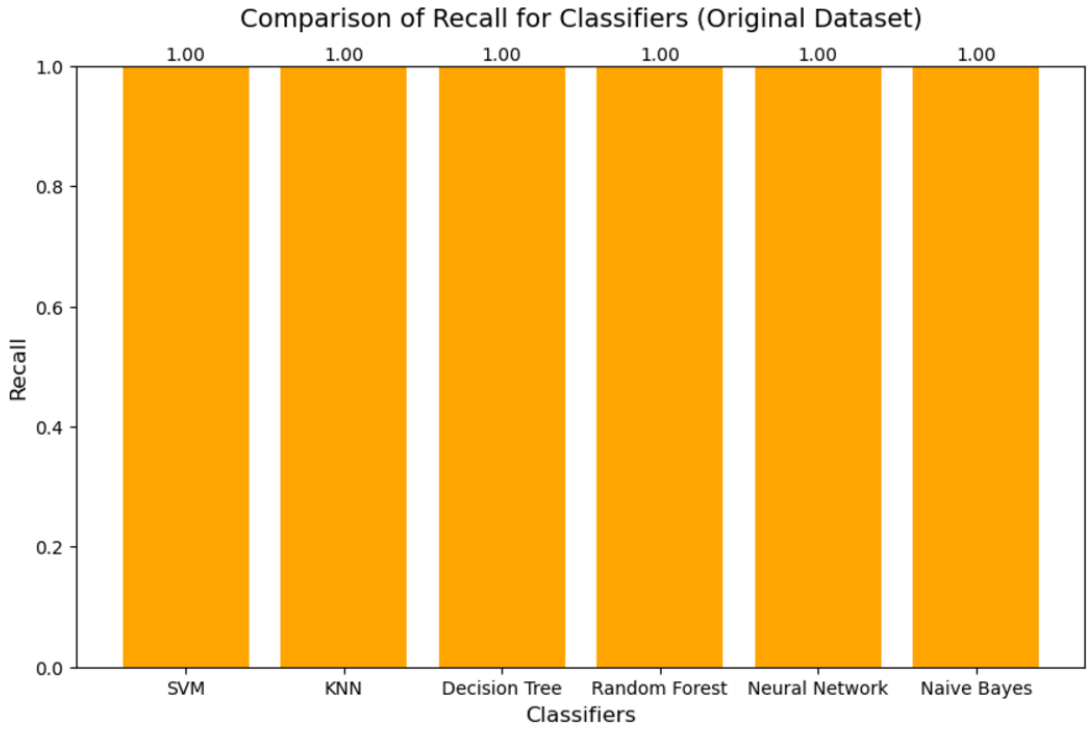
1. ORIGINAL DATASET:

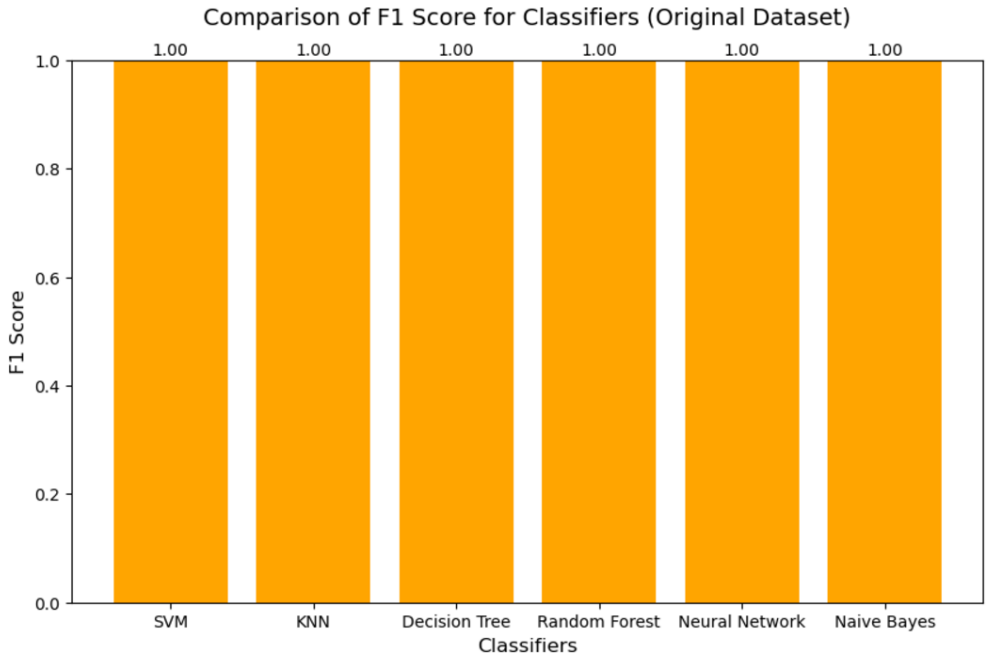
Classifier Metrics (Original Dataset):

	Classifier	Accuracy	Precision	Recall	F1 Score
0	SVM	1.000	1.000	1.000	1.000
1	KNN	1.000	1.000	1.000	1.000
2	Decision Tree	0.999	0.999	0.999	0.999
3	Random Forest	0.999	0.999	0.999	0.999
4	Neural Network	0.999	0.999	0.999	0.999
5	Naive Bayes	1.000	1.000	1.000	1.000




Comparison of Precision for Classifiers (Original Dataset)
1.00
1.00
1.00
1.00
1.00
1.00
1.0
0.8
0.6
0.4
0.2
0.0
Precision
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes
Classifiers

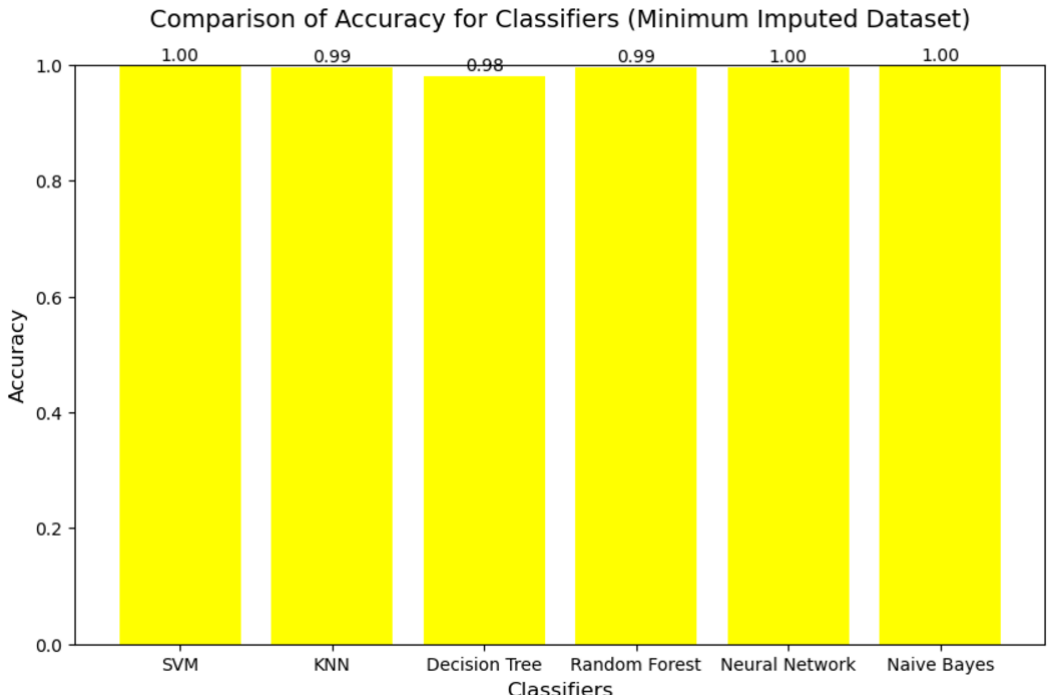

Comparison of Recall for Classifiers (Original Dataset)
1.00
1.00
1.00
1.00
1.00
1.00
1.0
0.8
0.6
0.4
0.2
0.0
Recall
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes
Classifiers

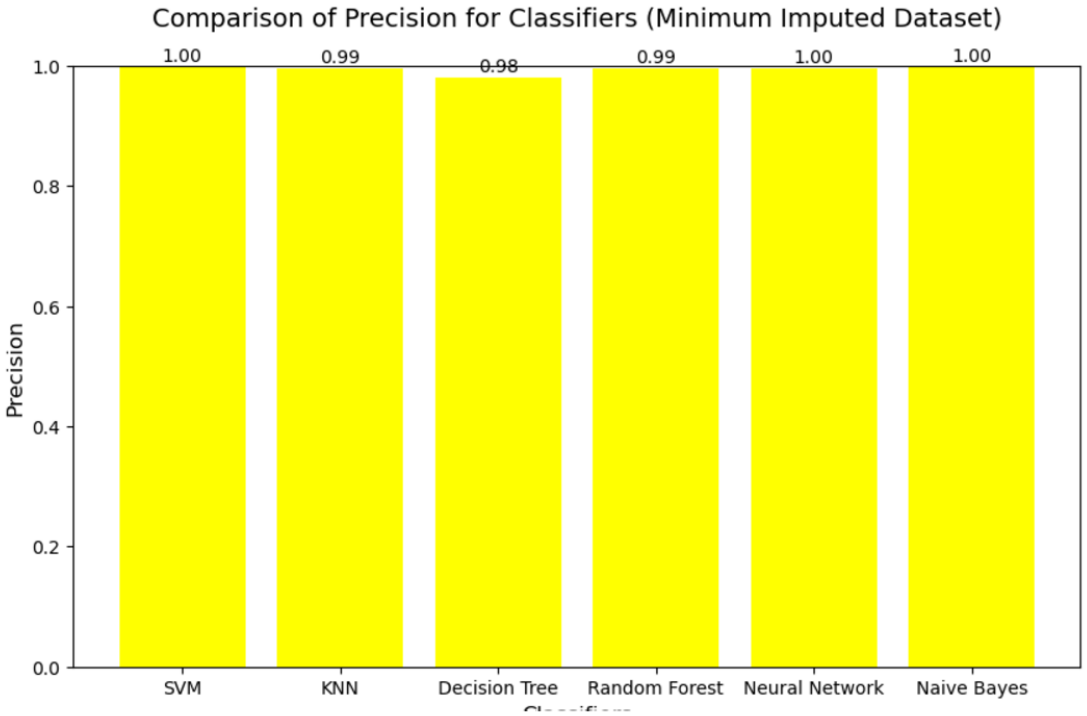


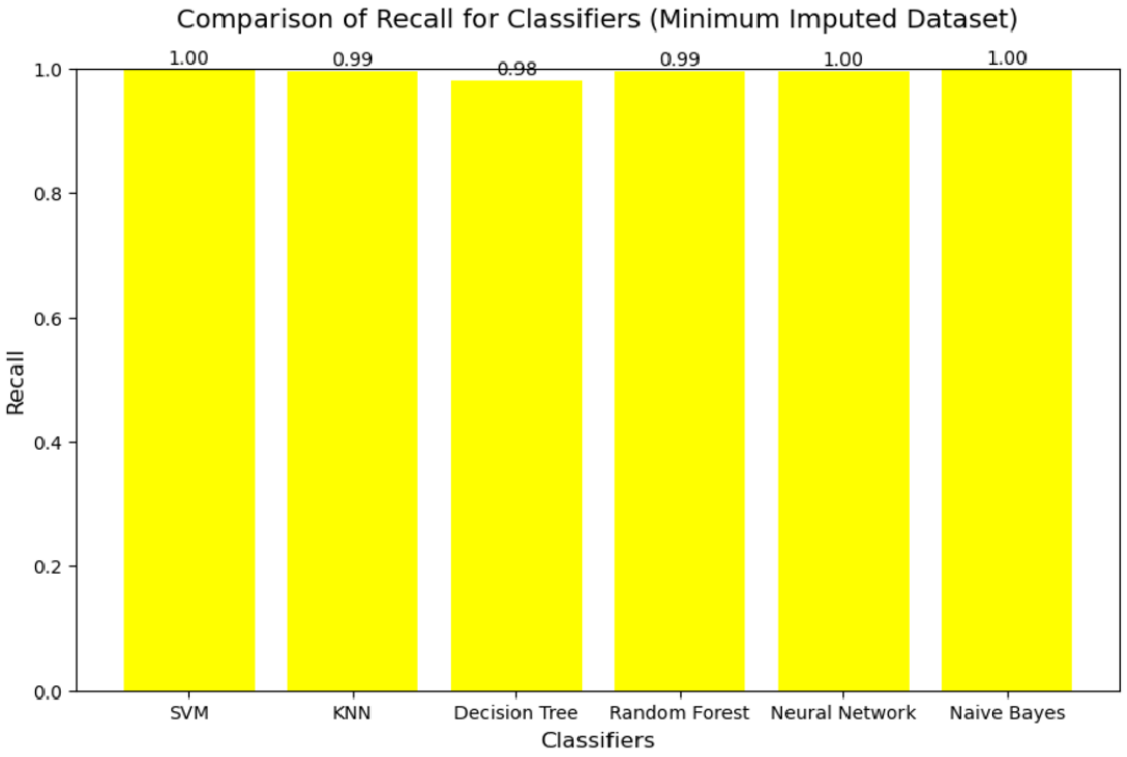
2. MIN IMPUTED DATASET:

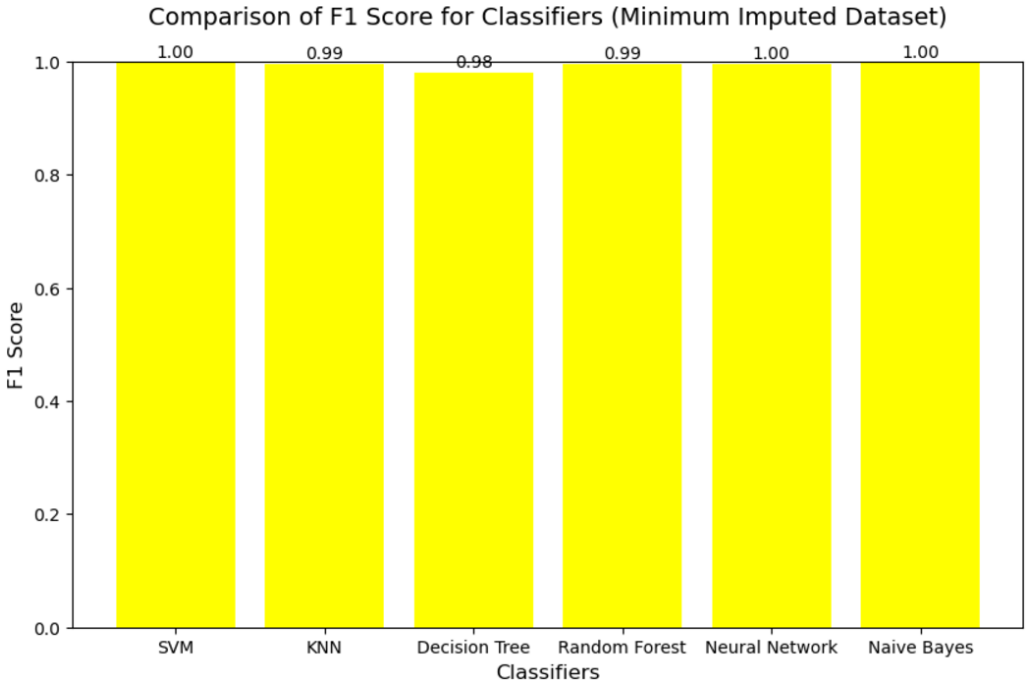
Classifier Metrics (Minimum Imputed Dataset):

	Classifier	Accuracy	Precision	Recall	F1 Score
0	SVM	0.997	0.997	0.997	0.997
1	KNN	0.994	0.994	0.994	0.994
2	Decision Tree	0.979	0.980	0.979	0.979
3	Random Forest	0.995	0.995	0.995	0.995
4	Neural Network	0.996	0.996	0.996	0.996
5	Naive Bayes	0.997	0.997	0.997	0.997




Comparison of Precision for Classifiers (Minimum Imputed Dataset)
Precision
1.00
0.99
0.98
0.99
1.00
1.00
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes

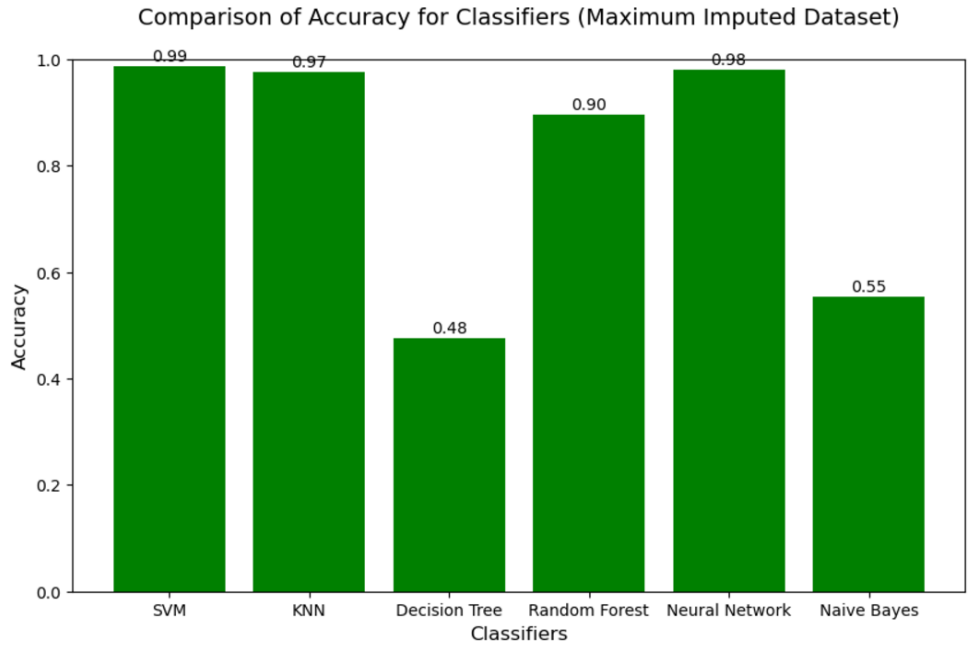

Comparison of Recall for Classifiers (Minimum Imputed Dataset)
Recall
1.00
0.99
0.98
0.99
1.00
1.00
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes
Classifiers

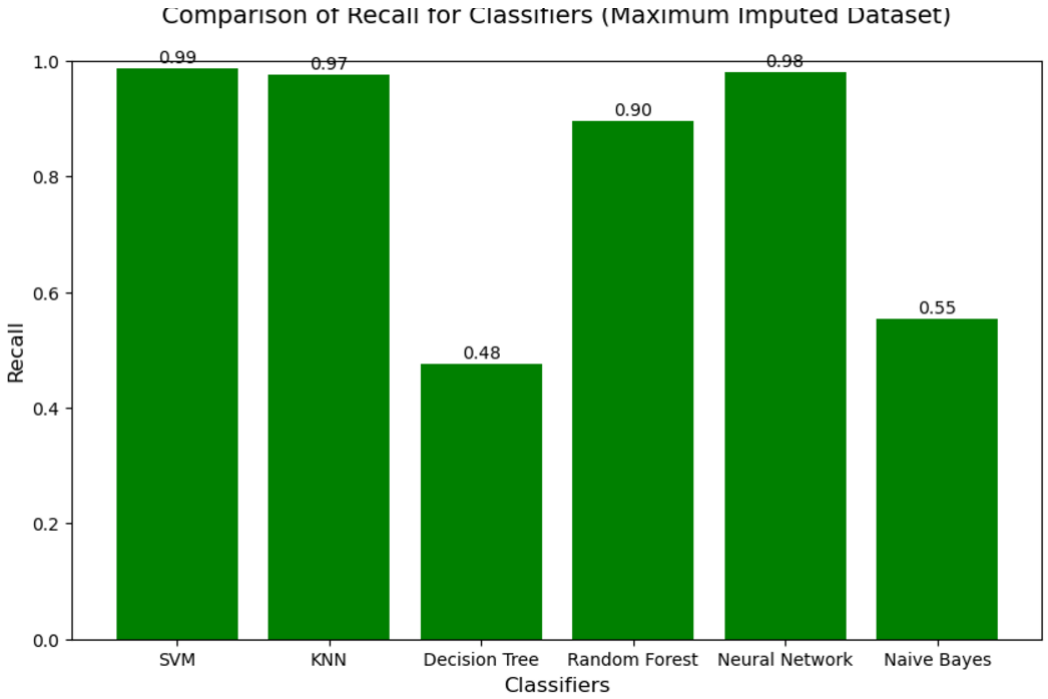


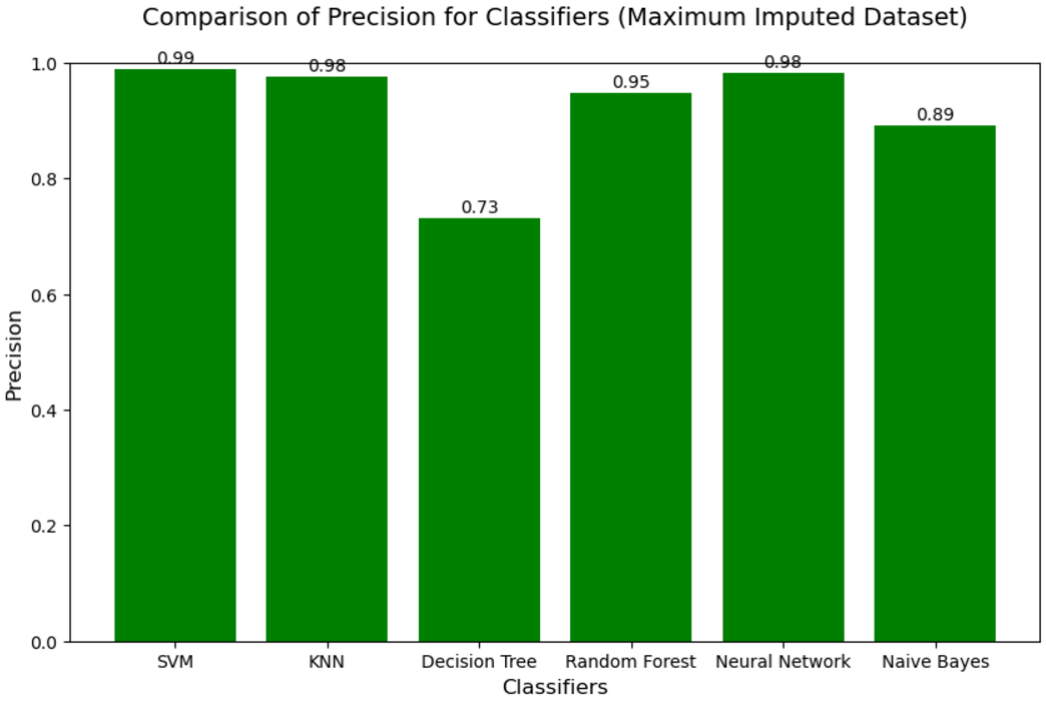
3. MAX IMPUTED DATASET:

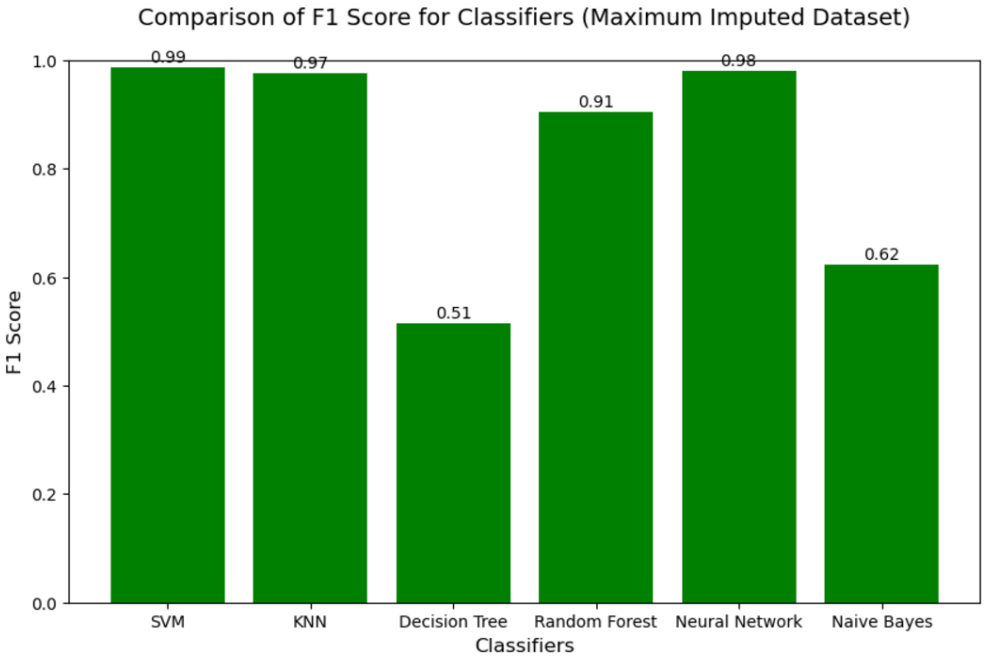
Classifier Metrics (Maximum Imputed Dataset):

	Classifier	Accuracy	Precision	Recall	F1 Score
0	SVM	0.987	0.988	0.987	0.987
1	KNN	0.975	0.976	0.975	0.975
2	Decision Tree	0.476	0.730	0.476	0.514
3	Random Forest	0.896	0.948	0.896	0.905
4	Neural Network	0.980	0.982	0.980	0.980
5	Naive Bayes	0.554	0.891	0.554	0.622




Comparison of Recall for Classifiers (Maximum Imputed Dataset)
Recall
Classifiers
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes
0.99
0.97
0.48
0.90
0.98
0.55
0.0
0.2
0.4
0.6
0.8
1.0

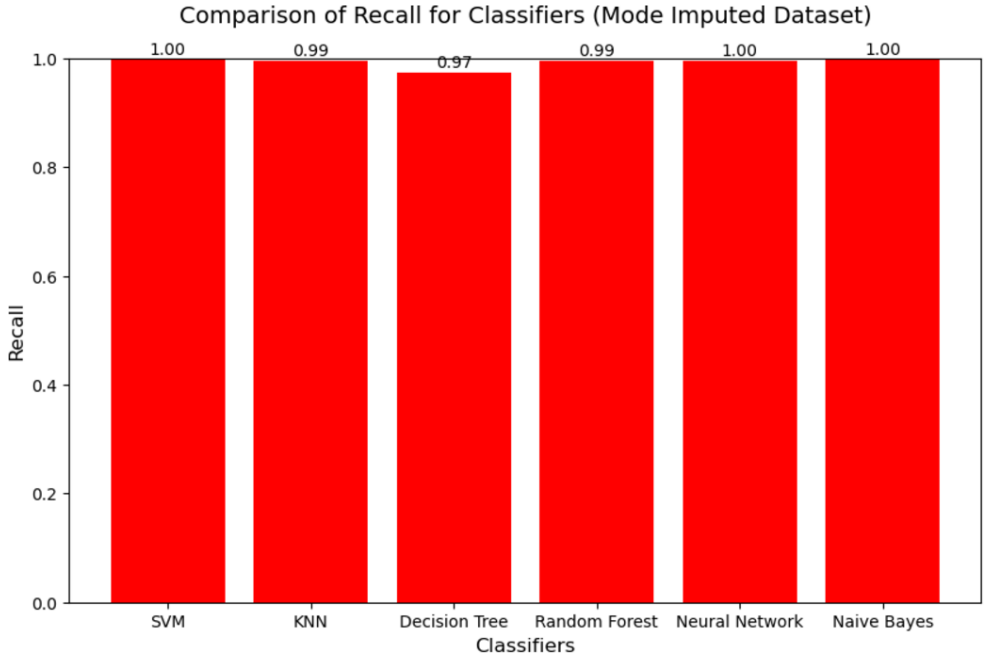

Comparison of Precision for Classifiers (Maximum Imputed Dataset)
Precision
Classifiers
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes
0.99
0.98
0.73
0.95
0.98
0.89
0.0
0.2
0.4
0.6
0.8
1.0

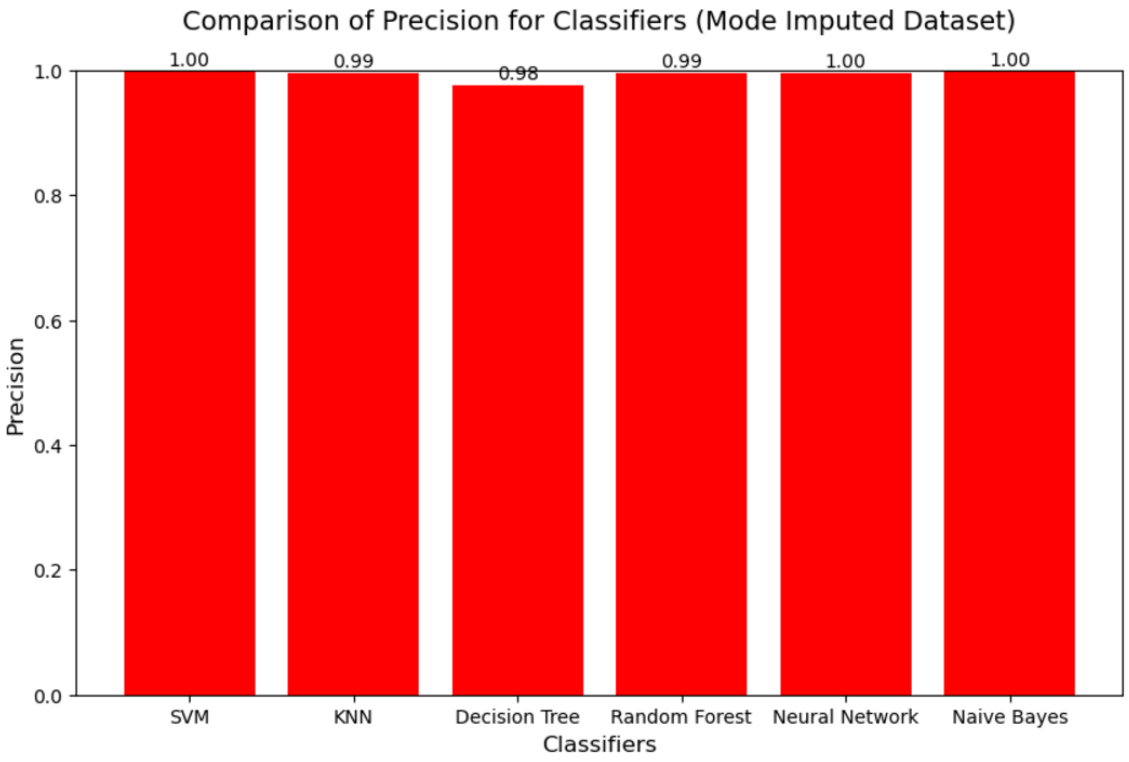


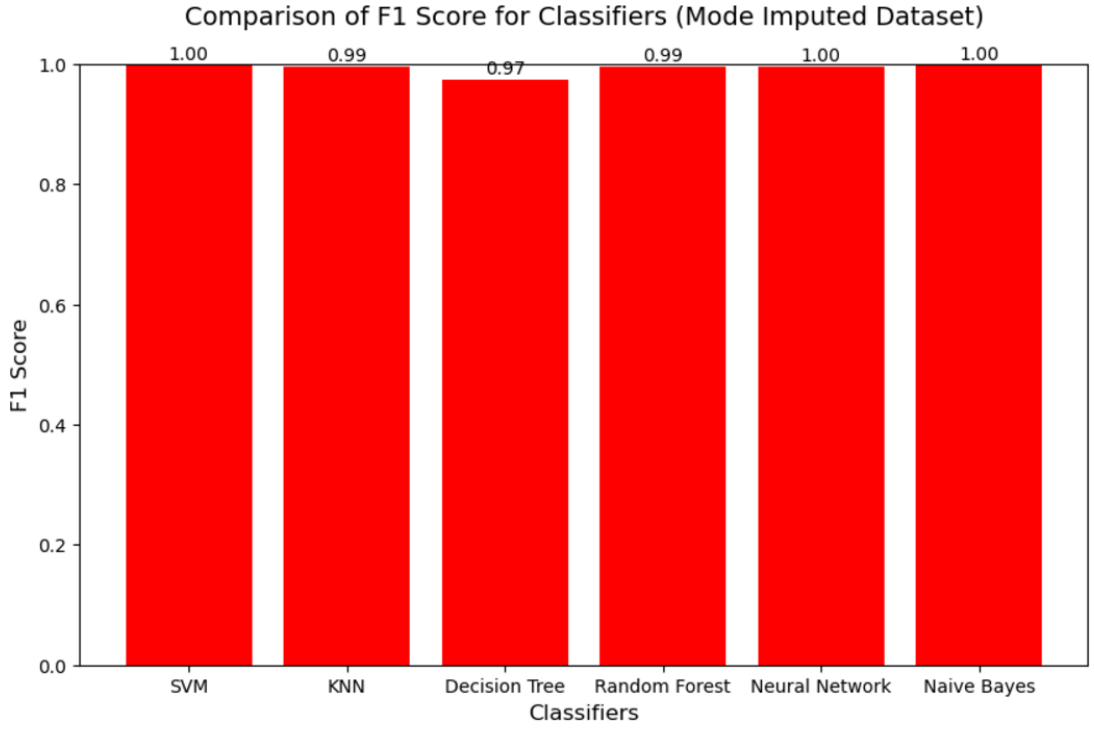
4. MODE IMPUTED DATASET:

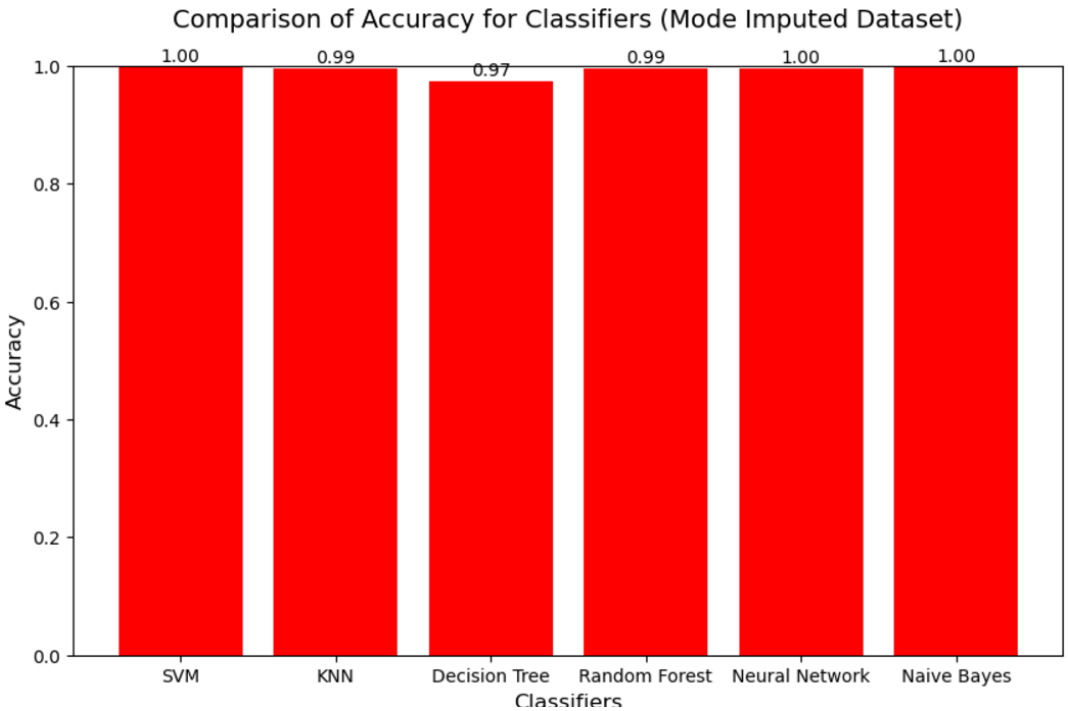
Classifier Metrics (Mode Imputed Dataset):

	Classifier	Accuracy	Precision	Recall	F1 Score
0	SVM	0.997	0.997	0.997	0.997
1	KNN	0.994	0.994	0.994	0.994
2	Decision Tree	0.974	0.976	0.974	0.974
3	Random Forest	0.994	0.994	0.994	0.994
4	Neural Network	0.996	0.996	0.996	0.996
5	Naive Bayes	0.997	0.997	0.997	0.997




Comparison of Precision for Classifiers (Mode Imputed Dataset)
Precision
Classifiers
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes
1.00
0.99
0.98
0.99
1.00
1.00
0.0
0.2
0.4
0.6
0.8
1.0


Comparison of F1 Score for Classifiers (Mode Imputed Dataset)
F1 Score
Classifiers
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes
1.00
0.99
0.97
0.99
1.00
1.00
0.0
0.2
0.4
0.6
0.8
1.0


Comparison of Accuracy for Classifiers (Mode Imputed Dataset)
1.00
0.99
0.97
0.99
1.00
1.00
1.0
0.8
0.6
0.4
0.2
0.0
Accuracy
SVM
KNN
Decision Tree
Random Forest
Neural Network
Naive Bayes
Classifiers

6. Conclusion

This project focused on comparing the performance of various classifiers on a disease recognition dataset, evaluating them on the original dataset as well as on three imputed datasets (minimum, maximum, and mode imputed). Based on the analysis, the following conclusions can be drawn:

1. **Performance on the Original Dataset:**
   All classifiers—SVM, KNN, Decision Tree, Random Forest, Neural Network, and Naive Bayes—achieved exceptionally high performance metrics, with accuracies ranging from 0.999 to 1.000. This indicates that the dataset is well-structured, and the classifiers were able to accurately map the relationship between symptoms and diseases without any missing values. SVM, KNN, and Naive Bayes stood out with perfect scores in all metrics, showcasing their effectiveness in the original dataset.

2. **Performance on Imputed Datasets:**

   - **Minimum Imputed Dataset:**
     The classifiers maintained robust performance, with SVM, Naive Bayes, and Neural Network showing the highest accuracy (0.997). KNN and Random Forest also performed strongly with accuracies above 0.99. Decision Tree, however, exhibited a slightly lower accuracy (0.979), indicating sensitivity to missing value imputation using the minimum strategy.

   - **Maximum Imputed Dataset:**
     This dataset introduced more significant variability in classifier performance. SVM and Neural Network continued to deliver high accuracy (0.987 and 0.980, respectively), but Decision Tree accuracy dropped drastically to 0.476, showing that this imputation method greatly impacted its reliability. Random Forest also showed reduced accuracy (0.896), highlighting its sensitivity to this imputation method. Naive Bayes struggled with accuracy (0.554) but retained a high precision score (0.891), indicating potential biases in handling certain classes.

   - **Mode Imputed Dataset:**
     Similar to the minimum imputation method, the mode imputed dataset yielded high performance across classifiers. SVM, Naive Bayes, and Neural Network achieved an accuracy of 0.997 or higher, while KNN and Random Forest also performed well (0.994). Decision Tree again showed slightly lower accuracy (0.974) compared to other models, but still performed better than on the maximum imputed dataset.

3. **Insights on Classifier Robustness:**

   - **SVM, Neural Network, and Naive Bayes:** These classifiers demonstrated consistently high performance across all datasets, indicating their robustness and adaptability to missing value imputation methods.

   - **KNN and Random Forest:** Both performed well overall, but their accuracy showed minor drops with certain imputation methods, highlighting moderate sensitivity to how missing data is handled.

   - **Decision Tree:** This model was the most sensitive to imputation strategies, with its performance dropping significantly in the maximum imputed dataset. This highlights

the need for careful feature engineering and imputation strategies when using Decision Trees.

4. **Overall Findings:**
   The project demonstrates that SVM, Neural Network, and Naive Bayes are the most reliable classifiers for disease recognition tasks in this dataset. These models consistently achieved high accuracy, precision, recall, and F1 scores regardless of missing value handling. Decision Tree, while interpretable and effective on the original dataset, requires careful tuning and data preprocessing to maintain performance with imputed datasets.

2 Future Scope

Future work could explore advanced imputation methods (e.g., iterative imputation or machine learning-based imputation) to further enhance classifier performance. Additionally, expanding the dataset and testing on real-world data with larger sample sizes could provide deeper insights into the scalability and robustness of these models. This study provides a solid foundation for implementing machine learning models in disease recognition tasks, with practical implications for healthcare applications.

7. References

- Scikit-learn Documentation. Machine Learning in Python. Retrieved from https://scikit-learn.org, consulted for implementing Support Vector Machines, Decision Trees, Random Forest Classifier, K-Nearest Neighbors, and Naïve Bayes models.
- TensorFlow/Keras Documentation. Deep Learning Framework for Neural Networks. Retrieved from https://www.tensorflow.org, referenced for training Neural Networks used in the analysis.
- Python Official Documentation. Libraries for Data Science. Retrieved from https://www.python.org, used for exploring Pandas, NumPy, and Matplotlib for data manipulation and visualization.
- Google Search. Accessed via https://www.google.com, utilized for exploring classification techniques, preprocessing, and performance metrics.
- Brownlee, J. (2020). *Introduction to Classification*. Retrieved from https://machinelearningmastery.com, a guide for understanding classification techniques and evaluation metrics.
- Matplotlib Documentation. Visualization Tools for Data Science. Retrieved from https://matplotlib.org, used for creating performance charts.

Annexure 1:

LOADING THE DATASET

```
import pandas as pd
import numpy as np

file_path = r"C:\Users\Dell\Downloads\diseases.csv"

df_original= pd.read_csv(file_path)
df_original.head()
```

ANALYZING THE DISTRIBUTION OF DATA

```
import seaborn as sns
import matplotlib.pyplot as plt
# Analyze the distribution of the target variable
df_original['prognosis'].value_counts().plot(kind="bar", figsize=(6, 4))
plt.title("Target Variable(PROGNOSIS) Distribution")
plt.show()
grouped_data = df_original.groupby('prognosis').sum()
grouped_data.T.plot(kind='bar', stacked=True, figsize=(14, 8), title="Symptom Distribution Across Prognoses")
plt.ylabel("Symptom Count")
plt.show()
```

LABEL ENCODING THE "PROGNOSIS" COLUMN

```
from sklearn.preprocessing import LabelEncoder

# Encode the 'diseases' column
encoder = LabelEncoder()
df_original['prognosis'] = encoder.fit_transform(df_original['prognosis'])

# Save the encoding mappings
label_mapping = dict(zip(encoder.classes_, encoder.transform(encoder.classes_)))
print("Label encoding mapping for prognosis:", label_mapping)
```

CREATING NEW DATASETS AFTER IMPUTATION

```
# Exclude the last column (label-encoded prognosis) from modification
prognosis_col = df_original.columns[-1]

# Randomly introduce NaN values into non-prognosis columns
np.random.seed(42)  # For reproducibility
nan_probability = 0.1  # 10% of the values will be set to NaN

# Create a copy of the original dataset to introduce NaN values
df_with_nulls = df_original.copy()

# Introduce NaN values in non-prognosis columns
for col in df_with_nulls.columns[:-1]:  # Exclude prognosis column
    df_with_nulls.loc[df_with_nulls.sample(frac=nan_probability).index, col] = np.nan

# Impute missing values using Min, Max, and Mode strategies
df_min_imputed = df_with_nulls.fillna(df_with_nulls.min(numeric_only=True))
df_max_imputed = df_with_nulls.fillna(df_with_nulls.max(numeric_only=True))
df_mode_imputed = df_with_nulls.fillna(df_with_nulls.mode().iloc[0])

# Inspect the imputed datasets
print("Min Imputed Dataset:\n", df_min_imputed.head())
print("Max Imputed Dataset:\n", df_max_imputed.head())
print("Mode Imputed Dataset:\n", df_mode_imputed.head())
```

Train-Test Split:

```
from sklearn.model_selection import train_test_split

# Ensure there are no null values in the datasets
assert not df_original.isnull().values.any(), "df_original contains null values!"
assert not df_min_imputed.isnull().values.any(), "df_min_imputed contains null values!"
assert not df_max_imputed.isnull().values.any(), "df_max_imputed contains null values!"
assert not df_mode_imputed.isnull().values.any(), "df_mode_imputed contains null values!"

# Separate features (X) and target (y) for each dataset
X_original = df_original.iloc[:, :-1]
y_original = df_original.iloc[:, -1]

X_min_imputed = df_min_imputed.iloc[:, :-1]
y_min_imputed = df_min_imputed.iloc[:, -1]

X_max_imputed = df_max_imputed.iloc[:, :-1]
y_max_imputed = df_max_imputed.iloc[:, -1]

X_mode_imputed = df_mode_imputed.iloc[:, :-1]
y_mode_imputed = df_mode_imputed.iloc[:, -1]
# Train-test split for each dataset
X_train_original, X_test_original, y_train_original, y_test_original = train_test_split(
    X_original, y_original, test_size=0.2, random_state=42
)
X_train_min, X_test_min, y_train_min, y_test_min = train_test_split(
    X_min_imputed, y_min_imputed, test_size=0.2, random_state=42
)
X_train_max, X_test_max, y_train_max, y_test_max = train_test_split(
    X_max_imputed, y_max_imputed, test_size=0.2, random_state=42
)
X_train_mode, X_test_mode, y_train_mode, y_test_mode = train_test_split(
    X_mode_imputed, y_mode_imputed, test_size=0.2, random_state=42
)
# Print confirmation
print("Datasets have been split into training and testing sets successfully!")
```

PRINTING HEADS OF TRAIN AND TEST SETS

```
# Original Dataset
print("Original Dataset:")
print("X_train_original:\n", X_train_original.head())
print("y_train_original:\n", y_train_original.head())
print("X_test_original:\n", X_test_original.head())
print("y_test_original:\n", y_test_original.head())
```

```
# Min Imputed Dataset
print("Min Imputed Dataset:")
print("X_train_min:\n", X_train_min.head())
print("y_train_min:\n", y_train_min.head())
print("X_test_min:\n", X_test_min.head())
print("y_test_min:\n", y_test_min.head())
```

```
# Max Imputed Dataset
print("Max Imputed Dataset:")
print("X_train_max:\n", X_train_max.head())
print("y_train_max:\n", y_train_max.head())
print("X_test_max:\n", X_test_max.head())
print("y_test_max:\n", y_test_max.head())
```

```
# Mode Imputed Dataset
print("Mode Imputed Dataset:")
print("X_train_mode:\n", X_train_mode.head())
print("y_train_mode:\n", y_train_mode.head())
print("X_test_mode:\n", X_test_mode.head())
print("y_test_mode:\n", y_test_mode.head())
```

Validating the presence of NaN values and checking data types

```
def validate_data(X, y, dataset_name):
    print(f"\n### Validation for {dataset_name} ###")

    # Check for NaN values in features and target
    print(f"Number of NaN values in X ({dataset_name}): {X.isnull().sum().sum()}")
    print(f"Number of NaN values in y ({dataset_name}): {y.isnull().sum()}")

    # Check data types
    print(f"Data types of X ({dataset_name}):\n{X.dtypes}")
    print(f"Data type of y ({dataset_name}): {y.dtype}")

    # Check for unique values in the target (optional sanity check)
    print(f"Unique values in y ({dataset_name}): {y.unique()}")
    print(f"Shape of X: {X.shape}, Shape of y: {y.shape}")

# Validate Original Dataset
validate_data(X_original, y_original, "Original")

# Validate Min-Imputed Dataset
validate_data(X_min_imputed, y_min_imputed, "Min Imputed")

# Validate Max-Imputed Dataset
validate_data(X_max_imputed, y_max_imputed, "Max Imputed")

# Validate Mode-Imputed Dataset
validate_data(X_mode_imputed, y_mode_imputed, "Mode Imputed")
```

Performing classification and evaluation on MIN imputed dataset

SVM Classification on min imputed dataset

```
from sklearn.svm import SVC
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
    classification_report,
)
# Function to train and evaluate SVM on a dataset
def svm_classification(X_train, X_test, y_train, y_test, dataset_name):
    print(f"\n### SVM Classification for {dataset_name} ###")
    # Train the SVM classifier
    svm = SVC()
    svm.fit(X_train, y_train)
    # Predict the test set
    y_pred = svm.predict(X_test)
    # Metrics calculation
    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')
    conf_matrix = confusion_matrix(y_test, y_pred)
    class_report = classification_report(y_test, y_pred)
    # Print metrics
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"Confusion Matrix:\n{conf_matrix}")
    print(f"Classification Report:\n{class_report}")
# Perform SVM classification and evaluation on the minimum imputed dataset
svm_classification(X_train_min, X_test_min, y_train_min, y_test_min, "Minimum Imputed Dataset")
```

KNN

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
    classification_report,
)
# Function to train and evaluate KNN on a dataset
def knn_classification(X_train, X_test, y_train, y_test, dataset_name, n_neighbors=5):
    print(f"\n### KNN Classification for {dataset_name} ###")
     # Train the KNN classifier
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X_train, y_train)
    # Predict the test set
    y_pred = knn.predict(X_test)
    # Metrics calculation
    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')
    conf_matrix = confusion_matrix(y_test, y_pred)
    class_report = classification_report(y_test, y_pred)
    # Print metrics
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"Confusion Matrix:\n{conf_matrix}")
    print(f"Classification Report:\n{class_report}")
# Perform KNN classification and evaluation
knn_classification(X_train_min, X_test_min, y_train_min, y_test_min, "Minimum Imputed Dataset")
```

Decision Tree classifier

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
    classification_report,
)
# Function to train and evaluate Decision Tree on a dataset
def decision_tree_classification(X_train, X_test, y_train, y_test, dataset_name, max_depth=None):
    print(f"\n### Decision Tree Classification for {dataset_name} ###")

    # Train the Decision Tree classifier
    dt = DecisionTreeClassifier()
    dt.fit(X_train, y_train)

    # Predict the test set
    y_pred = dt.predict(X_test)
    # Metrics calculation
    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')
    conf_matrix = confusion_matrix(y_test, y_pred)
    class_report = classification_report(y_test, y_pred)
    # Print metrics
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"Confusion Matrix:\n{conf_matrix}")
    print(f"Classification Report:\n{class_report}")
# Perform Decision Tree classification and evaluation
decision_tree_classification(X_train_min, X_test_min, y_train_min, y_test_min, "Minimum Imputed Dataset")
```

Naive Bayes

```python
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
    classification_report,
)

# Function to train and evaluate Naive Bayes on a dataset
def naive_bayes_classification(X_train, X_test, y_train, y_test, dataset_name):
    print(f"\n### Naive Bayes Classification for {dataset_name} ###")

    # Train the Naive Bayes classifier
    nb = GaussianNB()
    nb.fit(X_train, y_train)
    # Predict the test set
    y_pred = nb.predict(X_test)
    # Metrics calculation
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted', zero_division=0)
    recall = recall_score(y_test, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y_test, y_pred, average='weighted', zero_division=0)
    conf_matrix = confusion_matrix(y_test, y_pred)
    class_report = classification_report(y_test, y_pred, zero_division=0)
    # Print metrics
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"Confusion Matrix:\n{conf_matrix}")
    print(f"Classification Report:\n{class_report}")
# Perform Naive Bayes classification on all datasets
naive_bayes_classification(X_train_min, X_test_min, y_train_min, y_test_min, "Minimum Imputed Dataset")
```

Neural Network

```
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
    classification_report,
)
# Function to train and evaluate Neural Network on a dataset
def neural_network_classification(X_train, X_test, y_train, y_test, dataset_name):
    print(f"\n### Neural Network Classification for {dataset_name} ###")

    # Train the Neural Network classifier
    nn = MLPClassifier(hidden_layer_sizes=(100,), max_iter=300)
    nn.fit(X_train, y_train)

    # Predict the test set
    y_pred = nn.predict(X_test)
    # Metrics calculation
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted', zero_division=0)
    recall = recall_score(y_test, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y_test, y_pred, average='weighted', zero_division=0)
    conf_matrix = confusion_matrix(y_test, y_pred)
    class_report = classification_report(y_test, y_pred, zero_division=0)
    # Print metrics
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"Confusion Matrix:\n{conf_matrix}")
    print(f"Classification Report:\n{class_report}")
# Perform Neural Network classification on all datasets
neural_network_classification(X_train_min, X_test_min, y_train_min, y_test_min, "Minimum Imputed Dataset")
```

Random Forest

```python
from sklearn.ensemble import RandomForestClassifier

# Function to train and evaluate Random Forest on a dataset
def random_forest_classification(X_train, X_test, y_train, y_test, dataset_name, n_estimators=100):
    print(f"\n### Random Forest Classification for {dataset_name} ###")

    # Train the Random Forest classifier
    rf = RandomForestClassifier(n_estimators=n_estimators, random_state=70)
    rf.fit(X_train, y_train)

    # Predict the test set
    y_pred = rf.predict(X_test)

    # Metrics calculation
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted', zero_division=0)
    recall = recall_score(y_test, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y_test, y_pred, average='weighted', zero_division=0)
    conf_matrix = confusion_matrix(y_test, y_pred)
    class_report = classification_report(y_test, y_pred, zero_division=0)

    # Print metrics
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"Confusion Matrix:\n{conf_matrix}")
    print(f"Classification Report:\n{class_report}")

# Perform Random Forest classification on all datasets
random_forest_classification(X_train_min, X_test_min, y_train_min, y_test_min, "Minimum Imputed Dataset")
```

Visualization of Metrics compared on all classifiers

```
# Metrics storage for the minimum imputed dataset
metrics_min = {
    "Classifier": ["SVM", "KNN", "Decision Tree", "Random Forest", "Neural Network", "Naive Bayes"],
    "Accuracy": [],
    "Precision": [],
    "Recall": [],
    "F1 Score": [],
}

# Function to append metrics with rounding to 3 decimal places
def append_metrics_with_precision(classifier_name, accuracy, precision, recall, f1_score):
    metrics_min["Accuracy"].append(round(accuracy, 3))
    metrics_min["Precision"].append(round(precision, 3))
    metrics_min["Recall"].append(round(recall, 3))
    metrics_min["F1 Score"].append(round(f1_score, 3))

# Function to generate a random seed dynamically based on time or other factors
def generate_random_seed():
    return np.random.randint(0, 10000)  # Generating a random seed within a range

# Collect metrics for each classifier
# SVM
svm = SVC(random_state=generate_random_seed())
svm.fit(X_train_min, y_train_min)
y_pred_svm = svm.predict(X_test_min)
append_metrics_with_precision(
    "SVM",
    accuracy_score(y_test_min, y_pred_svm),
    precision_score(y_test_min, y_pred_svm, average="weighted"),
    recall_score(y_test_min, y_pred_svm, average="weighted"),
    f1_score(y_test_min, y_pred_svm, average="weighted"),
)
```

```
# KNN
knn = KNeighborsClassifier()
knn.fit(X_train_min, y_train_min)
y_pred_knn = knn.predict(X_test_min)
append_metrics_with_precision(
    "KNN",
    accuracy_score(y_test_min, y_pred_knn),
    precision_score(y_test_min, y_pred_knn, average="weighted"),
    recall_score(y_test_min, y_pred_knn, average="weighted"),
    f1_score(y_test_min, y_pred_knn, average="weighted"),
)

# Decision Tree
dt = DecisionTreeClassifier(random_state=generate_random_seed())
dt.fit(X_train_min, y_train_min)
y_pred_dt = dt.predict(X_test_min)
append_metrics_with_precision(
    "Decision Tree",
    accuracy_score(y_test_min, y_pred_dt),
    precision_score(y_test_min, y_pred_dt, average="weighted"),
    recall_score(y_test_min, y_pred_dt, average="weighted"),
    f1_score(y_test_min, y_pred_dt, average="weighted"),
)

# Random Forest
rf = RandomForestClassifier(random_state=generate_random_seed())
rf.fit(X_train_min, y_train_min)
y_pred_rf = rf.predict(X_test_min)
append_metrics_with_precision(
    "Random Forest",
    accuracy_score(y_test_min, y_pred_rf),
    precision_score(y_test_min, y_pred_rf, average="weighted"),
    recall_score(y_test_min, y_pred_rf, average="weighted"),
    f1_score(y_test_min, y_pred_rf, average="weighted"),
)
```

```python
# Neural Network
nn = MLPClassifier(random_state=generate_random_seed(), max_iter=500)
nn.fit(X_train_min, y_train_min)
y_pred_nn = nn.predict(X_test_min)
append_metrics_with_precision(
    "Neural Network",
    accuracy_score(y_test_min, y_pred_nn),
    precision_score(y_test_min, y_pred_nn, average="weighted"),
    recall_score(y_test_min, y_pred_nn, average="weighted"),
    f1_score(y_test_min, y_pred_nn, average="weighted"),
)

# Naive Bayes
nb = GaussianNB()
nb.fit(X_train_min, y_train_min)
y_pred_nb = nb.predict(X_test_min)
append_metrics_with_precision(
    "Naive Bayes",
    accuracy_score(y_test_min, y_pred_nb),
    precision_score(y_test_min, y_pred_nb, average="weighted"),
    recall_score(y_test_min, y_pred_nb, average="weighted"),
    f1_score(y_test_min, y_pred_nb, average="weighted"),
)

# Convert metrics to a pandas DataFrame for better table printing
df_metrics_min = pd.DataFrame(metrics_min)

# Print the metrics table
print("Classifier Metrics (Minimum Imputed Dataset):")
print(df_metrics_min)
```

```python
# Visualization of Metrics
def plot_metrics(metrics_dict, metric_name):
    classifiers = metrics_dict["Classifier"]
    values = metrics_dict[metric_name]

    plt.figure(figsize=(10, 6))
    plt.bar(classifiers, values, color="yellow")
    plt.title(f"Comparison of {metric_name} for Classifiers (Minimum Imputed Dataset)\n", fontsize=14)
    plt.ylabel(metric_name, fontsize=12)
    plt.xlabel("Classifiers", fontsize=12)
    plt.ylim(0, 1)
    for i, v in enumerate(values):
        plt.text(i, v + 0.01, f"{v:.2f}", ha="center", fontsize=10)
    plt.show()

# Plot each metric
plot_metrics(metrics_min, "Accuracy")
plot_metrics(metrics_min, "Precision")
plot_metrics(metrics_min, "Recall")
plot_metrics(metrics_min, "F1 Score")
```