AIA Mini Project Documentation

C. Hasya Reddy

BU22CSEN0101505

Contents

Problem Definition:

Diagnostically predict whether a patient has diabetes on the basis of a given Dataset.

Data Source Details:

Pima Indians Diabetes Dataset

- Dataset originally from the National Institute of Diabetes and Digestive and Kidney Diseases.
- The objective of the dataset is to diagnostically predict whether a patient has diabetes.
- Link to download dataset: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

Features of Pima Indians Diabetes Dataset:

- The datasets consist of several medical predictor (independent) variables and one target (dependent) variable, Outcome.
- All patients here are females at least 21 years old
- Total records: 768
- Independent variables include:
  - ➢ Pregnancies: Number of times pregnant
  - ➢ Glucose: Glucose concentration
  - ➢ BloodPressure: Diastolic blood pressure
  - ➢ Skin Thickness: Triceps skin fold thickness
  - ➢ Insulin: 2-Hour serum insulin
  - ➢ BMI: Body mass index
  - ➢ Diabetes Pedigree Function: Diabetes pedigree function
  - ➢ Age
- Dependent Variable
  - ➢ Outcome: Class variable (0 or 1) 268 of 768 are 1, the others are 0

Steps:

1)Importing dataset

Overview

In this step, the dataset is imported into the program using the `pandas` library. The dataset is stored in a CSV (Comma-Separated Values) file named `diabetes.csv`. The aim of this step is to load the data into a pandas DataFrame for easy access and manipulation.

Code Explanation

```
import pandas as pd
```

import numpy as np

df = pd.read_csv("diabetes.csv")

import pandas as pd:

Imports the `pandas` library, which provides data manipulation tools, especially for structured data like tables and time series.

import numpy as np:

Imports the `numpy` library, which is used for numerical computing, including array operations.

pd.read_csv("diabetes.csv"):

This function reads the CSV file named `diabetes.csv` and loads its content into a pandas DataFrame. The CSV format is common for storing tabular data and contains rows and columns separated by commas.

df :

- This is the DataFrame object where the dataset is stored.

- The DataFrame allows for easy data manipulation, querying, and preprocessing.

Purpose of Importing the Dataset

- Load the data into a format (DataFrame) that allows for efficient manipulation, exploration, and analysis.

- Perform various operations like data preprocessing, visualization, and model training.

- Access the data through methods provided by pandas, such as filtering, grouping, and descriptive statistics.

2) Describing dataset

Overview

Once the dataset is imported into a DataFrame (df), the next step is to understand its structure and basic statistics. This involves getting an overview of the dataset's dimensions, data types, and summary statistics.

Code Explanation

print(df.info())

print(df.describe())

df.info():

This method provides a concise summary of the DataFrame. It includes:

- Number of rows and columns.

- Column names and their data types.

- Number of non-null values in each column.

df.describe():

This method provides a summary of the statistics for each numerical column, including:

- Count: Number of non-null entries.

- Mean: Average value of each feature.

- Standard Deviation (std): A measure of spread or variability.

- Min/Max: Minimum and maximum values.

- 25%, 50%, 75% (percentiles): Quartiles that give a sense of the distribution.

3)Visualization of the Dataset

Overview

Visualization is a key step in understanding the distribution, relationships, and patterns within the data. By plotting graphs and charts, we can gain insights that may not be obvious from raw data or statistical summaries.

Library Imports for Visualization

```
import seaborn as sns

import matplotlib.pyplot as plt
```

matplotlib.pyplot:

A widely used library for creating static, interactive, and animated visualizations in Python. It is often used for basic charts such as line plots, bar charts, histograms, etc.

seaborn:

Built on top of matplotlib, Seaborn provides a high-level interface for drawing attractive and informative statistical graphics. It simplifies complex visualizations like heatmaps, pair plots, and distribution plots.

Visualization techniques

1.Histogram

```
sns.histplot([df["Glucose"],df["BMI"],df["Age"]],bins=50,kde=True,alpha=0.5,palette=["gree
n","blue","orange"])
```

sns.histplot():

This function from Seaborn is used to plot histograms of one or more variables. In this case, you are visualizing the distributions of Glucose, BMI, and Age in a single plot.

[df["Glucose"], df["BMI"], df["Age"]]:

The three numerical columns (Glucose, BMI, and Age) from the dataset are provided as inputs to create the histogram. This allows you to compare the distribution of these three variables in one plot.

bins=50:

This specifies the number of bins to divide the data into. A higher number of bins provides more granularity but may make the plot harder to interpret.

kde=True:

This adds a Kernel Density Estimate (KDE) curve to the histogram, which provides a smooth estimate of the data's distribution.

alpha=0.5:

This sets the transparency of the bars, making overlapping areas easier to see. A value of 0.5 means 50% transparency.

palette=["green", "blue", "orange"]:

This specifies the colors used for each histogram. Here, green will be for Glucose, blue for BMI, and orange for Age.

2. Heat map

k = df.corr()

sns.heatmap(k, annot=True, cmap="bwr")

df.corr():

- This function computes the correlation matrix for the DataFrame. The correlation matrix contains the pairwise correlation coefficients between the variables (features) in the dataset.

- Correlation values range from -1 (perfect negative correlation) to +1 (perfect positive correlation), with 0 indicating no correlation.

sns.heatmap():

This function is used to plot the heatmap based on the correlation matrix k.

annot=True:

This ensures that the correlation coefficients are annotated (displayed as text) on each cell of the heatmap.

cmap="bwr":

Specifies the color map used for the heatmap. The "bwr" color map stands for blue-white-red, which shows negative correlations in blue, no correlation in white, and positive correlations in red.

3. Pair plot

Pair Plot Represents:

Scatter Plots:

- Each scatter plot in the matrix represents the relationship between two numerical variables. For example, it may show the relationship between Glucose and BMI, Age and BloodPressure, etc.

- The different colors indicate how these relationships vary based on the outcome (whether the person has diabetes or not).

Diagonal Plots:

The diagonal of the pair plot shows the distribution of individual features, often in the form of a histogram or kernel density estimate (KDE).

```
sns.pairplot(df, hue="Outcome")
```

sns.pairplot():

- This Seaborn function creates a matrix of scatter plots (and optionally, histograms) for each pair of numerical variables in the dataset.

- It’s a very useful tool for exploring relationships between multiple variables simultaneously.

df:

Refers to the DataFrame df that contains your dataset, in this case, the diabetes dataset.

hue="Outcome":

- The hue parameter is used to color the plots according to a categorical variable, which in this case is Outcome (whether the patient has diabetes or not).

- Points corresponding to different values of Outcome will be plotted in different colors (e.g., blue for Outcome=0 and orange for Outcome=1), allowing you to visually distinguish the distribution of features based on the target variable.

Purpose of Visualization:

Explore Relationships: Visualizations like heatmaps and pair plots help uncover relationships between features and the target variable.

Detect Outliers: Box plots are useful for identifying outliers or extreme values in the dataset that may need attention during preprocessing.

Understand Feature Distributions: Histograms and distribution plots allow you to check if features are normally distributed or skewed, which affects model performance and choices for feature scaling.

Class Distribution: A count plot shows whether the dataset is balanced or imbalanced with respect to the target variable. If the classes are imbalanced, this could impact model performance and require resampling techniques.

4) Data Processing

Data preprocessing is a critical step in preparing the raw data for modelling. This step involves cleaning the data, handling missing values, transforming variables, and preparing it in a format suitable for machine learning algorithms.

1.Handling class Imbalance

If the target variable (Outcome) is imbalanced (i.e., more cases of one class than the other), we can use techniques like oversampling or undersampling to balance it.

Code

```
from imblearn.over_sampling import RandomOverSampler

ros = RandomOverSampler (random_state=42)

X, Y = ros.fit_resample(X, Y)

print("Resampled X:\n", X)

print("Resampled Y:\n", Y)
```

Imbalanced Data:

In many machine learning tasks, especially classification problems, you might face imbalanced datasets, where one class (such as Outcome=0, representing no diabetes) is more frequent than the other class (such as Outcome=1, representing diabetes).

If the dataset is imbalanced, models might become biased, predicting the majority class more often, and performing poorly on the minority class (e.g., diabetic patients).

RandomOverSampler:

The RandomOverSampler is a technique from the imbalanced-learn (imblearn) library that addresses this issue by randomly duplicating examples from the minority class to create a balanced dataset.

This method helps improve the performance of the classifier by ensuring that the minority class has enough representation during training.

ros = RandomOverSampler(random_state=42):

Creates an instance of the RandomOverSampler. The parameter random_state=42 ensures reproducibility, meaning the results will be the same every time you run the code.

X, Y = ros.fit_resample(X, Y):

fit_resample() method applies the over-sampling strategy to both the feature set X and the target Y.

It duplicates random samples from the minority class to create a new dataset where both classes are equally represented.

Balanced Dataset:

After resampling, the feature matrix X and the target Y will have an equal number of examples for both classes (i.e., Outcome=0 and Outcome=1).

This ensures that the classifier doesn't become biased toward the majority class and treats both classes equally during training.

Printing Resampled Data:

The print() statements show the resampled X and Y to verify that the data has been balanced after applying the RandomOverSampler.

2.Data Scaling

Data scaling is an essential step in preprocessing, particularly when machine learning algorithms are sensitive to the magnitude of features. Features like Glucose, BMI, and Age can have different scales, which may cause issues in certain algorithms. Scaling ensures that all features contribute equally to the model's predictions.

Code

```
from sklearn.preprocessing import StandardScaler

data_scaler = StandardScaler()

data_rescaled = data_scaler.fit_transform(X)

X = pd.DataFrame(data_rescaled, columns=X.columns)

print(X)
```

StandardScaler:

The StandardScaler from sklearn.preprocessing is used to standardize the dataset. It transforms the data in such a way that each feature will have a mean of 0 and a standard deviation of 1.

data_scaler.fit_transform(X):

fit_transform(X) does two things:

Fit: The StandardScaler calculates the mean and standard deviation of each feature in the dataset X.

Transform: The StandardScaler then transforms the data using the calculated mean and standard deviation, rescaling each feature so that its mean is 0 and standard deviation is 1.

X = pd.DataFrame(data_rescaled, columns=X.columns):

The rescaled data is converted back into a Pandas DataFrame to retain the original structure, with the same column names.

This ensures that the data can be used easily in subsequent model-building steps.

Output:

The transformed X is printed, showing the scaled feature values.

After scaling, each feature will have values centered around 0 with a standard deviation of 1.

5) Model Training

Model training is the process of feeding a machine learning algorithm with data so that it can learn the relationships between input features (independent variables) and target labels (dependent variables). The model uses the training data to adjust its parameters and make predictions.

1. Splitting the Data

Before training the model, we need to split the dataset into training and testing sets. This ensures that the model can be trained on one subset of data and tested on another to evaluate its performance on unseen data.

Code

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test =train_test_split(X,Y, test_size=0.2, random_state=42)

dia_en=pd.DataFrame()

res=pd.DataFrame()

dia_en["Outcome"]= Y_test
```

train_test_split(): This function splits the data into two subsets: the training set (80% of the data) and the testing set (20% of the data). The training set will be used to fit the model, and the testing set will be used to evaluate the model's performance.

The random_state=42 ensures reproducibility.

2.Training the data

Trained seven different machine learning models to predict the Outcome (diabetes or not) using the preprocessed dataset. Each model has its own strengths and weaknesses, and by training multiple models, we can compare their performances and select the best one for the given problem.

Model Training Steps for Each Algorithm:

Logistic Regression: Logistic regression is a linear model that estimates the probability that an instance belongs to a particular class (0 or 1). It is especially useful for binary classification problems.

Code

```
from sklearn.linear_model import LogisticRegression
```

```
model = LogisticRegression(random_state=42)
model.fit(X_train, y_train)
```

Decision Tree Classifier: Decision Tree is a tree-based algorithm that splits the data based on feature values to arrive at a decision. It is easy to interpret and performs well with complex datasets.

Code

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)
```

Random Forest Classifier: Random Forest is an ensemble learning method that combines multiple decision trees to improve accuracy and robustness. It reduces the risk of overfitting.

Code

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
```

K-Nearest Neighbors (KNN): KNN is a distance-based algorithm that classifies a new data point based on the majority class of its k-nearest neighbors. It's simple but effective for many tasks.

Code

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
```

Naive Bayes Classifier: Naive Bayes is a probabilistic classifier based on Bayes' theorem. It assumes that the features are conditionally independent given the class label. It is very fast and often used for text classification.

Code

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X_train, y_train)
```

Gradient Boosting Classifier: Gradient Boosting is a powerful boosting technique that builds trees sequentially. Each new tree attempts to correct the errors made by the previous trees.

Code

```
from sklearn.ensemble import GradientBoostingClassifier
```

```
model = GradientBoostingClassifier(random_state=42)

model.fit(X_train, y_train)
```

AdaBoost Classifier: AdaBoost is another boosting algorithm that adjusts the weights of incorrectly classified instances and focuses more on them in subsequent trees. It combines weak learners to form a strong classifier.

Code

```
from sklearn.ensemble import AdaBoostClassifier

model = AdaBoostClassifier(random_state=42)

model.fit(X_train, y_train)
```

6) Model Creation

- Model creation in machine learning refers to the process of selecting and constructing a mathematical model that can make predictions based on input data.
- The process involves choosing the appropriate machine learning algorithm, training the model with historical data, and adjusting parameters to optimize performance.
- The end goal is to develop a model that can generalize well on unseen data to provide accurate predictions.

Different algorithms can be used for model creation, and the choice depends on the nature of the data and the problem being solved.

1. Linear Regression

Definition: Linear regression is a supervised learning algorithm used for predicting a continuous target variable based on one or more input features. It assumes a linear relationship between the input variables and the target variable.

Key Points:

- Finds the line of best fit for predicting continuous values.

- Optimized using a loss function, such as Mean Squared Error (MSE).

2. Decision Tree

Definition: A decision tree is a supervised learning algorithm used for both classification and regression tasks. It works by splitting the data into subsets based on the value of input features. Each internal node represents a decision based on a feature, and each leaf node represents an outcome.

Key Points:

- The tree is built by recursively splitting the data at each node.

- The goal is to create the purest possible subsets where each leaf node has a homogenous label.

3. Random Forest

Definition: Random forest is an ensemble learning method that combines multiple decision trees to improve predictive accuracy and control overfitting. It builds many decision trees (each trained on a random subset of data) and averages their predictions for regression tasks or takes the majority vote for classification tasks.

Key Points:

- Combines the power of multiple decision trees to create a more robust model.

- Reduces variance and prevents overfitting.

4. K-Nearest Neighbors (KNN)

Definition: KNN is a supervised learning algorithm used for classification and regression tasks. It works by finding the 'K' most similar instances (neighbors) in the training data and making predictions based on the majority class (classification) or average value (regression) of those neighbors.

Key Points:

- KNN is a non-parametric algorithm, meaning it doesn't make assumptions about the underlying data distribution.

- It is computationally expensive during prediction as it requires searching the entire dataset to find the nearest neighbors.

5. Naive Bayes

Definition: Naive Bayes is a family of supervised learning algorithms based on applying Bayes' theorem with the assumption of conditional independence between the features. Despite its simplistic assumptions, it works well for many applications.

Key Points:

-Works based on Bayes' theorem:

-Works well when features are independent but is effective even when the independence assumption is violated in practice.

6. Gradient Boosting

Definition: Gradient Boosting is an ensemble learning technique where weak learners (typically decision trees) are sequentially trained, with each new tree attempting to correct the errors made by the previous ones. The model minimizes a loss function by taking the gradient of the error and boosting the learning process.

Key Points:

- Sequentially builds trees, where each tree corrects the residuals from the previous one.

- Can be prone to overfitting, but often produces high-performing models.

7. AdaBoost (Adaptive Boosting)

Definition: AdaBoost is a boosting ensemble technique that combines multiple weak classifiers to create a strong classifier. It works by assigning more weight to instances that were incorrectly classified by previous classifiers and updating the model accordingly.

Key Points:

- Focuses more on misclassified examples to improve accuracy.

- Weak learners are typically decision stumps (one-level decision trees).

- Each classifier’s contribution is weighted by its performance.

7) Model Evaluation

- Model evaluation is the process of assessing the performance and effectiveness of a predictive model using various metrics and techniques.
- It helps determine how well the model generalizes to unseen data and how accurately it can make predictions.
- The evaluation phase is crucial in the machine learning workflow, as it provides insights into the model's strengths and weaknesses, guiding further improvements and refinements.

Key Aspects of Model Evaluation:

Performance Metrics: These metrics quantify the model's performance. Common metrics include:

1.Accuracy: The proportion of correctly predicted instances among the total instances.

2.Precision: The ratio of true positive predictions to the total predicted positives, indicating the correctness of positive predictions.

3.Recall (Sensitivity): The ratio of true positive predictions to the total actual positives, measuring the model's ability to identify relevant instances.

4.F1 Score: The harmonic mean of precision and recall, providing a balance between the two metrics.

5.ROC-AUC: The area under the Receiver Operating Characteristic curve, representing the trade-off between true positive rate and false positive rate.

Conclusion

Without Preprocessing: Naive Bayes is the best model due to its strong performance across all metrics.

With Preprocessing: Naive Bayes is still the best-performing model, maintaining the same high performance.

Naive Bayes is the best method in dataset, offering the best balance between accuracy, precision, recall, F1-score, and AUC, both with and without preprocessing.