COLLAGE OF COMPUTING

DEPARTMENT OF SOFTWAR ENGINEERING

NAME:ASREGIDE FIRDIE

COURSE TITLE :MACHINE LEARNING

ID:1400928

SUBMITTED TO: DERBEW FELASMAN

SUBMISSION DATE:6/2/201

Diabetes Prediction Machine Learning Project - Documentation

1. Problem Definition

Diabetes is a chronic metabolic disease characterized by high blood sugar levels over a prolonged period. If left untreated, it can lead to serious health complications, including heart disease, kidney failure, and vision loss. Early detection and timely intervention are essential for effective management and prevention of complications.

This project aims to develop a machine learning-based diabetes prediction system that can help identify individuals at high risk of developing diabetes based on various medical and physiological features. By analyzing key health indicators such as glucose levels, blood pressure, insulin levels, BMI, age, and genetic predisposition, the model will predict whether a person is likely to have diabetes.

The predictive model will be trained on the Pima Indians Diabetes Dataset, a well-known dataset in the medical and machine learning community. The final model will be deployed as a FastAPI-based web service, allowing users to input relevant health parameters and receive a real-time diabetes prediction.

This project provides an opportunity to explore the full machine learning pipeline, from data acquisition and preprocessing to model training, evaluation, and deployment, offering practical insights into applying AI in healthcare.

2. Data Source and Description

- **Dataset Used:** ASRE Diabetes Dataset
- **Source:https://github.com/asre459/Diabete-predict-machine-learniing-project/blob/main/diabetes.csv**
-
- **License:** Publicl available for research and educational use.
- **Structure:** The dataset contains 768 observations with 8 medical predictor variables and 1 target variable (`Outcome`).
- **Features:**
    - Pregnancies: Number of times pregnant
    - Glucose: Plasma glucose concentration
    - Blood Pressure: Diastolic blood pressure (mm Hg)
    - Skin Thickness: Triceps skinfold thickness (mm)
    - Insulin: 2-Hour serum insulin (mu U/ml)
    - BMI: Body mass index (weight in kg/(height in m)^2)
    - Diabetes Pedigree Function: Genetic influence on diabetes
    - Age: Age in years

- Outcome: Target variable (0 = No diabetes, 1 = Diabetes)

3. Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is a crucial step in understanding the dataset and preparing it for modeling. It helps in identifying patterns, inconsistencies, outliers, and relationships between variables. Here's a detailed explanation of the steps followed during the EDA:

1. **Basic Statistics:**
   To understand the data distribution and central tendencies, we used the describe() method in pandas, which provides a summary of the dataset's numerical features.

```
1. Basic Statistics:
To understand the data distribution and central tendencies, we used the describe()
method in pandas, which provides a summary of the dataset's numerical features.
```

Output:

Count: The number of non-null entries in each column.

Mean: The average value of each column.

Standard Deviation: The spread of the data from the mean.

Min and Max: The range of values in each column.

25%, 50%, 75% Percentiles: These represent the distribution of the data, with the 50% representing the median.

Key Insights:

The Glucose values range from 0 to 199 mg/dL, with a mean of 121 mg/dL, indicating a wide range of glucose levels in the dataset.

The BMI values have a wide spread, with values ranging from 0 to 67, and a mean of around 31, which could indicate overweight individuals.

The Age column has a range from 21 to 81 years, with an average of 33.2 years.

2. Data Quality Issues:

EDA is also an opportunity to identify data quality issues that might impact the model's performance. Here are the key data quality issues discovered during this phase:

- **Missing Values:**
  - The dataset did not have explicit missing values, but certain features contained zero values that should be treated as missing data. Specifically:
    - **Glucose**, **Blood Pressure**, **Skin Thickness**, and **Insulin** have zero values that do not represent valid measurements. These zero values likely correspond to missing or unrecorded data.
- **Handling Zero Values:**
  - To address these issues, we replaced the zero values with `NaN` (Not a Number) to indicate missing values, allowing for proper handling later in the process (such as imputing missing data).
  - For numerical features, we used the median to fill the missing values, as it is a robust statistic that is less sensitive to outliers.

```
df[columns_with_zeros] = df[columns_with_zeros].replace(0, np.nan)
df.fillna(df.median(numeric_only=True), inplace=True)

```

3. Visualizations:

Visualizations are a powerful tool to help identify patterns, trends, and anomalies in the dataset. During EDA, several types of visualizations were used:

a. Histograms for Feature Distributions:

Histograms help visualize the distribution of each feature and assess whether the data is normally distributed or skewed.

Purpose: To observe the distribution and spread of individual features.

Findings:

Features like Glucose and BMI are right-skewed, meaning most values fall on the lower end, with fewer high values.

Pregnancies show a skewed distribution towards fewer pregnancies.

Age appears fairly normal, with most individuals falling between 21 and 60 years.

```
df.hist(figsize=(12, 10))
plt.show()
```

b. Boxplots for Detecting Outliers:

Boxplots are used to detect outliers in the data. They show the median, quartiles, and any data points that are far from the interquartile range (outliers).

Purpose: To detect any outliers in the features that could affect the performance of the model.

Findings:

Insulin and BMI features have some extreme values that are considered outliers.

Glucose and Blood Pressure show a more balanced distribution without extreme outliers.

Pregnancies does not show significant outliers.

```
plt.figure(figsize=(12, 8))
df.boxplot()
plt.show()
```

c. Correlation Heatmap to Understand Relationships Between Features:

A correlation heatmap provides a visual representation of the relationships between features. It helps identify which features are highly correlated and which may contain redundant information.

Purpose: To visualize the strength and direction of relationships between features.

Findings:

Glucose and Insulin show a moderate positive correlation (0.46), indicating that as glucose levels increase, insulin levels also tend to increase.

BMI and Insulin are positively correlated, suggesting that higher BMI may be associated with higher insulin levels.

Age and Blood Pressure show a slight correlation, as expected, since older individuals may have higher blood pressure.

```
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

d. Target Variable Distribution Plot:

The distribution of the target variable outcome (whether a person has diabetes or not) was also analyzed. This plot helps understand the class balance of the data.

Purpose: To visualize how the target variable is distributed and identify any class imbalance.

Findings:

The dataset has an imbalanced distribution of the target variable, with approximately 68% of patients being non-diabetic (0) and 32% being diabetic (1).

This imbalance might require attention during model training, such as using techniques like oversampling or adjusting class weights to handle the imbalanced classes.

```
sns.countplot(x='Outcome', data=df)
plt.title('Distribution of Diabetes Outcome')
plt.show()
```

4. Data Preprocessing

Data preprocessing is a critical step in machine learning, as it ensures that the dataset is clean, consistent, and ready for modeling. In this section, we describe the steps taken during data preprocessing to prepare the dataset for analysis and model building.

1. Standardizing Column Names:

In real-world datasets, column names can sometimes contain inconsistencies such as extra spaces, case differences, or special characters. These inconsistencies can lead to errors when trying to access or manipulate the data. To ensure consistency, we standardized the column names by:

Stripping extra spaces around the column names.

Converting all column names to lowercase to avoid case sensitivity issues.

This step ensures that column names are uniform and can be easily accessed without errors.

Reason for Standardizing Column Names: This step prevents errors when matching expected column names (e.g., for feature selection or data cleaning) and allows the dataset to be more predictable and less prone to mismatches.

```
df.columns = df.columns.str.strip().str.lower()
```

2. Handling Missing Values:

Missing data is a common issue in real-world datasets. Missing values can arise due to various reasons, such as incomplete data collection or errors during data entry. If not properly handled, missing values can distort the results of machine learning models. Below, we explain the steps taken to deal with missing values in this dataset:

a. Replacing Zero Values with NaN:

Certain features, such as Glucose, Blood Pressure, Skin Thickness, and Insulin, contain values of 0, which are not medically plausible. For example:

Glucose levels cannot be zero for a healthy individual.

Blood Pressure cannot be zero for a person, except in extreme conditions (e.g., during a medical emergency or in case of data errors).

Since these values are likely to represent missing or unrecorded data rather than valid measurements, we replaced zero values with NaN (Not a Number), which is a more appropriate way to indicate missing data.

```
columns_with_zeros = ['glucose', 'bloodpressure', 'skinthickness', 'insulin']
df[columns_with_zeros] = df[columns_with_zeros].replace(0, np.nan)
```

b. Imputing Missing Values:

Once the zero values were replaced with NaN, we needed to handle the missing values. Removing rows with missing data could lead to significant loss of information, especially when dealing with relatively small datasets. Thus, we chose median imputation as the strategy to fill the missing values.

Why Median?

The median is preferred over the mean when imputing missing values in the presence of skewed distributions or outliers. The median is less sensitive to extreme values, which makes it a robust statistic for imputation.

For example, if a feature like Insulin has extreme outliers, using the mean would distort the imputation, while the median provides a better estimate of the central tendency.

```
df.fillna(df.median(numeric_only=True), inplace=True)
```

Effect: After replacing the zero values with NaN and filling missing values with the median, the dataset is now free of missing data, and each feature is fully populated with plausible values.

3. Feature Scaling:

In machine learning, feature scaling is a technique used to standardize the range of independent variables or features. The goal is to normalize the data so that each feature contributes equally to the model. Feature scaling is particularly important when the model involves distance-based calculations (like in k-Nearest Neighbors, Support Vector Machines, or Logistic Regression) or gradient descent optimization.

a. Why Standardization (Z-Score Normalization)?

Standardization (also known as Z-score normalization) transforms the data such that each feature has a mean of 0 and a standard deviation of 1. This is especially useful when the features have different units and scales (e.g., Glucose in mg/dL, BMI in kg/m², and Age in years). Without scaling, the model may give more importance to features with larger ranges.

Standardization is performed using the StandardScaler from sklearn.preprocessing, which scales the features by subtracting the mean and dividing by the standard deviation.

b. Applying StandardScaler:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# Apply scaling to the features (X) but not the target (y)
X = df.drop('outcome', axis=1)  # Drop target column
X_scaled = scaler.fit_transform(X)
```

Explanation of the Transformation:

fit(): This step calculates the mean and standard deviation of each feature in the training data.

transform(): Using the calculated mean and standard deviation, the data is scaled to have zero mean and unit variance.

fit_transform(): A combination of both, used for transforming the data while fitting the scaler.

Effect of Scaling: After scaling, each feature in X_scaled has a mean of 0 and a standard deviation of 1. This ensures that no feature dominates others due to differences in scale, leading to better model performance, especially for distance-based algorithms and gradient-based optimization methods.

5. Model Selection and Training

After completing data preprocessing, the next step in building a machine learning pipeline is to select an appropriate model and train it. In this case, we chose Logistic Regression as our model for binary classification, since we are predicting whether a patient has diabetes (outcome variable: 1 for diabetes, 0 for no diabetes).

1. Chosen Model: Logistic Regression

Logistic Regression is a popular machine learning algorithm used for binary classification problems. It predicts the probability that an instance belongs to a particular class (e.g., positive outcome vs negative outcome). Here, we are trying to predict the likelihood of a person having diabetes based on their medical features.

Why Logistic Regression?

Interpretable Model:

One of the key reasons for selecting Logistic Regression is its interpretability. Unlike more complex models like deep neural networks, Logistic Regression provides clear insights into the relationships between features and the target variable.

The model coefficients indicate the strength and direction of the relationship between each feature and the likelihood of having diabetes. For example, a positive coefficient means that as the feature increases, the likelihood of diabetes increases.

Suitability for Binary Classification:

Logistic Regression is specifically designed for binary classification problems, making it a good fit for our dataset, where the target variable ("Outcome") is binary (0 or 1).

Efficiency for Small to Medium Datasets:

Logistic Regression performs well on small to medium-sized datasets. It does not require large computational resources or complex tuning, making it a solid choice for datasets like the diabetes dataset.

2. Training Process

a. Splitting the Dataset:

To train the model, we need to split the dataset into two subsets:

Training Set: This portion is used to fit the model (train the model's parameters).

Testing Set: This portion is used to evaluate the model's performance after training.

We split the data into 80% for training and 20% for testing. This ratio is common because it provides enough data to train the model while leaving a substantial portion to evaluate the model's generalization ability.

```
from sklearn.model_selection import train_test_split

# Split the data into features (X) and target (y)
X = df.drop('outcome', axis=1)
y = df['outcome']

# Split the data into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42)
```

X_train and X_test contain the features (inputs).

y_train and y_test contain the target variable (outcome).

b. Hyperparameter Tuning with GridSearchCV:

Once the data is split, we can move on to hyperparameter tuning. Logistic Regression, like most machine learning algorithms, has hyperparameters that can influence the model's performance. The goal is to find the optimal hyperparameters that result in the best model performance.

In this case, the most relevant hyperparameter for Logistic Regression is C, which controls the regularization strength. Regularization helps prevent overfitting by penalizing large coefficients. A larger value of C results in weaker regularization (less penalization), while a smaller value of C increases regularization (more penalization).

We used GridSearchCV to perform an exhaustive search over a specified parameter grid. GridSearchCV evaluates multiple combinations of hyperparameters using cross-validation, which provides a more reliable estimate of model performance.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Define the parameter grid for hyperparameter tuning
param_grid = {'C': [0.01, 0.1, 1, 10, 100]}

# Initialize GridSearchCV with Logistic Regression model and cross-validation
(cv=5)
grid_search = GridSearchCV(LogisticRegression(), param_grid, cv=5)

# Perform grid search on the training data
grid_search.fit(X_train, y_train)

# Output the best parameters and the best cross-validation score
print('Best Parameters:', grid_search.best_params_)
print('Best Score:', grid_search.best_score_)
```

C values: The parameter grid includes different values of C, ranging from 0.01 to 100, allowing the GridSearchCV to test various levels of regularization.

cv=5: We use 5-fold cross-validation, which means the training data is divided into 5 subsets, and the model is trained and validated 5 times, with each fold serving as the validation set once.

The best parameters and best score provide us with the most optimal values for hyperparameters that result in the best model performance on the training data.

c. Final Model Training:

After identifying the best hyperparameters through GridSearchCV, we retrain the Logistic Regression model using those optimal hyperparameters. This ensures that the model is trained on the most suitable configuration.

```
from sklearn.linear_model import LogisticRegression

# Initialize the model
model = LogisticRegression()

# Train the model
model.fit(X_train, y_train)
```

best_estimator_: GridSearchCV returns the model with the best hyperparameters found during the search.

This trained model is now ready to make predictions on new, unseen data.

3. Model Evaluation:

Once the model is trained, we evaluate its performance on the test data (which was not used during training) to ensure that the model generalizes well to new data.

We make predictions on the test data and calculate various evaluation metrics such as accuracy, precision, recall, F1-score, and AUC score. These metrics will help assess the performance of the model.

```
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report, roc_auc_score

# Evaluate the model
print('Accuracy:', accuracy_score(y_test, y_pred))
print('Confusion Matrix:\n', confusion_matrix(y_test, y_pred))
print('Classification Report:\n', classification_report(y_test, y_pred))
print('ROC AUC Score:', roc_auc_score(y_test, y_pred))
```

6. Model Evaluation

After training the model, it's important to evaluate its performance to understand how well it generalizes to unseen data. We use several performance metrics to assess the model's effectiveness in predicting diabetes. These metrics provide a detailed view of the model's strengths and weaknesses.

1. Performance Metrics

Here are the key performance metrics we used to evaluate the model:

a. Accuracy Score

Accuracy is the most straightforward metric, defined as the ratio of correctly predicted instances to the total number of instances. However, accuracy alone may not always be the best indicator of model performance, especially in imbalanced datasets (where one class is more prevalent than the other).

The formula for accuracy is:

Accuracy=True Positives+True Negatives /Total Instances

In the case of the diabetes prediction model, accuracy tells us the overall percentage of correct predictions (whether the model correctly classified a person as having diabetes or not)**.**

```
from sklearn.metrics import accuracy_score

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

b. Confusion Matrix

The Confusion Matrix provides a breakdown of the model's predictions into four categories:

True Positives (TP): Correctly predicted as having diabetes.

True Negatives (TN): Correctly predicted as not having diabetes.

False Positives (FP): Incorrectly predicted as having diabetes (Type I error).

False Negatives (FN): Incorrectly predicted as not having diabetes (Type II error).

The confusion matrix helps visualize how the model is making errors and can help identify which type of error (false positive or false negative) is more common.

```
from sklearn.metrics import confusion_matrix

# Calculate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print('Confusion Matrix:\n', conf_matrix)
```

c. Precision, Recall, and F1-score

Precision, recall, and F1-score are more informative metrics, especially when dealing with imbalanced classes (e.g., if more people in the dataset don't have diabetes than do). These metrics give us a better understanding of how the model performs on each class.

Precision tells us how many of the instances predicted as positive (i.e., diabetes) are actually positive.

Precision =TP/TP+TF

Recall (also known as Sensitivity or True Positive Rate) tells us how many of the actual positive instances (i.e., people who actually have diabetes) were correctly predicted by the model.

Recall =TP/TP+FN

F1-score is the harmonic mean of precision and recall. It gives a balanced view when you have an uneven class distribution and want to consider both false positives and false negatives.

F1-score= 2*Recall* Precision/Recall+ Precision

```
from sklearn.metrics import classification_report

# Get precision, recall, and F1-score
class_report = classification_report(y_test, y_pred)
print('Classification Report:\n', class_report)
```

d. ROC-AUC Score

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the model's ability to distinguish between classes. The AUC (Area Under the Curve) score quantifies the overall ability of the model to correctly classify positive and negative instances.

ROC Curve: The curve plots the True Positive Rate (Recall) against the False Positive Rate (FPR). A model that performs well will have a curve that hugs the top-left corner of the plot.

AUC Score: The AUC score ranges from 0 to 1, where a score of 1 represents perfect classification, and 0.5 represents a model that performs no better than random guessing.

```
from sklearn.metrics import roc_curve

# Plot ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
plt.plot(fpr, tpr, label='ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```

2. Baseline Comparison: Dummy Classifier

To better understand the performance of our model, it's useful to compare it to a baseline model. A simple baseline model, such as the Dummy Classifier, predicts the most frequent class or makes random guesses based on the distribution of the target variable. This provides a benchmark to see how much better our trained model is compared to a basic, untrained model.

For the baseline comparison:

Dummy Classifier Strategy: We used the "most frequent" strategy, which simply predicts the most common class (in this case, the majority class, "no diabetes").

```
from sklearn.dummy import DummyClassifier

# Initialize the dummy classifier
dummy_clf = DummyClassifier(strategy="most_frequent")

# Train the dummy classifier
dummy_clf.fit(X_train, y_train)

# Make predictions
y_pred_dummy = dummy_clf.predict(X_test)

# Evaluate the dummy classifier
print('Dummy Classifier Accuracy:', accuracy_score(y_test, y_pred_dummy))
```

By comparing the accuracy and other performance metrics of the trained model with those of the Dummy Classifier, we can determine whether the trained model provides significant improvements over random guessing.

3. Results Interpretation

After evaluating the model's performance, here are some key observations:

a. Model's Performance:

Good Balance Between Precision and Recall: The model achieved a solid balance between precision and recall, meaning it was able to correctly identify a significant portion of the people who had diabetes (high recall) while avoiding too many false positives (high precision).

ROC AUC Score: The AUC score (close to 1) suggests that the model can effectively distinguish between people with and without diabetes. A high AUC score indicates that the model is good at predicting both classes, minimizing false positives and false negatives.

b. Identifying Trade-offs in Prediction Errors:

Precision-Recall Trade-off: Depending on the application, there may be a trade-off between precision and recall. For instance:

If it's more important to avoid false positives (e.g., unnecessary treatments or tests), the model might be tuned to favor higher precision.

If it's more important to catch as many true positives as possible (even at the cost of false positives), recall may be prioritized.

Confusion Matrix Analysis: By examining the confusion matrix, we can identify where the model is making the most errors:

If the number of False Positives (predicting someone has diabetes when they don't) is high, the model may be too sensitive.

If the number of False Negatives (predicting someone doesn't have diabetes when they do) is high, the model may be underestimating the likelihood of diabetes.

7. Model Deployment

Once the model has been trained and evaluated, the next step is to deploy it so that it can be used for real-world predictions. In this project, we used FastAPI as the deployment framework to create an API that can accept patient data and return diabetes predictions. Below, I will explain the steps involved in the deployment process.

1. Deployment Framework: FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints. It is designed to be easy to use and provides automatic interactive documentation (using Swagger UI). FastAPI is an excellent choice for deploying machine learning models because it is lightweight, supports asynchronous operations, and integrates well with Python-based libraries.

2. Deployment Steps

a. Saving the Trained Model and Scaler

The first step is to save the trained machine learning model and the scaler used for feature scaling (StandardScaler). This is done using joblib, a library that allows us to serialize (save) Python objects, including machine learning models, and later reload them.

The model is saved as diabetes_model.pkl.

The scaler used to scale input features is saved as scaler.pkl.

```
import joblib

# Save the trained model and scaler
joblib.dump(model, "diabetes_model.pkl")
joblib.dump(scaler, "scaler.pkl")
```

By saving these files, we can reload them at any time during the deployment process, making it possible to use the same trained model and scaler for real-time predictions.

b. Creating the FastAPI API

With the model and scaler saved, we proceed to create an API using FastAPI. The API accepts input data (patient information) from a POST request and returns a diabetes prediction (0 for no diabetes, 1 for diabetes). The structure of the input is defined using a Pydantic model, which is used by FastAPI for data validation.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import numpy as np
import joblib
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Load model and scaler with error handling
try:
    model = joblib.load("diabetes_model.pkl")  # Ensure your model file is in the
same directory
    scaler = joblib.load("scaler.pkl")  # Ensure your scaler file is available
    logger.info("Model and scaler loaded successfully.")
except Exception as e:
```

```
    logger.error(f"Error loading model or scaler: {e}")
    model, scaler = None, None  # Set to None to prevent crashes

# Initialize FastAPI
app = FastAPI()

# Define request body structure
class DiabetesInput(BaseModel):
    pregnancies: int
    glucose: float
    blood_pressure: float
    skin_thickness: float
    insulin: float
    bmi: float
    diabetes_pedigree_function: float
    age: int

@app.post("/predict")
def predict_diabetes(input: DiabetesInput):
    if model is None or scaler is None:
        logger.error("Model or scaler not loaded.")
        raise HTTPException(status_code=500, detail="Model or scaler not loaded.
Check server logs.")

    try:
        input_data = np.array([[
            input.pregnancies,
            input.glucose,
            input.blood_pressure,
            input.skin_thickness,
            input.insulin,
            input.bmi,
            input.diabetes_pedigree_function,
            input.age
        ]])

        logger.info(f"Input data shape: {input_data.shape}")

        # Scale input data
        scaled_input = scaler.transform(input_data)

        # Make prediction
        prediction = model.predict(scaled_input)
        logger.info(f"Prediction: {prediction}")
        return {"prediction": int(prediction[0])}
```

```
    except Exception as e:
        logger.error(f"Error during prediction: {e}")
        raise HTTPException(status_code=500, detail="Error processing
prediction.")

# Run the API using:
# uvicorn app:app --reload
```

- **Input Validation with Pydantic:** The `DiabetesInput` class ensures that incoming data matches the expected format, making it easy to handle incoming JSON data and ensuring data quality.
- **Prediction Process:** The data is transformed into a numpy array, then scaled using the saved scaler, and finally passed through the trained model for prediction.
- **Error Handling and Logging:** Error handling is implemented to ensure that any issues during model loading, data scaling, or prediction are caught, and an appropriate error message is returned to the user. Logging helps track the flow of requests and errors.

3. Running the API

Once the FastAPI app has been defined, it can be run using **Uvicorn**, a high-performance ASGI server that runs FastAPI apps.

To run the API, use the following command:

```
uvicorn app:app --reload
```

The --reload flag enables auto-reloading, which makes it easy to develop and test the API in real-time without restarting the server manually after making code changes.

This command will start the FastAPI server locally and make the model available through an HTTP interface.

4. Testing the API

The API can be tested by sending POST requests with sample patient data. You can use tools like Postman or cURL to interact with the API.

Example POST request:

Endpoint: http://127.0.0.1:8000/predict

Request Body: A JSON object with the following format:

{

```
  "pregnancies": 6,
  "glucose": 148,
  "blood_pressure": 72,
  "skin_thickness": 35,
  "insulin": 0,
  "bmi": 33.6,
  "diabetes_pedigree_function": 0.627,
  "age": 50
}
```


8. Potential Limitations & Future Improvements

- **Limitations:**
  - Dataset size is relatively small, which may impact model generalization.
  - Logistic Regression assumes linear relationships, which may not fully capture complex patterns.
- **Future Improvements:**
  - Experiment with ensemble models (Random Forest, Gradient Boosting) for better accuracy.
  - Deploy the model using cloud services like AWS Lambda or Flask for scalability.
  - Collect additional real-world medical data to improve model robustness.