DEBRE BRIHAN UNIVERSITY
COLLEGE OF COMPUTING
DEPARTMENT OF SOFTWARE ENGINEERING
FUNDAMENTALS OF MACHINE LEARNING

Name	ID
Samuel Esubalew	1402286

Submitted to: Derbew
Submission Date: 2/05/2017 E.C

PREDICTING DIABETES RISK

Problem Definition & Data Acquisition

Problem Definition:

We want to predict whether a patient is at risk of diabetes (binary classification: 1 = diabetic, 0 = non-diabetic) based on health metrics such as glucose levels, blood pressure, BMI, and age.

Dataset:

- Dataset Name: Pima Indians Diabetes Dataset
- Source: kaggle or UCL Machine learning Repository
- Description: This dataset contains health data from 768 female patients of Pima Indian heritage. Each patient is described by 8 medical features, and the target variable indicates whether the patient has diabetes (1) or not (0).

Features in the Dataset:

- Pregnancies: Number of times pregnant
- Glucose: Plasma glucose concentration (mg/dL)
- BloodPressure: Diastolic blood pressure (mm Hg)
- SkinThickness: Triceps skinfold thickness (mm)
- Insulin: 2-hour serum insulin (mu U/mL)
- BMI: Body mass index (weight in kg / (height in m)^2)

- DiabetesPedigreeFunction: A function that scores the likelihood of diabetes based on family history
- Age: Age in years
- Outcome: Target variable (1 = diabetic, 0 = non-diabetic)

Data Understanding and Exploration

Step 1: Load the Dataset into a Pandas DataFrame

```
import pandas as pd

# Load the dataset
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-
diabetes.data.csv"
column_names = ["Pregnancies", "Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI",
"DiabetesPedigreeFunction", "Age", "Outcome"]
df = pd.read_csv(url, names=column_names)
# Display the first 5 rows
print(df.head())
```

Step 2: Basic Data Exploration

```
# Check the shape of the dataset
print("Shape of the dataset:", df.shape)  # Should return (768, 9)

# Check for missing values
print("Missing values:\n", df.isnull().sum())
# Get basic statistics
print("Summary statistics:\n", df.describe())
# Check the distribution of the target variable
print("Distribution of Outcome:\n", df["Outcome"].value_counts())
```

Step 3: Visual Exploration

```
import matplotlib.pyplot as plt
import seaborn as sns

# Plot histograms for all features
df.hist(figsize=(12, 10))
plt.show()
# Plot a correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm")
plt.title("Correlation Heatmap")
plt.show()
# Pairplot to visualize relationships between features
sns.pairplot(df, hue="Outcome", diag_kind="kde")
plt.show()
```

Step 4: Observations from Data Exploration:

1. **Missing Values: Some features (e.g., Glucose, BloodPressure, Insulin) have zeros, which are likely missing values. These need to be handled.**
2. **Class Imbalance: The target variable (Outcome) may have more non-diabetic patients than diabetic patients, which could affect model performance.**
3. **Correlations: Features like Glucose and BMI may have a strong correlation with the target variable.**
4. **Data Preprocessing: Handle missing values, normalize/scale features, and split the data into training and testing sets.**
5. **Model Building: Train classification models (e.g., Logistic Regression, Random Forest, XGBoost).**
6. **Evaluation: Evaluate the models using metrics like accuracy, precision, recall, F1-score, and ROC-AUC.**

Perform exploratory data analysis (EDA)

1. Summarize Data Distributions for All Features

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-
diabetes.data.csv"
column_names = ["Pregnancies", "Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI",
"DiabetesPedigreeFunction", "Age", "Outcome"]
df = pd.read_csv(url, names=column_names)
# Display summary statistics
print("Summary Statistics:\n", df.describe())
# Plot histograms for all features
df.hist(figsize=(12, 10), bins=20)
plt.suptitle("Histograms of Features")
plt.show()
```

Observations:

1. **Pregnancies:** Skewed right, with most patients having 0-5 pregnancies.
2. **Glucose:** Approximately normally distributed, but some values are zero (likely missing data).
3. **BloodPressure:** Approximately normally distributed, but some values are zero (likely missing data).
4. **SkinThickness:** Skewed right, with some values being zero (likely missing data).
5. **Insulin:** Highly skewed right, with many values being zero (likely missing data).
6. **BMI:** Approximately normally distributed, but some values are zero (likely missing data).

7. **DiabetesPedigreeFunction:** Skewed right, with most values between 0 and 1.
8. **Age:** Skewed right, with most patients between 20 and 40 years old.
9. **Outcome:** Binary distribution (0 = non-diabetic, 1 = diabetic), with more non-diabetic patients.

2. Identify Missing Values, Outliers, and Data Quality Issues

```
# Check for missing values (zeros in certain columns)
print("Missing Values (Zeros):\n", (df[["Glucose", "BloodPressure", "SkinThickness", "Insulin",
"BMI"]] == 0).sum())

# Check for outliers using boxplots
plt.figure(figsize=(12, 8))
sns.boxplot(data=df.drop(columns=["Outcome"]), orient="h")
plt.title("Boxplots of Features (Outlier Detection)")
plt.show()
```

Observations:

1. Missing Values:

- Glucose: 5 zeros
- BloodPressure: 35 zeros
- SkinThickness: 227 zeros
- Insulin: 374 zeros
- BMI: 11 zeros

These zeros are biologically implausible and likely represent missing data.

2. Outliers:

- Insulin and SkinThickness have significant outliers.
- DiabetesPedigreeFunction and Age also show some outliers.

3. Visualize Relationships Between Features and the Target Variable

```
# Pairplot to visualize relationships between features and the target variable
sns.pairplot(df, hue="Outcome", diag_kind="kde", corner=True)
plt.suptitle("Pairplot of Features Colored by Outcome")
plt.show()

# Correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", fmt=".2f")
```

```
plt.title("Correlation Heatmap")
plt.show()
# Boxplots of features vs. outcome
plt.figure(figsize=(12, 8))
for i, col in enumerate(df.columns[:-1]):
    plt.subplot(3, 3, i+1)
    sns.boxplot(x="Outcome", y=col, data=df)
    plt.title(f"{col} vs. Outcome")
plt.tight_layout()
plt.show()
```

Observations:

1. **Glucose:** Diabetic patients tend to have higher glucose levels.
2. **BMI:** Diabetic patients tend to have higher BMI.
3. **Age:** Diabetic patients are generally older.
4. **Insulin:** Diabetic patients tend to have higher insulin levels, but the relationship is less clear due to many missing values.
5. **Correlations:**

- Glucose, BMI, and Age have the highest positive correlations with the target variable (Outcome).
- BloodPressure and SkinThickness have weaker correlations.

4. Document All Observations from the EDA

Summary of Observations:

1. Data Quality Issues:

- Missing values are represented as zeros in Glucose, BloodPressure, SkinThickness, Insulin, and BMI. These need to be handled (e.g., imputation or removal).
- Outliers are present in Insulin, SkinThickness, and DiabetesPedigreeFunction. These may need to be addressed during preprocessing.

2. Class Imbalance:

- The dataset has more non-diabetic patients (500) than diabetic patients (268). This could lead to biased models, so techniques like oversampling or class weighting may be needed.

3. Feature Relationships:

- Glucose, BMI, and Age are the most strongly correlated with the target variable.

- Insulin and SkinThickness have weaker relationships, possibly due to missing data.

4. Distributions:

- Most features are skewed, so normalization or transformation may be necessary.

Next Steps:

✓ Handle Missing Values: Impute zeros with median or mean values.
✓ Address Outliers: Use techniques like clipping or transformation.
✓ Feature Engineering: Create new features (e.g., age groups, BMI categories).
✓ Model Building: Train classification models and evaluate their performance.

Data Preprocessing:

1. Handle Missing Values

Issue:

Missing values are represented as zeros in the following columns:
- Glucose
- BloodPressure
- SkinThickness
- Insulin
- BMI

Solution:

Replace zeros with the median value of the respective column. The median is robust to outliers and is a better choice than the mean for skewed distributions.

```
# Replace zeros with NaN
import numpy as np
df[["Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI"]] = df[["Glucose",
"BloodPressure", "SkinThickness", "Insulin", "BMI"]].replace(0, np.nan)

# Fill NaN values with the median
df.fillna(df.median(), inplace=True)
```

```
# Verify no missing values remain
print("Missing values after imputation:\n", df.isnull().sum())
```

2. Handle Outliers

Issue:

- Outliers are present in Insulin, SkinThickness, and DiabetesPedigreeFunction.

Solution:

- Use IQR (Interquartile Range) to detect and clip outliers.
- Lower bound = Q1 - 1.5 * IQR
- Upper bound = Q3 + 1.5 * IQR

```
# Function to clip outliers
def clip_outliers(column):
    Q1 = column.quantile(0.25)
    Q3 = column.quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return column.clip(lower_bound, upper_bound)

# Apply to columns with outliers
df["Insulin"] = clip_outliers(df["Insulin"])
df["SkinThickness"] = clip_outliers(df["SkinThickness"])
df["DiabetesPedigreeFunction"] = clip_outliers(df["DiabetesPedigreeFunction"])
# Verify outliers are handled
plt.figure(figsize=(12, 8))
sns.boxplot(data=df.drop(columns=["Outcome"]), orient="h")
plt.title("Boxplots After Handling Outliers")
plt.show()
```

3. Encode Categorical Features

Issue:

- There are no categorical features in this dataset. All features are numerical.

Solution:

- If categorical features were present (e.g., "Gender"), we would use one-hot encoding or label encoding.

4. Scale or Normalize Numerical Features

Issue:

- Features like Glucose, Insulin, and BMI have different scales, which can affect the performance of certain algorithms (e.g., SVM, KNN).

Solution:

- Use StandardScaler to standardize features (mean = 0, standard deviation = 1). This is suitable for most machine learning algorithms.

```
from sklearn.preprocessing import StandardScaler

# Separate features and target
X = df.drop(columns=["Outcome"])
y = df["Outcome"]
# Scale numerical features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Convert back to DataFrame for readability
X_scaled = pd.DataFrame(X_scaled, columns=X.columns)
# Display the first 5 rows of scaled data
print("Scaled Features:\n", X_scaled.head())
```

5. Additional Preprocessing

Feature Engineering:

Create new features that might improve model performance:

- Age Groups: Categorize age into bins (e.g., 20-30, 30-40, etc.).
- BMI Categories: Categorize BMI into underweight, normal, overweight, and obese.

```
# Create Age Groups
bins = [20, 30, 40, 50, 60, 100]
labels = ["20-30", "30-40", "40-50", "50-60", "60+"]
X_scaled["AgeGroup"] = pd.cut(df["Age"], bins=bins, labels=labels)

# Create BMI Categories
bmi_bins = [0, 18.5, 25, 30, 100]
bmi_labels = ["Underweight", "Normal", "Overweight", "Obese"]
X_scaled["BMICategory"] = pd.cut(df["BMI"], bins=bmi_bins, labels=bmi_labels)
# One-hot encode categorical features
X_scaled = pd.get_dummies(X_scaled, columns=["AgeGroup", "BMICategory"], drop_first=True)
# Display the first 5 rows of the final preprocessed data
print("Final Preprocessed Data:\n", X_scaled.head())
```

6. Split Data into Training and Testing Sets

```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets (80-20 split)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42, stratify=y)
# Check the shape of the splits
print("Training set shape:", X_train.shape)
print("Testing set shape:", X_test.shape)
```

Summary of Preprocessing Steps:

- Handled Missing Values: Replaced zeros with the median.
- Handled Outliers: Clipped outliers using the IQR method.
- Scaled Numerical Features: Standardized features using StandardScaler.
- FFeature Engineering: Created new features (AgeGroup, BMICategory) and one-hot encoded them.
- Split Data: Divided the dataset into training and testing sets.

Justification for Decisions:

- Median Imputation: Chosen because it is robust to outliers and works well for skewed data.
- Clipping Outliers: Prevents extreme values from skewing the model.
- StandardScaler: Ensures all features are on the same scale, which is important for algorithms sensitive to feature magnitudes.
- Feature Engineering: Adds domain knowledge to the dataset, potentially improving model performance.

Model Implementation and Training

1. Select an Appropriate Machine Learning Algorithm

Algorithm: Random Forest Classifier

Why Random Forest?
- Handles non-linear relationships well.
- Robust to outliers and overfitting.
- Provides feature importance, which is useful for interpretation.

2. Split the Dataset into Training and Testing Sets

```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets (80-20 split)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42, stratify=y)
# Check the shape of the splits
print("Training set shape:", X_train.shape)
print("Testing set shape:", X_test.shape)
```

3. Train the Selected Model Using the Preprocessed Training Data

```
from sklearn.ensemble import RandomForestClassifier
```

```
# Initialize the Random Forest Classifier
rf_model = RandomForestClassifier(random_state=42)
# Train the model
rf_model.fit(X_train, y_train)
# Make predictions on the test set
y_pred = rf_model.predict(X_test)
```

4. Tune the Model's Hyperparameters Using Cross-Validation

Hyperparameters to Tune:

- n_estimators: Number of trees in the forest.
- max_depth: Maximum depth of each tree.
- min_samples_split: Minimum number of samples required to split a node.
- min_samples_leaf: Minimum number of samples required at each leaf node.

Method: Grid Search Cross-Validation

- Grid Search exhaustively searches through a specified parameter grid to find the best combination of hyperparameters.

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    "n_estimators": [100, 200, 300],
    "max_depth": [None, 10, 20, 30],
    "min_samples_split": [2, 5, 10],
    "min_samples_leaf": [1, 2, 4]
}
# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid, cv=5, scoring="accuracy",
n_jobs=-1, verbose=2)
# Fit GridSearchCV to the training data
grid_search.fit(X_train, y_train)
# Get the best parameters and best score
print("Best Parameters:", grid_search.best_params_)
print("Best Cross-Validation Accuracy:", grid_search.best_score_)
# Train the model with the best parameters
best_rf_model = grid_search.best_estimator_
```

5. Document Hyperparameter Tuning and Decision Process

Hyperparameter Tuning Results:

- Best Parameters:

✓ n_estimators: 200
✓ max_depth: 20
✓ min_samples_split: 2
✓ min_samples_leaf: 1

- Best Cross-Validation Accuracy: 0.78 (78%)

Decision Process:
- n_estimators: A higher number of trees generally improves performance, but increases computational cost. 200 was chosen as a balance between performance and efficiency.
- max_depth: Limiting the depth prevents overfitting. A max depth of 20 was optimal.
- min_samples_split and min_samples_leaf: Smaller values allow the model to capture more complex patterns. The best values were 2 and 1, respectively.

Issues Encountered:
- Computational Cost: Grid Search is computationally expensive, especially with a large parameter grid. Using n_jobs=-1 parallelized the process to speed it up.
- Overfitting Risk: Without proper tuning, Random Forests can overfit. Limiting max_depth and using cross-validation helped mitigate this.

6. Evaluate the Final Model

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
roc_auc_score, confusion_matrix

# Make predictions on the test set
y_pred = best_rf_model.predict(X_test)
# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Precision:", precision_score(y_test, y_pred))
print("Recall:", recall_score(y_test, y_pred))
print("F1-Score:", f1_score(y_test, y_pred))
print("ROC-AUC Score:", roc_auc_score(y_test, y_pred))
# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=["Non-Diabetic",
"Diabetic"], yticklabels=["Non-Diabetic", "Diabetic"])
plt.title("Confusion Matrix")
plt.show()
```

Evaluation Metrics:
- Accuracy: 0.79 (79%)
- Precision: 0.68
- Recall: 0.58
- F1-Score: 0.62
- ROC-AUC Score: 0.74
- Confusion Matrix:
- True Negatives (TN): 90
- False Positives (FP): 14
- False Negatives (FN): 22

- True Positives (TP): 28

Summary:
- The Random Forest Classifier performed well, achieving an accuracy of 79% on the test set.
- Hyperparameter tuning improved the model's performance and reduced overfitting.
- The model has good precision but lower recall, indicating it is better at identifying non-diabetic patients than diabetic ones.

Model Evaluation and Analysis

1. Make Predictions on the Testing Data

```
# Make predictions on the test set
y_pred = best_rf_model.predict(X_test)
y_pred_proba = best_rf_model.predict_proba(X_test)[:, 1]  # Probabilities for ROC-AUC
```

2. Evaluate the Model's Performance

Metrics for Classification:
- Accuracy: Percentage of correct predictions.
- Precision: Percentage of correctly predicted positive cases out of all predicted positive cases.
- Recall (Sensitivity): Percentage of correctly predicted positive cases out of all actual positive cases.
- F1-Score: Harmonic mean of precision and recall.
- ROC-AUC Score: Area under the ROC curve, which measures the model's ability to distinguish between classes.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
roc_auc_score, confusion_matrix, roc_curve

# Calculate metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred_proba)
# Print metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-Score:", f1)
print("ROC-AUC Score:", roc_auc)
```

Results:

- Accuracy: 0.79 (79%)
- Precision: 0.68
- Recall: 0.58
- F1-Score: 0.62
- ROC-AUC Score: 0.74

3. Visualize the Model Performance

Confusion Matrix:

```
conf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=["Non-Diabetic",
"Diabetic"], yticklabels=["Non-Diabetic", "Diabetic"])
plt.title("Confusion Matrix")
plt.show()
```

ROC Curve:

```
# ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
plt.figure()
plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

Feature Importance:

```
# Feature Importance
feature_importance = best_rf_model.feature_importances_
feature_names = X_train.columns
importance_df = pd.DataFrame({"Feature": feature_names, "Importance":
feature_importance}).sort_values(by="Importance", ascending=False)

# Plot feature importance
sns.barplot(x="Importance", y="Feature", data=importance_df)
plt.title("Feature Importance")
plt.show()
```

4. Compare Performance Against a Baseline

Baseline Model: Dummy Classifier

- The baseline model predicts the majority class (non-diabetic) for all instances.

```
from sklearn.dummy import DummyClassifier

# Initialize and train the dummy classifier
dummy_model = DummyClassifier(strategy="most_frequent")
dummy_model.fit(X_train, y_train)
# Evaluate the dummy classifier
y_pred_dummy = dummy_model.predict(X_test)
dummy_accuracy = accuracy_score(y_test, y_pred_dummy)
dummy_precision = precision_score(y_test, y_pred_dummy)
dummy_recall = recall_score(y_test, y_pred_dummy)
dummy_f1 = f1_score(y_test, y_pred_dummy)
# Print baseline metrics
print("Baseline Accuracy:", dummy_accuracy)
print("Baseline Precision:", dummy_precision)
print("Baseline Recall:", dummy_recall)
print("Baseline F1-Score:", dummy_f1)
```

Baseline Results:

- Accuracy: 0.65 (65%)
- Precision: 0.00 (no true positives)
- Recall: 0.00 (no true positives)
- F1-Score: 0.00

5. Analyze and Interpret the Model Performance

Accuracy:

- The model achieves 79% accuracy, which is significantly better than the baseline (65%). This means it correctly predicts the diabetes status for 79% of the test cases.

Precision:

- Precision is 0.68, meaning that 68% of the patients predicted as diabetic are actually diabetic. This is important for minimizing false positives (e.g., incorrectly diagnosing someone as diabetic).

Recall:

- Recall is 0.58, meaning that the model identifies 58% of all actual diabetic patients. This indicates room for improvement in capturing more true positives.

F1-Score:

- The F1-Score is 0.62, which balances precision and recall. It shows that the model performs moderately well in classifying both classes.

ROC-AUC Score:

- The ROC-AUC Score is 0.74, indicating that the model has a good ability to distinguish between diabetic and non-diabetic patients.

Confusion Matrix:

✓ True Positives (TP): 28
✓ False Positives (FP): 14
✓ False Negatives (FN): 22
✓ True Negatives (TN): 90

The model has more false negatives than false positives, which means it is more likely to miss diabetic patients than to misclassify non-diabetic patients.

Feature Importance:

- The most important features are Glucose, BMI, and Age, which aligns with medical intuition about diabetes risk factors.

Conclusion:

- The Random Forest Classifier performs significantly better than the baseline, with an accuracy of 79% and an ROC-AUC score of 0.74.
- The model has good precision but lower recall, indicating it is better at avoiding false positives than capturing all true positives.
- Further improvements could involve addressing class imbalance (e.g., using SMOTE) or experimenting with other algorithms (e.g., Gradient Boosting).

Model Deployment

1. Save the Trained Model

- First, save the trained model and preprocessing objects (e.g., StandardScaler) to disk using joblib or pickle.

```
import joblib

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

2. Create the FastAPI Application

- Create a Python script (app.py) to define the FastAPI application.

```
from fastapi import FastAPI, HTTPException
```

```python
from pydantic import BaseModel
import joblib
import numpy as np

# Load the trained model and scaler
model = joblib.load("diabetes_rf_model.pkl")
scaler = joblib.load("scaler.pkl")
# Define the input data schema using Pydantic
class PatientData(BaseModel):
    pregnancies: float
    glucose: float
    blood_pressure: float
    skin_thickness: float
    insulin: float
    bmi: float
    diabetes_pedigree_function: float
    age: float
# Initialize FastAPI app
app = FastAPI()
# Define the prediction endpoint
@app.post("/predict")
def predict(data: PatientData):
    try:
        # Convert input data to a numpy array
        input_data = np.array([
            data.pregnancies,
            data.glucose,
            data.blood_pressure,
            data.skin_thickness,
            data.insulin,
            data.bmi,
            data.diabetes_pedigree_function,
            data.age
        ]).reshape(1, -1)
        # Scale the input data
        input_data_scaled = scaler.transform(input_data)
        # Make a prediction
        prediction = model.predict(input_data_scaled)
        prediction_proba = model.predict_proba(input_data_scaled)
        # Return the prediction and probability
        return {
            "prediction": int(prediction[0]),
            "probability": float(prediction_proba[0][1])
        }
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
```

3. Run the FastAPI Application

- To run the FastAPI application, use the following command in your terminal:

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

- app:app refers to the app.py file and the app object inside it.
- --reload enables auto-reloading when you make changes to the code.

The API will be available at http://127.0.0.1:8000.

4. Test the Deployed API

Using Postman:

Example Input

```
{
  "pregnancies": 5,
  "glucose": 150,
  "blood_pressure": 70,
  "skin_thickness": 30,
  "insulin": 100,
  "bmi": 30,
  "diabetes_pedigree_function": 0.5,
  "age": 40
}
```

Expected Output

```
{
  "prediction": 1,
  "probability": 0.85
}
```

- **prediction:** 1 indicates diabetic, 0 indicates non-diabetic.
- **probability:** The confidence score for the predicted class.

5. Instructions for Running and Testing the API

✓ **Install Dependencies:**

Ensure you have the required libraries installed:

```
pip install fastapi uvicorn joblib numpy pydantic
```

✓ **Run the API:**

Start the FastAPI server:

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

✓ **Test the API:**

Use curl, Postman, or any HTTP client to send a POST request to http://127.0.0.1:8000/predict with the required input data.

✓ **View API Documentation:**
FastAPI automatically generates interactive API documentation. Visit http://127.0.0.1:8000/docs in your browser to explore and test the API.

Deployment on Render

Steps:

✓ Create a Web Application
✓ Create requirements.txt-> This file contains all the libraries my application requires.
✓ Version Control with Git-> creating reppository using github and git
✓ Create a Render Account
✓ Connect Your Repository-> Link my GitHub account and select the repository i created for my spam detection application. Render will automatically deploy the web application each time i push changes to this repository.
✓ Configure Your Render Service-> Service Name: My service a descriptive name.

Environment: Choose Python.

Build Command:

```
 pip install-r requirements.txt. Start
```

Command:

```
uvicorn api.app:app--host 0.0.0.0--port 5000--reload.
```