Diabetes Prediction System
Detailed Project Report

Diabetes Prediction System – Detailed Notebook Documentation

This document explains each section and function in the notebook, clarifying what it does and why it is used, following best practices for machine learning projects.

---

1. Import Necessary Libraries

This section imports all required libraries for the project:

• **Data handling:** `pandas`, `numpy`
• **Visualization:** `matplotlib`, `seaborn`
• **Preprocessing:** `StandardScaler`, `SimpleImputer`
• **Modeling & evaluation:** `train_test_split`, classification metrics
• **Imbalanced data handling:** Custom oversampling implementation

These imports support the full ML pipeline from data exploration to model deployment.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

---

2. Data Loading & Initial Exploration

load_data(path)

Loads the Pima Indians Diabetes dataset from a CSV file and returns a Pandas DataFrame.

data = pd.read_csv('diabetes.csv')
data.head()

Dataset Overview:
• **Source:** Pima Indians Diabetes Database
• **Samples:** 768 patients
• **Features:** 8 physiological measurements
• **Target:** Binary classification (Diabetic/Non-Diabetic)

---

3. Basic Data Exploration Functions

This section performs comprehensive data exploration:

3.1 Dataset Structure

• `data.shape`: Returns (768, 9) 768 rows, 9 columns
• `data.info()`: Displays data types and memory usage
• `data.describe()`: Provides descriptive statistics

3.2 Missing Values Detection

Identifies zero values that represent missing data in medical context:

cols_cannot_be_zero = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']

for col in cols_cannot_be_zero:
zero_count = (data[col] == 0).sum()
percentage = (zero_count / len(data)) * 100
print(f"{col}: {zero_count} ({percentage:.2f}%)")

Missing Values Summary:
Feature	Zero Count	Percentage
Glucose	5	0.65%
BloodPressure	35	4.56%
SkinThickness	227	29.56%
Insulin	374	48.70%
BMI	11	1.43%

3.3 Duplicate Detection

data.duplicated().sum() # Returns 0 - No duplicates found

---

4. Feature Description

Feature	Description	Unit	Normal Range
Pregnancies	Number of pregnancies	Count	0-17
Glucose	Plasma glucose concentration	mg/dL	70-100
BloodPressure	Diastolic blood pressure	mm Hg	60-80
SkinThickness	Triceps skin fold thickness	mm	10-50
Insulin	2-Hour serum insulin	mu U/ml	16-166
BMI	Body mass index	kg/m²	18.5-25
DiabetesPedigreeFunction	Diabetes hereditary risk	Score	0.08-2.42
Age	Patient age	Years	21-81
Outcome	Diabetes diagnosis	Binary	0 or 1

---

5. Target Class Distribution Analysis

target_distribution(df)

Analyzes the balance between diabetic and non-diabetic cases:

data['Outcome'].value_counts(normalize=True) * 100

Results:
• **Non-Diabetic (0):** 65.10% (500 cases)
• **Diabetic (1):** 34.90% (268 cases)

■■ **Observation:** The dataset is imbalanced, requiring special handling during model training.

---

6. Correlation Analysis

correlation_with_target(df)

Calculates and ranks feature correlations with the target variable:

corr_matrix = data.corr(numeric_only=True)
print(corr_matrix['Outcome'].sort_values(ascending=False))

Correlation Ranking:

Feature	Correlation
Glucose	0.467
BMI	0.293
Age	0.238
Pregnancies	0.222
DiabetesPedigreeFunction	0.174
Insulin	0.131
SkinThickness	0.075
BloodPressure	0.065

Key Insight: Glucose level is the strongest predictor of diabetes.

---

7. Data Preprocessing – Missing Values Handling

handle_missing_values(df)

Replaces biologically impossible zero values with NaN, then imputes using median strategy:

zero_features = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
data[zero_features] = data[zero_features].replace(0, np.nan)

imputer = SimpleImputer(strategy='median')
data[zero_features] = imputer.fit_transform(data[zero_features])

Why Median Imputation?
• Robust to outliers
• Preserves data distribution
• Appropriate for skewed medical data

---

8. Outlier Detection & Treatment

8.1 Outlier Detection Using IQR

Visualizes outliers using boxplots for each numerical feature:

numerical_cols = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age']

8.2 Outlier Removal Using IQR Method

for col in numerical_cols:
Q1 = data[col].quantile(0.25)
Q3 = data[col].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
data = data[(data[col] >= lower_bound) & (data[col] <= upper_bound)]

IQR Method Explanation:
• Q1: 25th percentile
• Q3: 75th percentile
• IQR = Q3 Q1
• Valid range: [Q1 1.5×IQR, Q3 + 1.5×IQR]

This step removes extreme values while preserving the majority of data.

---

9. Feature & Target Separation

split_features_target(df)

Separates the dataset into features (X) and target (y):

X = data[numerical_cols]
y = data['Outcome']

Features (X): 8 physiological measurements
Target (y): Binary outcome (0 = Non-Diabetic, 1 = Diabetic)

---

10. Train/Test Split

train_test_split_stratified(X, y)

Splits data while maintaining class distribution:

```
X_train, X_test, y_train, y_test = train_test_split(
X, y,
test_size=0.2,
random_state=42,
stratify=y
)
```

Split Configuration:
• Training set: 80%
• Test set: 20%
• Stratified: Yes (preserves class ratio)
• Random state: 42 (reproducibility)

---

11. Feature Scaling

scale_features(X_train, X_test)

Applies StandardScaler to normalize features:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Why StandardScaler?
• Centers data (mean = 0)
• Scales to unit variance (std = 1)
• Essential for gradient-based algorithms
• Improves convergence speed

■■ **Important:** Scaler is fitted only on training data to prevent data leakage.

---

12. Class Imbalance Handling – Oversampling

balance_classes(X_train, y_train)

Implements random oversampling to balance minority class:

from collections import Counter

class_counts = Counter(y_train)
majority_class = max(class_counts, key=class_counts.get)
minority_class = min(class_counts, key=class_counts.get)

minority_indices = np.where(y_train.values == minority_class)[0]
n_samples_needed = class_counts[majority_class] - class_counts[minority_class]

np.random.seed(42)
oversample_indices = np.random.choice(minority_indices, size=n_samples_needed, replace=True)
X_train_balanced = np.vstack([X_train_scaled, X_train_scaled[oversample_indices]])
y_train_balanced = np.concatenate([y_train.values, y_train.values[oversample_indices]])

Why Oversampling?
• Addresses class imbalance (65% vs 35%)
• Prevents model bias toward majority class
• Applied only on training data

---

13. Model Implementation – Manual Logistic Regression

LogisticRegressionManual Class

A custom implementation of Logistic Regression built from scratch:

class LogisticRegressionManual:
def __init__(self, learning_rate=0.01, n_iterations=1000,
regularization='l2', lambda_reg=0.01):
self.learning_rate = learning_rate
self.n_iterations = n_iterations
self.regularization = regularization
self.lambda_reg = lambda_reg
self.weights = None
self.bias = None
self.cost_history = []

13.1 Sigmoid Activation Function

def sigmoid(self, z):
return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

Purpose: Converts linear output to probability [0, 1]

13.2 Cost Function with L2 Regularization

def compute_cost(self, y, y_pred, weights):
m = len(y)
epsilon = 1e-15
cost = -1/m * np.sum(y * np.log(y_pred + epsilon) +
(1 - y) * np.log(1 - y_pred + epsilon))
if self.regularization == 'l2':
cost += (self.lambda_reg / (2 * m)) * np.sum(weights ** 2)
return cost

Components:
• Binary Cross-Entropy Loss
• L2 Regularization term (prevents overfitting)
• Epsilon for numerical stability

13.3 Gradient Descent Training

def fit(self, X, y):
m, n = X.shape
self.weights = np.zeros(n)
self.bias = 0
y = np.array(y)

for i in range(self.n_iterations):
z = np.dot(X, self.weights) + self.bias
y_pred = self.sigmoid(z)

dw = (1/m) * np.dot(X.T, (y_pred - y))
db = (1/m) * np.sum(y_pred - y)

if self.regularization == 'l2':
dw += (self.lambda_reg / m) * self.weights

self.weights -= self.learning_rate * dw
self.bias -= self.learning_rate * db

cost = self.compute_cost(y, y_pred, self.weights)
self.cost_history.append(cost)

13.4 Prediction Methods

def predict_proba(self, X):
z = np.dot(X, self.weights) + self.bias
return self.sigmoid(z)

def predict(self, X, threshold=0.5):
return (self.predict_proba(X) >= threshold).astype(int)

---

14. Model Training

train_model()

Trains the Logistic Regression model with optimized hyperparameters:

model = LogisticRegressionManual(
learning_rate=0.1,
n_iterations=1000,
regularization='l2',
lambda_reg=0.01
)
model.fit(X_train_balanced, y_train_balanced)

Hyperparameters:
Parameter	Value	Description
learning_rate	0.1	Step size for gradient descent
n_iterations	1000	Number of training epochs
regularization	L2	Ridge regularization
lambda_reg	0.01	Regularization strength

---

15. Model Evaluation

evaluate_model(model, X_test, y_test)

Evaluates model performance on test data:

y_pred = model.predict(X_test_scaled)
accuracy = np.mean(y_pred == y_test.values)
print(f"Model Accuracy: {accuracy:.1%}")

Results:
Metric	Value
Accuracy	**80.6%**
Recall	~75%
Precision	~78%

---

16. Risk Level Classification

classify_risk(probability)

Categorizes prediction probability into risk levels:

if probability > 0.7:
risk_level = "High"
elif probability > 0.4:
risk_level = "Medium"
else:
risk_level = "Low"

Risk Categories:
Probability	Risk Level	Color Code
> 70%	High	Red
40% - 70%	Medium	Orange
< 40%	Low	Green

---

17. Web Application – Flask Implementation

17.1 Application Structure

diabetes_web_app/
■■■ app.py # Main Flask application
■■■ requirements.txt # Dependencies
■■■ templates/
■■■ index.html # Input form page
■■■ result.html # Results display page

17.2 Flask Routes

Home Route:
@app.route('/')
def home():
return render_template('index.html')

Prediction Route:
@app.route('/predict', methods=['POST'])
def predict():
features = [
float(request.form['pregnancies']),
float(request.form['glucose']),
float(request.form['blood_pressure']),
float(request.form['skin_thickness']),
float(request.form['insulin']),
float(request.form['bmi']),
float(request.form['dpf']),
float(request.form['age'])
]

features_array = np.array(features).reshape(1, -1)
features_scaled = scaler.transform(features_array)

probability = model.predict_proba(features_scaled)[0]
prediction = 1 if probability >= 0.5 else 0

return render_template('result.html', result=result)

17.3 Running the Local Application

cd diabetes_web_app
pip install -r requirements.txt

python app.py

Open browser: http://localhost:5000

---

18. Cloud Deployment – Hugging Face Spaces

18.1 FastAPI Implementation

from fastapi import FastAPI, Form
from fastapi.responses import HTMLResponse

app = FastAPI()

@app.get("/", response_class=HTMLResponse)
async def home():
return INDEX_HTML

@app.post("/predict", response_class=HTMLResponse)
async def predict(
pregnancies: float = Form(...),
glucose: float = Form(...),

... other parameters

):

Prediction logic

return RESULT_HTML.format(...)

18.2 Docker Configuration

FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "7860"]

18.3 Deployment Files

File	Purpose
app.py	FastAPI application
Dockerfile	Container configuration
requirements.txt	Python dependencies
diabetes.csv	Training data
README.md	Space description

---

19. User Interface Features

19.1 Input Form

• Arabic RTL interface
• 8 health indicator inputs
• Input validation
• Helpful tooltips with normal ranges

19.2 Results Display

• Clear diagnosis (Diabetic/Non-Diabetic)
• Probability percentage
• Risk level indicator
• Progress bar visualization
• Personalized medical recommendations

19.3 Medical Recommendations

For Diabetic Prediction:
• Consult a doctor immediately
• Perform HbA1c test
• Monitor blood sugar regularly
• Follow low-sugar diet
• Exercise 30 minutes daily

For Non-Diabetic Prediction:
• Maintain healthy lifestyle
• Exercise 150 minutes weekly

• Eat balanced diet
• Annual diabetes screening

---

20. Project Summary

20.1 Technical Stack

Category	Technologies
Language	Python 3.x
Data Processing	Pandas, NumPy
Visualization	Matplotlib, Seaborn
ML Framework	Scikit-learn (utilities)
Web Framework	Flask, FastAPI
Frontend	HTML, CSS, Bootstrap 5
Deployment	Docker, Hugging Face Spaces

20.2 Key Achievements

■ Built custom Logistic Regression from scratch
■ Achieved **80.6% accuracy** on test data
■ Handled missing values and outliers
■ Addressed class imbalance with oversampling
■ Developed bilingual web interface (Arabic)
■ Deployed to cloud platform

20.3 Challenges & Solutions

Challenge	Solution
Missing values disguised as zeros	Median imputation
Class imbalance (65% vs 35%)	Random oversampling
Outliers in medical data	IQR-based removal
Model overfitting	L2 regularization

20.4 Future Improvements

• Implement additional algorithms (Random Forest, XGBoost)
• Add more health features
• Develop mobile application
• Integrate with hospital systems
• Add patient history tracking

Conclusion

This notebook demonstrates a complete, professional ML workflow for diabetes prediction, including:

• **Exploratory Data Analysis** with comprehensive statistics
• **Data preprocessing** with missing value and outlier handling
• **Feature engineering** and scaling
• **Class imbalance treatment** using oversampling
• **Custom model implementation** from scratch
• **Model evaluation** with multiple metrics
• **Web application development** with Flask and FastAPI
• **Cloud deployment** on Hugging Face Spaces

The project achieves **80.6% accuracy** and provides a user-friendly Arabic interface for diabetes risk assessment.

■■ **Disclaimer:** This system is for educational purposes only and should not replace professional medical diagnosis.

Documentation Date: December 2024
Project: Diabetes Prediction System