DEBRE BERHAN UNIVERSITY
COLLEGE OF COMPUTING
DEPARTMENT OF SOFTWARE ENGINEERING

INDIVIDUAL ASSIGNMENT

Course Name: *:Fundamentals of Machine Learning*
Course code : **SEng4091**

Name: **Naol Meseret**
ID: **DBU1402037**

SUBMITED TO DEREBEW F

Machine Learning Project Report

Title: Diabètes *Prediction using Machine Learning*
Author: Naol Meseret
Date: *February 2025*
Course: *Fundamentals of Machine Learning*

Introduction

Diabetes is a chronic disease that affects millions of people worldwide. It is a metabolic disorder that results in high blood sugar levels due to the body's inability to produce or effectively use insulin. Early detection of diabetes is crucial for effective management and treatment, helping to prevent severe complications such as heart disease, kidney failure, and nerve damage.

In this project, I aim to build a **machine learning model** that can predict whether a person has diabetes based on medical diagnostic measurements. Using a dataset containing various health indicators such as **glucose level, blood pressure, insulin level, BMI, and age**, we will train and evaluate different machine learning algorithms to determine the most accurate model for diabetes prediction.

The objective of this project is to:

- **Analyze** the dataset and understand key patterns using **Exploratory Data Analysis (EDA)**.
- **Preprocess** the data by handling missing values and normalizing features.
- **Train multiple machine learning models** (e.g., Logistic Regression, Random Forest, SVM) and compare their performance.
- **Evaluate** the models using accuracy, precision, recall, and F1-score.
- **Deploy** the best-performing model as an API or web application for real-world use.

By leveraging **machine learning techniques**, this project aims to provide a reliable tool for predicting diabetes risk, which can assist healthcare professionals and individuals in early diagnosis and treatment planning.

2. Dataset Description

The dataset used in this project contains medical diagnostic measurements that help predict the likelihood of diabetes in patients. The data consists of multiple features that represent important health indicators related to diabetes.
Dataset Overview:

- **Source:** The dataset is based on real medical records and is commonly used for diabetes prediction tasks.
- **Format:** CSV (Comma-Separated Values)
- **Number of Instances (Rows):** 769
- **Number of Features (Columns):** 9
- **Target Variable:** The last column (Outcome), which indicates whether a patient has diabetes (1) or not (0).

Features (Columns) in the Dataset:

Feature Name	Description
Pregnancies	Number of times the patient has been 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 represents the genetic likelihood of diabetes

Feature Name	Description
Age	Age of the patient (years)
Outcome	Target variable: 1 = Diabetes, 0 = No Diabetes

Data Cleaning & Preprocessing

Before training our machine learning model, I performed the following preprocessing steps:

- **Checked for missing values** and handled them by using mean/mode imputation.
- **Replaced zero values** (e.g., in Glucose, BloodPressure, BMI, Insulin) with NaN and imputed them.
- **Standardized the numerical features** using StandardScaler to improve model performance.
- **Split the dataset** into training (80%) and testing (20%) subsets.

This dataset serves as the foundation for training our model to accurately predict the likelihood of diabetes in new patients.

3,Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is a crucial step in understanding the dataset and uncovering patterns, trends, and potential issues such as missing values or outliers. In this section, we analyze the dataset using statistical summaries and visualizations to gain insights into the data.

3.1 Statistical Summary

To understand the distribution of the dataset, we computed basic descriptive statistics, including mean, standard deviation, minimum, and maximum values for each feature.

Example of dataset summary using data.describe():

Feature	Mean	Std Dev	Min	25%	50%	75%	Max
Pregnancies	X.XX	X.XX	0	X.XX	X.XX	X.XX	XX
Glucose	X.XX	X.XX	0	X.XX	X.XX	X.XX	XXX
BloodPressure	X.XX	X.XX	0	X.XX	X.XX	X.XX	XXX
SkinThickness	X.XX	X.XX	0	X.XX	X.XX	X.XX	XXX
Insulin	X.XX	X.XX	0	X.XX	X.XX	X.XX	XXX
BMI	X.XX	X.XX	0	X.XX	X.XX	X.XX	XX.X
DiabetesPedigreeFunction	X.XX	X.XX	0.078	X.XX	X.XX	X.XX	X.XX
Age	X.XX	X.XX	21	X.XX	X.XX	X.XX	XX

Observations:

- ❖ The **minimum values** of some features (e.g., Glucose, BloodPressure, BMI, Insulin) are **zero**, which is unrealistic for medical data. These need to be handled appropriately.
- ❖ The **mean and standard deviation** suggest significant variation in feature values, indicating the need for **feature scaling** before model training.

3.2 Data Visualization

I visualized the data using different plots to gain insights.

3.2.1 Distribution of Features

I plotted **histograms** for each feature to observe their distribution.

Example: Histogram of Glucose Levels

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

plt.figure(figsize=(8, 5))
sns.histplot(data['Glucose'], bins=30, kde=True, color="blue")
plt.title("Distribution of Glucose Levels")
plt.xlabel("Glucose Level")
plt.ylabel("Frequency")
plt.show()
```

Insights:

❖ The distribution appears right-skewed, meaning some patients have extremely high glucose levels.
❖ There are zero values, which need to be replaced with the median or mean.

3.2.2 Correlation Heatmap

We used a **correlation matrix** to identify relationships between features.

Heatmap Code:

```
plt.figure(figsize=(10, 6))
sns.heatmap(data.corr(), annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Feature Correlation Heatmap")
plt.show()
```

Insights:

❖ Glucose and Outcome have a **strong positive correlation**, suggesting that higher glucose levels are associated with diabetes.
❖ BMI and Insulin also show some correlation with Outcome, indicating their significance in prediction.

3.2.3 Diabetes vs. Non-Diabetes Count

I checked how many individuals in the dataset have diabetes (Outcome = 1) vs. those who don't (Outcome = 0).

Bar Plot Code:

```
sns.countplot(x=data['Outcome'], palette=["green", "red"])


plt.title("Diabetes vs. Non-Diabetes Count")
plt.xlabel("Diabetes (1 = Yes, 0 = No)")
plt.ylabel("Count")
plt.show()
```

Insights:

❖ The dataset is **imbalanced**, with more non-diabetic cases than diabetic cases.
❖ We might need **SMOTE (Synthetic Minority Over-sampling Technique)** or class-weight adjustments during model training.

3.3 Handling Missing or Incorrect Values

- ❖ Replaced **zero values** in Glucose, BloodPressure, BMI, Insulin with **NaN**, then filled them with the **mean/median**.
- ❖ Checked for **outliers** using **boxplots** and removed extreme values.
- ❖ Standardized numerical features using StandardScaler.

Summary of EDA

✅ **Glucose, BMI, and Age** are **important features** for predicting diabetes.
✅ **Missing values and zeros** were handled to improve model reliability.
✅ **Feature scaling** ensures better model performance.
✅ **Correlation heatmap** helped identify relationships between variables.

4. Data Preprocessing

Data preprocessing is a crucial step in preparing the dataset for model training. It involves cleaning, transforming, and structuring the data to improve model performance. In this section, we handle missing values, encode categorical variables, and scale numerical features.

4.1 Handling Missing Values

From the Exploratory Data Analysis (EDA), I dentified that some numerical columns have **zero values**, which are unrealistic for medical features like Glucose, BloodPressure, BMI, and Insulin.

✅ **Solution:** I replace these zero values with the **mean/median** of the respective columns.

Code to Handle Missing Values:

```
import numpy as np

# Replace 0 values with NaN (only for relevant columns)
cols_with_zeros = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
data[cols_with_zeros] = data[cols_with_zeros].replace(0, np.nan)

# Fill missing values with the median
data.fillna(data.median(), inplace=True)
```

Why Median?

- ❖ The median is more **robust to outliers** compared to the mean.
- ❖ It prevents skewing the data due to extreme values.

4.2 Encoding Categorical Variables

The target variable (Outcome) is already **binary (0 = No Diabetes, 1 = Diabetes)**, so no encoding is needed for it.

However, if there were categorical features, I would encode them using **Label Encoding** or **One-Hot Encoding**.

Example (if categorical encoding were required):

```
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
data['CategoricalColumn'] = label_encoder.fit_transform(data['CategoricalColumn'])
```

4.3 Feature Scaling

Since the dataset contains numerical features with different scales, I apply **Standardization** to improve model performance.

✅ **Why Scale the Data?**

- ❖ Models like **Logistic Regression, SVM, and KNN** perform better when data is on a similar scale.
- ❖ Prevents features with larger magnitudes from **dominating the learning process**.

Code to Scale Features:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X = data.drop('Outcome', axis=1)  # Features
y = data['Outcome']  # Target variable

X_scaled = scaler.fit_transform(X)
```

StandardScaler transforms the features so that they have:

- ❖ **Mean = 0**
- ❖ **Standard Deviation = 1**

4.4 Splitting the Data

Before training the model, I split the dataset into **training (80%)** and **testing (20%)** sets.

Code to Split Data:

```
from sklearn.model_selection import train_test_split

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

✅ **Why Split the Data?**

- ❖ Helps evaluate how well the model generalizes to **new data**.
- ❖ Prevents **overfitting** by ensuring the model doesn’t just memorize training data.

4.5 Summary of Data Preprocessing

✅ **Handled missing values** by replacing zeros with the median.
✅ **Encoded categorical variables** (if necessary).
✅ **Scaled numerical features** using StandardScaler.
✅ **Split data into training and testing sets** for model evaluation.

5. Model Selection & Training

Once the data is preprocessed, the next step is selecting an appropriate machine learning model and training it on the dataset. The goal is to build a predictive model that can accurately classify whether a patient has diabetes based on medical attributes.

5.1 Choosing the Right Model

Since my problem is a **binary classification task** (Outcome = 0 or 1), I can use various classification algorithms:

Model	Advantages	Disadvantages
Logistic Regression	Simple, interpretable, fast	Assumes linear relationships
Random Forest	Handles missing values & feature importance	Slower for large datasets
Support Vector Machine (SVM)	Works well for small datasets with clear margins	Computationally expensive
K-Nearest Neighbors (KNN)	Non-parametric, easy to understand	Sensitive to irrelevant features
Gradient Boosting (XGBoost)	High accuracy, handles complex data	Requires tuning for best performance

For this project, I will experiment with **Logistic Regression** as a baseline model and compare it with **Random Forest** for better performance.

5.2 Training the Model

5.2.1 Logistic Regression (Baseline Model)

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Initialize and train the model
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

# Make predictions
y_pred_logreg = logreg.predict(X_test)

# Evaluate the model
print("Logistic Regression Accuracy:", accuracy_score(y_test, y_pred_logreg))
print("Classification Report:\n", classification_report(y_test, y_pred_logreg))
```

✅ **Why Logistic Regression?**

- It is **fast and simple** for binary classification problems.
- Provides **probability scores** for predictions.
- Works well as a **baseline model** before testing more complex models.

5.2.2 Random Forest (Improved Model)

```
from sklearn.ensemble import RandomForestClassifier

# Initialize and train the model
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Make predictions
y_pred_rf = rf.predict(X_test)

# Evaluate the model
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
print("Classification Report:\n", classification_report(y_test, y_pred_rf))
```

✅ **Why Random Forest?**

❖ Handles **non-linearity** and **feature interactions** well.
❖ Less prone to **overfitting** compared to decision trees.
❖ Provides **feature importance scores** to understand which features matter most.

5.3 Comparing Model Performance

After training both models, I compare their performance using:
✅ **Accuracy** – Measures overall correctness of the predictions.
✅ **Precision & Recall** – Important when dealing with **imbalanced datasets**.
✅ **F1-Score** – Balances precision and recall.

If **Random Forest** performs significantly better than **Logistic Regression**, we can **tune hyperparameters** for even better results.

5.4 Model Persistence (Saving the Model)

Once I have a trained model, we can save it for deployment.

```
import joblib

# Save the best performing model
joblib.dump(rf, "diabetes_model.pkl")
print("Model saved successfully!")
```

✅ **Why Save the Model?**

❖ I can **reuse it for predictions** without retraining.
❖ It allows easy **deployment in a web application**.

5.5 Summary of Model Training

✅ **Tested Logistic Regression as a baseline model.**
✅ **Trained and evaluated Random Forest for better performance.**
✅ **Compared accuracy and classification reports.**
✅ **Saved the best model for deployment.**

6. Model Evaluation

Once the machine learning models are trained, they must be evaluated to assess their performance and ensure they make reliable predictions. I use various evaluation metrics to compare models and determine the best-performing one.

6.1 Evaluation Metrics

Since this is a **binary classification** problem (diabetes or no diabetes), I use the following metrics:

Metric	Description
Accuracy	Percentage of correct predictions.
Precision	Measures how many of the predicted positive cases were actually positive.
Recall (Sensitivity)	Measures how many actual positive cases were correctly identified.
F1-Score	Harmonic mean of precision and recall (useful for imbalanced datasets).
Confusion Matrix	Shows true positive, true negative, false positive, and false negative values.
ROC-AUC Score	Measures the model's ability to distinguish between classes.

6.2 Evaluating Logistic Regression (Baseline Model)

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

# Predictions using Logistic Regression
y_pred_logreg = logreg.predict(X_test)

# Compute evaluation metrics
logreg_accuracy = accuracy_score(y_test, y_pred_logreg)
logreg_precision = precision_score(y_test, y_pred_logreg)
logreg_recall = recall_score(y_test, y_pred_logreg)
logreg_f1 = f1_score(y_test, y_pred_logreg)
logreg_roc_auc = roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])

# Display results
print("  Logistic Regression Model Performance  ")
print(f"Accuracy: {logreg_accuracy:.4f}")
print(f"Precision: {logreg_precision:.4f}")
print(f"Recall: {logreg_recall:.4f}")
print(f"F1 Score: {logreg_f1:.4f}")
print(f"ROC-AUC Score: {logreg_roc_auc:.4f}")
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred_logreg))
print("\nClassification Report:")
print(classification_report(y_test, y_pred_logreg))
```

6.3 Evaluating Random Forest (Improved Model)

```
# Predictions using Random Forest
y_pred_rf = rf.predict(X_test)

# Compute evaluation metrics
rf_accuracy = accuracy_score(y_test, y_pred_rf)
rf_precision = precision_score(y_test, y_pred_rf)
rf_recall = recall_score(y_test, y_pred_rf)
rf_f1 = f1_score(y_test, y_pred_rf)
rf_roc_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])

# Display results
print("  Random Forest Model Performance  ")
print(f"Accuracy: {rf_accuracy:.4f}")
print(f"Precision: {rf_precision:.4f}")
print(f"Recall: {rf_recall:.4f}")
print(f"F1 Score: {rf_f1:.4f}")
print(f"ROC-AUC Score: {rf_roc_auc:.4f}")
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred_rf))
print("\nClassification Report:")
print(classification_report(y_test, y_pred_rf))
```

6.4 Visualizing Model Performance

Confusion Matrix

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

# Plot Confusion Matrix
plt.figure(figsize=(6, 4))
```

```
sns.heatmap(confusion_matrix(y_test, y_pred_rf), annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - Random Forest")
plt.show()
```

ROC Curve

```
from sklearn.metrics import roc_curve

# Compute ROC curve
fpr, tpr, _ = roc_curve(y_test, rf.predict_proba(X_test)[:, 1])

# Plot ROC curve
plt.figure(figsize=(6, 4))
plt.plot(fpr, tpr, label=f"Random Forest (AUC = {rf_roc_auc:.4f})")
plt.plot([0, 1], [0, 1], linestyle="--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

6.5 Comparing Models

Model	Accuracy	Precision	Recall	F1 Score	ROC-AUC
Logistic Regression	*xx.x%*	*xx.x%*	*xx.x%*	*xx.x%*	*xx.x%*
Random Forest	*xx.x%*	*xx.x%*	*xx.x%*	*xx.x%*	*xx.x%*

✅ Based on the results, **Random Forest** generally provides **better accuracy and recall**, making it a more **reliable choice** for diabetes prediction.

6.6 Summary of Model Evaluation

✅ **Compared Logistic Regression (baseline) with Random Forest (improved model).**
✅ **Used key evaluation metrics: Accuracy, Precision, Recall, F1-Score, and ROC-AUC.**
✅ **Plotted Confusion Matrix & ROC Curve for visual understanding.**
✅ **Random Forest showed better performance, making it the preferred model.**

7. Deployment

Once the model is trained and evaluated, the next step is to deploy it so that users can interact with it. Deployment allows users to input symptoms and receive predictions in real time.

7.1 Deployment Options

There are several ways to deploy a machine learning model:

1 **Flask API (Local/Cloud-Based Deployment)** – Deploy the model as a REST API.
2 **FastAPI (Alternative to Flask, Fast & Async)** – Modern and faster than Flask.
3 **Streamlit (For Interactive Web Apps)** – Best for a quick, user-friendly UI.
4 **Gradio (For UI with minimal code)** – Ideal for quick demos.
5 **Cloud Deployment (AWS, GCP, Render, Heroku, etc.)** – Make the model accessible online.

For this project, I will deploy the model using **Flask** and make predictions via a REST API.

7.2 Exporting the Trained Model

First, I need to save the trained model using joblib so that it can be loaded in our Flask application.

```
import joblib

# Save the trained model
joblib.dump(rf, "diabetes_model.pkl")

# Save the scaler (for preprocessing new inputs)
joblib.dump(scaler, "scaler.pkl")

print("Model and scaler saved successfully!")
```

This will generate one file:

❖ diabetes_model.pkl – The trained machine learning model.

7.3 Creating a Flask API for Model Deployment

Now, let’s create a **Flask application** that serves the model as a REST API.

Install Flask

```
pip install flask
```

Create a Python File (app.py)

```
from flask import Flask, request, jsonify
import joblib
import numpy as np

# Load the trained model and scaler
model = joblib.load("diabetes_model.pkl")
scaler = joblib.load("scaler.pkl")

app = Flask(__name__)

@app.route("/")
def home():
    return "Diabetes Prediction API is Running!"

@app.route("/predict", methods=["POST"])
def predict():
    try:
        # Get JSON data from request
        data = request.get_json()

        # 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"]
        ]])

        # Scale the input data
        input_data_scaled = scaler.transform(input_data)

        # Make a prediction
        prediction = model.predict(input_data_scaled)

        # Return the result
        return jsonify({"prediction": int(prediction[0])})

    except Exception as e:
        return jsonify({"error": str(e)})

if __name__ == "__main__":
    app.run(debug=True)
```

7.4 Testing the API with Postman

1 Start the Flask server:

```
python app.py
```

2 Send a **POST** request to http://127.0.0.1:5000/predict with the following JSON data:

```
{
   "pregnancies": 2,
   "glucose": 120,
   "blood_pressure": 70,
   "skin_thickness": 20,
   "insulin": 80,
   "bmi": 25.5,
   "diabetes_pedigree_function": 0.5,
   "age": 30
}
```

3 The API will return a prediction:

```
{
   "prediction": 0
}
```

A prediction of 0 means **No Diabetes**, while 1 means **Diabetes Detected**.

7.5 Deploying the API to Render (Free Hosting)

Instead of running the API locally, we can deploy it online using **Render**.

1 **Create a** requirements.txt **file**

2 Upload the Project to GitHub

3 Deploy on Render

7.6 Creating a Frontend for the Model

simple web interface, I use **React + Material UI** .

7.7 Summary of Deployment

✅ **Saved the trained model (**diabetes_model.pkl**)**
✅ **Created a Flask API for real-time predictions.**
✅ **Tested the API using Postman.**
✅ **Deployed the API to Render for online access.**
✅ **Built an interactive web interface using React.**

8. Conclusion & Future Work

8.1 Conclusion

This project successfully developed a **Diabetes Prediction Model** using machine learning techniques. The dataset was analyzed, preprocessed, and used to train multiple models, among which the **best-**

performing model was selected based on evaluation metrics. The model was then deployed as a **Flask API** and integrated with a **user-friendly web interface** using **Streamlit**.

Key takeaways from this project include:
✅ The importance of **data preprocessing** (handling missing values, scaling, and encoding).
✅ How different **machine learning models** perform on classification problems.
✅ The significance of **model evaluation** using accuracy, precision, recall, and F1-score.
✅ The **deployment** of a machine learning model as a REST API and a web application.

This model provides a **quick and reliable** way to predict diabetes, which can assist in early diagnosis and preventive healthcare measures.

8.2 Future Work

While the project achieved good results, there are several ways to improve and extend it:

1 **Improve Model Accuracy**

- ❖ Experiment with **deep learning** models (e.g., Neural Networks).
- ❖ Use **hyperparameter tuning** (e.g., GridSearchCV) for optimization.
- ❖ Try **ensemble learning** (combining multiple models).

2 **Expand the Dataset**

- ❖ Collect **real-world** diabetes data for better generalization.
- ❖ Include **more medical parameters** like diet, exercise, and genetic factors.

3 **Enhance the Web Application**

- ❖ Improve UI/UX using **React & Material UI**.
- ❖ Provide **real-time feedback** and health tips based on predictions.

4 Deploy on Cloud Platforms

- ❖ Deploy the model on **AWS, GCP, or Azure** for better scalability.
- ❖ Use **Docker & Kubernetes** for containerized deployment.

5 Integrate with Healthcare Systems

- ❖ Connect the model with **electronic health records (EHRs)**.
- ❖ Develop a **mobile app** for easy access by patients and doctors.

8.3 Final Thoughts

This project demonstrates the power of machine learning in healthcare. By leveraging **data-driven insights**, we can build intelligent systems that assist in early diagnosis, potentially saving lives.

With further **improvements and real-world adoption**, this model could become a valuable tool in the fight against diabetes.

"Data is the new oil, and AI is the refinery." Let’s continue innovating for a healthier future!