****

**MSc in Artificial Intelligence and Machine Learning**

**CS6271 - Evolutionary Algorithms and Humanoid Robotics 2024**

**Final Project**

**Module Leader:** Conor Ryan

**Developers:** Rishi Vardhan Kankanala - 24141739,

Kedhar Eashwar Seetammagari - 24165409

**Video link:** [Diabetes prediction using GP.mp4](https://drive.google.com/file/d/1camIfdZ7pPl4CN6bykATLdaSke8E7dfE/view?usp=drive_link), [Link to YouTube video](https://youtu.be/KhGARwvOdg8https:/youtu.be/KhGARwvOdg8)

**Presentation link:**[Diabetes prediction using GP.pptx](https://docs.google.com/presentation/d/1ooQ6O32brKXhSARxZdAimWgXme7Mdhcz/edit?usp=drive_link&ouid=110514656299790498137&rtpof=true&sd=true)

## Introduction

The aim of this project is to develop a genetic programming (GP) model to predict the likelihood of diabetes in individuals based on various health-related attributes. Diabetes prediction is a critical task in healthcare, where early detection can significantly improve patient outcomes. Genetic programming, a type of evolutionary algorithm, is a suitable choice for this task as it can discover complex, non-linear patterns in data by evolving computer programs that maximize predictive accuracy.

Using a smaller version of the diabetes dataset, we trained the GP model to identify patterns associated with diabetes risk. After preprocessing key features, the model was trained over multiple generations, iterating through crossover and mutation operations to enhance accuracy.

This project provides insight into the application of genetic programming for medical data analysis and highlights its potential in handling complex classification tasks. Our findings demonstrate GP’s effectiveness as an innovative tool for enhancing decision support in healthcare settings.

## Data Exploration

The dataset used for this project is a simplified version of the widely studied diabetes dataset. It comprises a range of health and lifestyle factors critical in assessing diabetes risk, including medical indicators such as BMI, physical health metrics, and demographic factors.

The dataset is split into two main files: "train.csv" for training the model and "test.csv" for evaluating predictive performance. Each entry represents an individual, with a target label indicating their diabetes status (presence or absence of the condition). The goal is to develop a predictive model capable of generalizing to new cases, providing valuable insights into the contributing factors for diabetes risk.

The dataset features a mix of categorical and continuous variables. Binary indicators include high blood pressure, cholesterol levels, physical activity, and dietary habits, which are crucial lifestyle factors in diabetes assessment. Continuous features, such as BMI and age, allow for quantitative analysis of health risks. Together, these features provide a comprehensive view of individual health profiles, enabling the model to uncover patterns in diabetes onset across various health and demographic factors.

## Data Preprocessing

In this project, data preprocessing involved several steps to prepare the dataset for training an effective classifier. The target variable, "output," was separated from the features, and three main techniques—correlation analysis, mutual information scoring, and feature importance using a Random Forest classifier—were used to assess feature relevance.

**Correlation Analysis:** A heatmap was generated to identify highly correlated features (correlation > 0.8), helping to reduce multicollinearity and simplify the model. Features with high correlations were flagged for potential exclusion from the analysis.

**Mutual Information (MI):** The mutual\_info\_classif function was used to calculate mutual information scores for each feature, indicating their relationship with the target variable. The features were ranked based on these scores, allowing the selection of high-impact variables for the model.

**Feature Importance via Random Forest:** Using a Random Forest classifier, feature importance scores were computed to further refine the list of critical features. This method provided insights into the features with the most predictive power for diabetes risk.

Based on these analyses, certain less relevant features—such as **"CholCheck," "AnyHealthcare," "NoDocbcCost," "MentHlth," "DiffWalk," "Smoker," and "Education"**—were removed.

To prepare the data for machine learning, it was split into training and testing sets. Columns were categorized as binary, continuous, or ordinal for targeted preprocessing. Binary columns were encoded as integers, continuous features were standardized using StandardScaler, and ordinal features were passed through unaltered. A ColumnTransformer pipeline ensured consistent preprocessing across the training and testing sets. The transformed datasets were then used as inputs for the model, with final shapes confirming successful transformation.

**StandardScaler** standardizes numerical data by removing the mean and scaling to unit variance, ensuring all features have a mean of 0 and a standard deviation of 1. This is important for models sensitive to feature magnitudes.

**OneHotEncoder** – Converts binary and categorical variables (HighBP, HighChol, Stroke, HeartDiseaseorAttack, PhysActivity, Fruits, Veggies, HvyAlcoholConsump, Sex) into binary columns. This technique is crucial for handling categorical data for machine learning models since they can't process non-numeric values directly.

**ColumnTransformer** applies different preprocessing steps to specific columns. It allows customization, such as scaling continuous features, one-hot encoding categorical features, and leaving other features unchanged, all in one step. This ensures efficient and consistent preprocessing for diverse feature types. It applies:

* StandardScaler to continuous columns (BMI, PhysHlth, Age).
* OneHotEncoder to binary/categorical columns (HighBP, HighChol, etc.).
* It passes through the ordinal columns (GenHlth, Income) without any transformation, as they are already in a format that can be used directly by some models.

## Methodology

### Genetic Programming Setup

Genetic Programming was chosen due to its ability to automatically evolve symbolic expressions that can handle non-linear relationships. GP iteratively evolves expressions by mimicking the process of natural selection, retaining better solutions and modifying them over time. The DEAP library was employed to implement GP due to its robustness and flexibility for evolving symbolic expressions.

**Problem Representation:** The dataset is preprocessed, and features are transformed using StandardScaler, OneHotEncoder, and ColumnTransformer. This results in a transformed dataset X\_train\_transformed and X\_test\_transformed, which serve as inputs to the GP model. The target variable y\_train is the classification label (binary outcome) used for training.

**Defining Fitness Function:** The goal of the GP is to maximize accuracy, which is used as the fitness function. The fitness function evaluates the accuracy of the predictions made by a GP-individual expression (a mathematical model) against the actual labels.

**Creating the GP Primitive Set:** The GP model is built using a primitive set (pset), which includes mathematical functions like addition, subtraction, multiplication, division (with checks to avoid division by zero), and common functions like log, sqrt, and exp. Additionally, trigonometric functions such as sin, cos, and tan are included. Random constants are also incorporated to introduce randomness into the expressions.

**Initialization of Individuals:** An individual in GP is represented as a tree of mathematical functions, where each node corresponds to a mathematical operator, and the leaves correspond to the input features or constants. The population is initialized using the Half and Half method to generate a diverse set of expressions.

**Genetic Operators:** The GP algorithm employs genetic operators such as crossover and mutation to evolve the population of individuals:

**Crossover** exchanges genetic material between two individuals (parent trees) to create offspring.

**Mutation** introduces small changes to individual trees to enhance diversity and avoid local minima. These operators are applied iteratively with a certain probability (crossover\_prob and mutation\_prob).

**Selection and Evolution:** At each generation, the population is evaluated using the tournament selection method, where individuals are selected based on their fitness scores. The selected individuals undergo crossover and mutation to generate the next generation. The fitness of offspring is recalculated, and the population is replaced with the newly evolved individuals. This process repeats for a set number of generations (num\_generations).

**Fitness Evaluation:** The fitness evaluation function applies the GP expression to the training data to make predictions. These predictions are compared with the actual labels (y\_train), and the accuracy score is calculated as the fitness value.

**Final Model Selection:** After the completion of the evolutionary process, the best individual (the one with the highest fitness value) is selected for making predictions. This individual represents the optimized mathematical expression that can be used as a model for classification.

**Testing:** The selected model is then applied to the transformed test data (X\_test\_transformed). Predictions are generated and converted into binary labels (0 or 1) using a threshold of 0.5.

**SETUP 1:**

# Define fitness function

creator.create("FitnessMax", base.Fitness, weights=(1.0,))

creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMax)

def safe\_divide(a, b):

return a / b if b != 0 else 1

def safe\_log(x):

return np.log(x) if x > 0 else 1

def safe\_sqrt(x):

return np.sqrt(x) if x > 0 else 1

def safe\_exp(x):

return np.exp(x) if x > 0 else 1

# Configure primitive set

pset = gp.PrimitiveSet("MAIN", X\_train\_transformed.shape[1])

pset.addPrimitive(operator.add, 2)

pset.addPrimitive(operator.sub, 2)

pset.addPrimitive(operator.mul, 2)

pset.addPrimitive(safe\_divide, 2)

pset.addPrimitive(operator.neg, 1)

pset.addPrimitive(operator.abs, 1)

pset.addPrimitive(safe\_log, 1)

pset.addPrimitive(safe\_sqrt, 1)

pset.addPrimitive(safe\_exp, 1)

pset.addPrimitive(np.sin,1)

pset.addPrimitive(np.cos,1)

pset.addPrimitive(np.tan,1)

pset.addEphemeralConstant("rand\_const", lambda: random.random())

# Add terminal set (input features)

for i in range(X\_train\_transformed.shape[1]):

pset.renameArguments(\*\*{f'ARG{i}': f'feature\_{i}'})

# Define toolbox and register functions

toolbox = base.Toolbox()

toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min\_=1, max\_=2)

toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)

toolbox.register("population", tools.initRepeat, list, toolbox.individual)

toolbox.register("compile", gp.compile, pset=pset)

# Define evaluation function

def evaluate\_individual(individual):

# Transform the individual into a callable function

func = toolbox.compile(expr=individual)

# Predict output using GP expression

predictions = [int(func(\*x) > 0.5) for x in X\_train\_transformed.values]

return accuracy\_score(y\_train, predictions),

toolbox.register("evaluate", evaluate\_individual)

toolbox.register("select", tools.selTournament, tournsize=3)

toolbox.register("mate", gp.cxOnePoint)

toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr, pset=pset)

toolbox.register("expr\_mut", gp.genFull, min\_=0, max\_=2)

# Genetic programming hyperparameters

population\_size = 100

num\_generations = 100

crossover\_prob = 0.7

mutation\_prob = 0.3

# Create initial population

population = toolbox.population(n=population\_size)

# Run the GP evolution

for gen in range(num\_generations):

offspring = toolbox.select(population, len(population))

offspring = list(map(toolbox.clone, offspring))

# Apply crossover and mutation

for child1, child2 in zip(offspring[::2], offspring[1::2]):

if random.random() < crossover\_prob:

toolbox.mate(child1, child2)

del child1.fitness.values

del child2.fitness.values

for mutant in offspring:

if random.random() < mutation\_prob:

toolbox.mutate(mutant)

del mutant.fitness.values

# Evaluate the fitness of individuals with invalid fitness

invalid\_ind = [ind for ind in offspring if not ind.fitness.valid]

fitnesses = map(toolbox.evaluate, invalid\_ind)

for ind, fit in zip(invalid\_ind, fitnesses):

ind.fitness.values = fit

# Replace population with offspring

population[:] = offspring

# Gather and print stats

fits = [ind.fitness.values[0] for ind in population]

best\_ind = tools.selBest(population, 1)[0]

print(f"Generation {gen}: Best accuracy = {best\_ind.fitness.values[0]}")

# Compile best individual

best\_func = toolbox.compile(expr=best\_ind)

# Generate predictions for the test set

test\_predictions = [int(best\_func(\*x) > 0.5) for x in X\_test\_transformed.values]

**Kaggle Accuracy: 0.67906**

**SETUP 2:**

The crossover and mutation probabilities were changed to explore results

crossover\_prob = 0.8

mutation\_prob = 0.2

**Kaggle Accuracy: 0.67449**

**SETUP 3: (Best setup)**

crossover\_prob = 0.6

mutation\_prob = 0.4

**Kaggle Accuracy: 0.68492**

## Results and Discussion

The genetic programming (GP) model achieved a training accuracy of approximately 72.59% on the subset of the diabetes dataset. This suggests that the GP model was able to identify certain patterns in the data that correlate with diabetes risk, but there is still room for improvement in terms of predictive power. The model was able to effectively evolve symbolic expressions from the dataset, leveraging a mix of mathematical and trigonometric operations to predict the likelihood of diabetes.

The Kaggle test accuracy, which varied across different experimental setups, ranged from 0.67449 to 0.68492, with the best setup achieving an accuracy of 68.49%. This indicates that the GP model has potential, but its performance on unseen data was slightly lower than on the training data, as expected due to overfitting, which is a common challenge in machine learning models. The setup with a balanced crossover and mutation probability (crossover\_prob = 0.6, mutation\_prob = 0.4) produced the best test accuracy, suggesting that this configuration helped achieve a good trade-off between exploration and exploitation during the evolutionary process.

## Conclusion and Future Scope

This study demonstrates the potential of Genetic Programming (GP) as a tool for training classifiers on complex datasets like diabetes prediction. The GP model showed that symbolic expressions can be evolved to capture underlying patterns in the data, offering a promising alternative to traditional machine learning algorithms. However, the model's performance indicates that further improvements are needed to enhance both its accuracy and generalization capabilities.

In terms of future work, one promising direction is the integration of more advanced evolutionary strategies, such as multi-objective optimization, to balance accuracy with model simplicity. Additionally, exploring alternative fitness functions could help improve the evolutionary search process, potentially leading to more robust solutions. The incorporation of ensemble learning techniques, where multiple GP models are combined, may also enhance the overall performance and reduce overfitting.

Furthermore, applying this approach to other domains and datasets, especially those with higher-dimensional data or more complex patterns, will be crucial in assessing the scalability and adaptability of GP. In addition, exploring hybrid models combining GP with other machine learning techniques, such as deep learning, could offer further insights into improving predictive accuracy for real-world applications. Overall, Genetic Programming holds significant potential, and its future development could make it a powerful tool in the field of data science and predictive analytics.