FINAL PROJECT:

Title: Machine Learning for Genetic Data Analysis

Introduction:

Machine learning in genetic data analysis merges computer science with biology. It's about teaching computers to spot patterns and connections in vast amounts of genetic information. By doing this, scientists can understand how genes influence traits and diseases. This technology helps predict risks for diseases, uncover genetic relationships, and tailor treatments based on individual genetic differences. Ultimately, it holds the potential to revolutionize healthcare by personalizing medicine to suit each person's genetic makeup.

When scientists use machine learning to analyze genetic data, they are essentially asking computers to assist them in determining how certain genes may affect our health. It's a detective job, attempting to connect specific genes to specific traits or diseases. Based on their genes, this can help predict if someone is more likely to get sick. It all comes down to using technology to unlock the secrets hidden in our genes, allowing doctors to provide better, more personalized care to patients based on their unique genetic makeup.

In our project we are going to develop a machine learning framework which can process a large amount of genetic data to identify patterns and mutations which are associated with specific diseases.

Summary of our project:

This code serves as a pipeline for analyzing genetic marker data. Initially, it imports a dataset containing genetic marker information, organizing it into features and a target variable. After splitting the data into training and validation subsets, it employs a Random Forest Regressor, a machine learning algorithm, to learn patterns and relationships within the training data.

The model then utilizes this learned information to predict disease likelihood based on the genetic markers in the validation set. Subsequently, it evaluates the model's predictive accuracy using the Mean Squared Error metric, quantifying the disparity between predicted and actual values. Overall, this workflow showcases how machine learning techniques can be applied to genetic data to predict disease

Getting to Know Our Dataset:

The dataset that is being provided seems to include both target variable 'Disease Likelihood' and genetic marker data. The terms are broken down as follows:

Markers (Marker_1 to Marker_10):

These columns represent different genetic markers or features obtained from a study or analysis. Each marker has numerical values associated with it.

Disease_Likelihood:

Based on genetic markers, the target variable in this column indicates the likelihood or probability of a disease occurring.

It is a continuous value that indicates the likelihood of disease or a score that indicates disease susceptibility.

The dataset contains approximately 300 rows of data, with each row representing a sample or individual and the columns (markers) containing numerical values related to that sample. The markers' values appear to be continuous and range between 0 and 1.

This data contains information about ten different genetic markers (like flags that indicate certain traits) as well as a column that indicates the likelihood of a disease. The numbers in each row indicate whether these markers are active or inactive. We'll look at how these numbers relate to the disease likelihood score to determine how important each marker is for predicting the disease. Understanding what these markers mean in terms of the disease they predict will help us understand their significance.

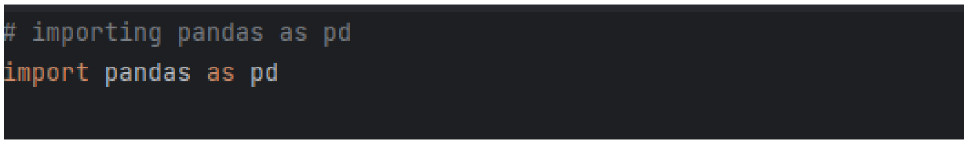
Step by Step Process :

LOADING OUR DATASETS IN PYCHARM:

Our First Step is to import necessary libraries like "panda"

"import pandas": This keyword is used to import the pandas library, which is a powerful tool for data manipulation and analysis in Python.

"pd". This alias is commonly used to reference pandas functions and objects throughout the code.



```
# importing pandas as pd
import pandas as pd
```

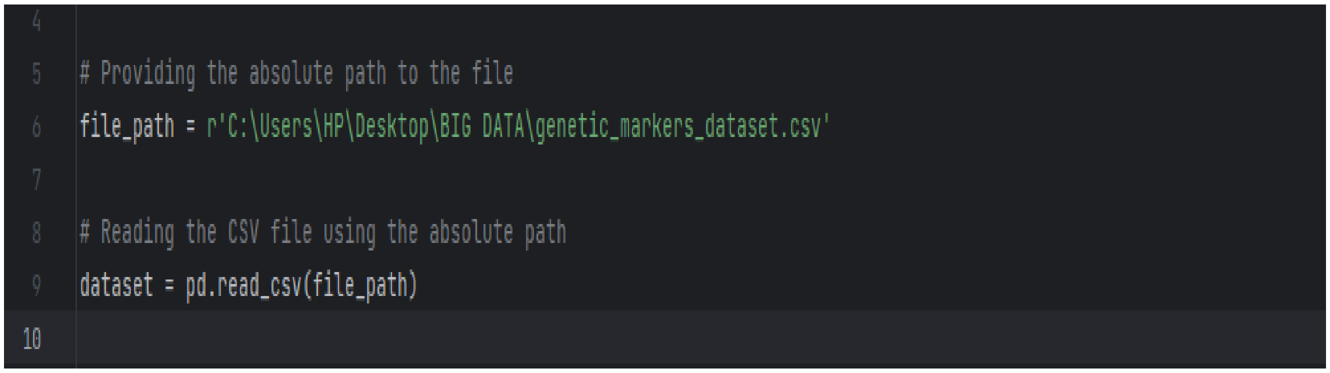
We have two path to upload a file one is absolute and other one is relative

Absolute path:

An absolute path refers to the complete location of a file or directory in a file system starting from the root directory. It specifies the file or directory location regardless of the current working directory. For instance, in Windows, C:\Users\HP\Desktop\file.txt is an absolute path.

Relative path:

A relative path, on the other hand, specifies the location of a file or directory relative to the current working directory. It does not start from the root directory but rather from the current location. For instance, if the current directory is C:\Users\HP, then a relative path to file.txt located in the Desktop directory would be Desktop\file.txt.



```
# Providing the absolute path to the file
file_path = r'C:\Users\HP\Desktop\BIG DATA\genetic_markers_dataset.csv'

# Reading the CSV file using the absolute path
dataset = pd.read_csv(file_path)

```

This line creates a variable named file path and assigns it a string value representing the absolute path to a CSV file named genetic_markers_dataset.csv. The r before the string indicates a raw string literal in Python, used particularly for file paths to interpret backslashes \ as literal characters rather than escape characters.

```
# Reading the CSV file using the absolute path
dataset = pd.read_csv(file_path)
```

Here, the pd.read_csv() function from the pandas library (pd) is used to read the CSV file specified by the file_path. The function reads the contents of the CSV file and creates a pandas DataFrame named dataset. This DataFrame holds the data from the CSV file, allowing for easy manipulation, analysis, and exploration of the data using pandas' functionalities.

#checking the dataset

```
print(dataset)
```

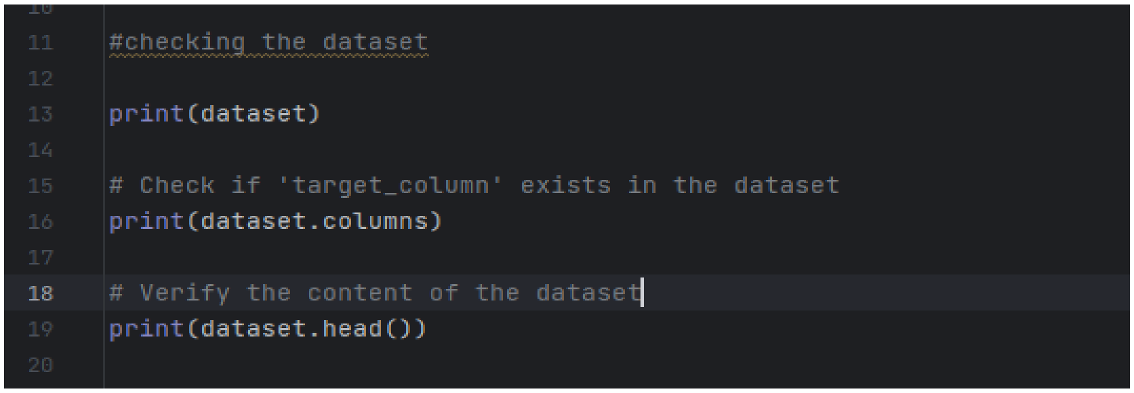
This code prints the entire contents of the dataset, displaying all the rows and columns contained within the DataFrame named dataset. It provides an overview of the data to the user.

```
# Check if 'target_column' exists in the dataset
print(dataset.columns)
```

This line prints the column names of the dataset. The .columns attribute of a Pandas DataFrame retrieves the column labels. It helps to confirm the presence of a specific column, denoted by 'target_column', in the dataset.

```
# Verify the content of the dataset
print(dataset.head())
```

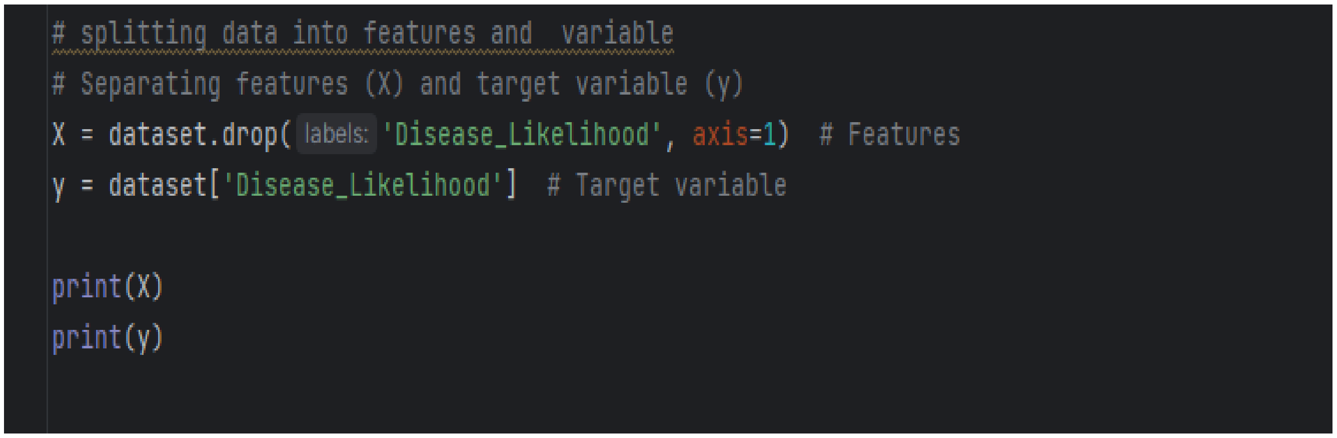
This code prints the first few rows (by default, the first five rows) of the dataset using the head() function. It allows a quick glimpse into the structure and content of the dataset, aiding in initial inspection and understanding.



```
#checking the dataset

print(dataset)

# Check if 'target_column' exists in the dataset
print(dataset.columns)

# Verify the content of the dataset
print(dataset.head())
```

We are splitting our data into features and variables

```
# splitting data into features and  variable
# Separating features (X) and target variable (y)
X = dataset.drop('Disease_Likelihood', axis=1)  # Features
y = dataset['Disease_Likelihood']  # Target variable
```



```
# splitting data into features and  variable
# Separating features (X) and target variable (y)
X = dataset.drop( labels: 'Disease_Likelihood', axis=1)  # Features
y = dataset['Disease_Likelihood']  # Target variable

print(X)
print(y)
```

Features (X): These are the columns or attributes used as input to predict the target variable. In this code, X represents all the columns in the dataset except Disease_Likelihood. Each column in X is considered a feature that might influence or contribute to predicting the likelihood of a disease.

Target Variable (y): This is the variable we're trying to predict or understand. In this case, it's Disease_Likelihood. y contains the values of this particular column, and typically, in a supervised learning setup, you'd use features (X) to predict the target variable (y).

X = dataset.drop('Disease_Likelihood', axis=1):

This line creates a new DataFrame X containing the features. It removes the column named 'Disease_Likelihood' along the columns (axis=1) from the dataset, resulting in X containing all columns except the target variable.

y = dataset['Disease_Likelihood']: This line creates a Series y containing only the target variable 'Disease_Likelihood' from the dataset. This will be the variable to be predicted or modeled.

We can print this by using

```
print(X)
print(y)
```

Model Training and validation:

We are going to use train_test-split function :

The train_test_split function in scikit-learn is like dividing a box of toys into two parts: one part to learn from (training) and the other to test how well we learned (validation). It randomly separates the data into training features and target values (X_train, y_train) and validation features and target values (X_val, y_val) based on a set proportion like 70% for training and 30% for validation. This helps ensure that our model learns from one set and gets tested on another, unseen set, helping us check how well it predicts new information.

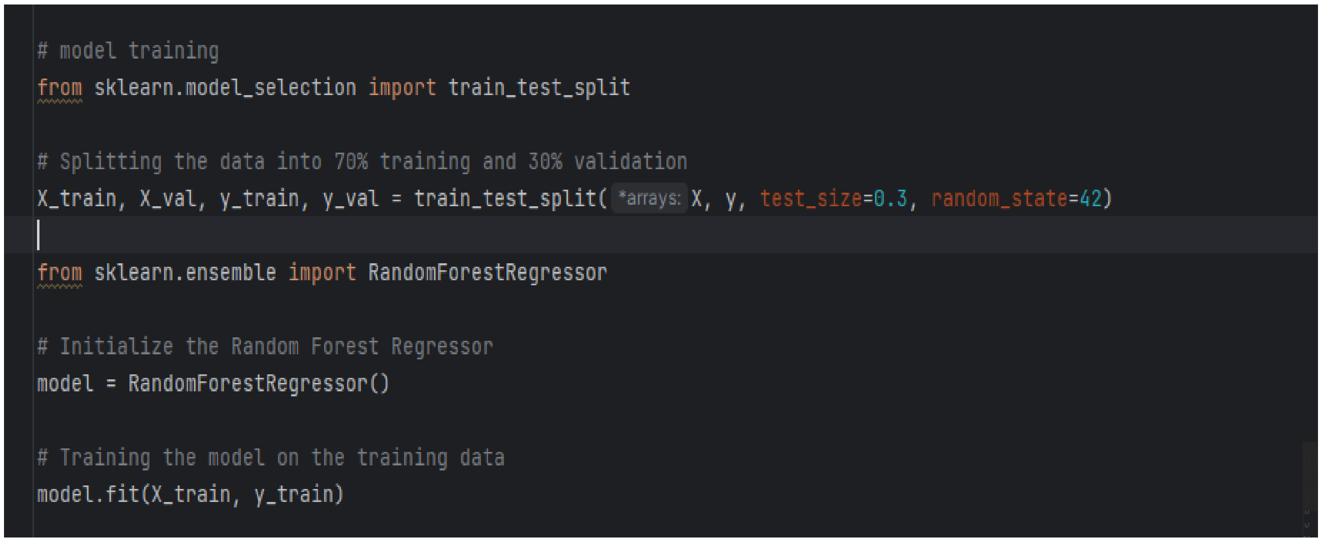
```
# model training
from sklearn.model_selection import train_test_split

# Splitting the data into 70% training and 30% validation
```

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, random_state=42)

from sklearn.ensemble import RandomForestRegressor

Initialize the Random Forest Regressor
model = RandomForestRegressor()

Training the model on the training data
model.fit(X_train, y_train)



```
# model training
from sklearn.model_selection import train_test_split

# Splitting the data into 70% training and 30% validation
X_train, X_val, y_train, y_val = train_test_split( *arrays: X, y, test_size=0.3, random_state=42)

from sklearn.ensemble import RandomForestRegressor

# Initialize the Random Forest Regressor
model = RandomForestRegressor()

# Training the model on the training data
model.fit(X_train, y_train)
```

Import Libraries:

from sklearn.model_selection import train_test_split: Imports a function that helps split the dataset into training and validation sets.

Splitting Data:

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, random_state=42): Splits the dataset into training and validation sets.

X contains the features.

y contains the target variable.

test_size=0.3 specifies that 30% of the data will be used for validation.

random_state=42 ensures reproducibility, making sure that the data split remains the same when you run this code again.

Import Model:

from sklearn.ensemble import RandomForestRegressor: Imports the Random Forest Regressor model from scikit-learn, a popular machine learning library.

Initialize Model:

model = RandomForestRegressor(): Creates an instance of the Random Forest Regressor model.

Model Training:

model.fit(X_train, y_train): Trains the Random Forest Regressor model using the training data (X_train and y_train). The model learns patterns and relationships between features (X_train) and the target variable (y_train). This is where the model adjusts its internal parameters to make predictions based on the training data.

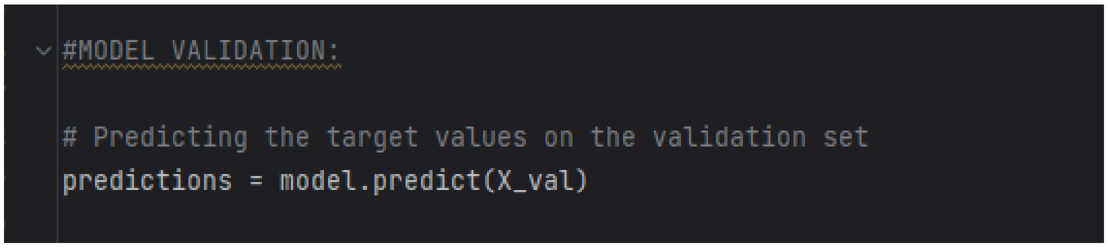
After this code runs, the model variable contains a trained Random Forest Regressor that can be used to make predictions on new data.

#MODEL VALIDATION:

```
# Predicting the target values on the validation set
predictions = model.predict(X_val)
```

predictions = model.predict(X_val):

This line uses the trained model to predict the target values (y_val) based on the features in the validation set (X_val). The predicted values are stored in the prediction's variable for further evaluation or comparison with the actual validation set values to assess the model's performance.

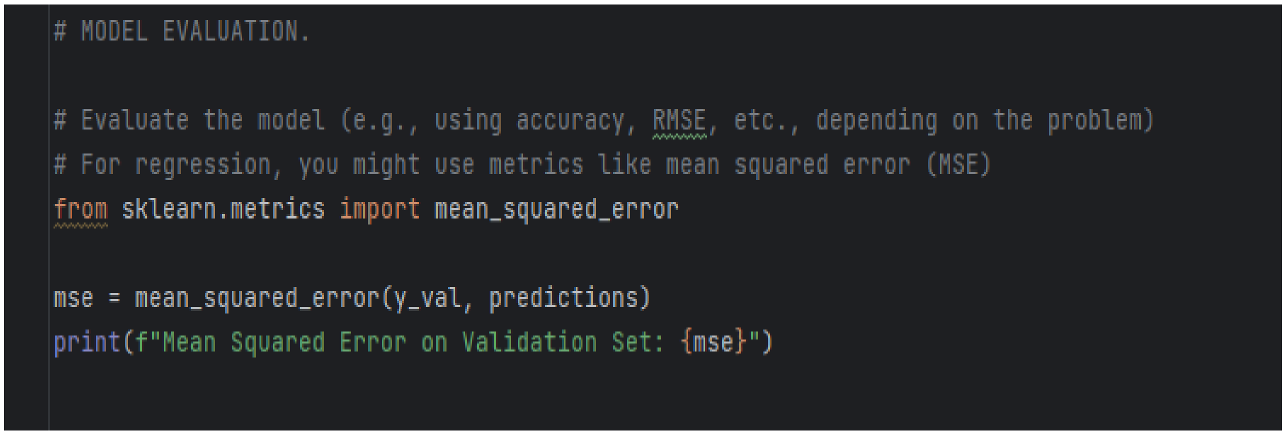


```
#MODEL VALIDATION:

# Predicting the target values on the validation set
predictions = model.predict(X_val)
```

# MODEL EVALUATION.

# Evaluate the model (e.g., using accuracy, RMSE, etc., depending on the problem)
# For regression, you might use metrics like mean squared error (MSE)
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_val, predictions)
print(f"Mean Squared Error on Validation Set: {mse}")



```
# MODEL EVALUATION.

# Evaluate the model (e.g., using accuracy, RMSE, etc., depending on the problem)
# For regression, you might use metrics like mean squared error (MSE)
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_val, predictions)
print(f"Mean Squared Error on Validation Set: {mse}")
```

MSE(mean_squared_error):

For me the simple understanding of mse is the squares of the difference between the predicted valuea and the actual values. It is a common metric used in regression problems to assess how well the model prediction align with the true value

from sklearn.metrics import mean_squared_error:

This line imports the mean_squared_error function from the sklearn.metrics module. This function is used to calculate the mean squared error, a commonly used metric for evaluating regression models.

mse = mean_squared_error(y_val, predictions):

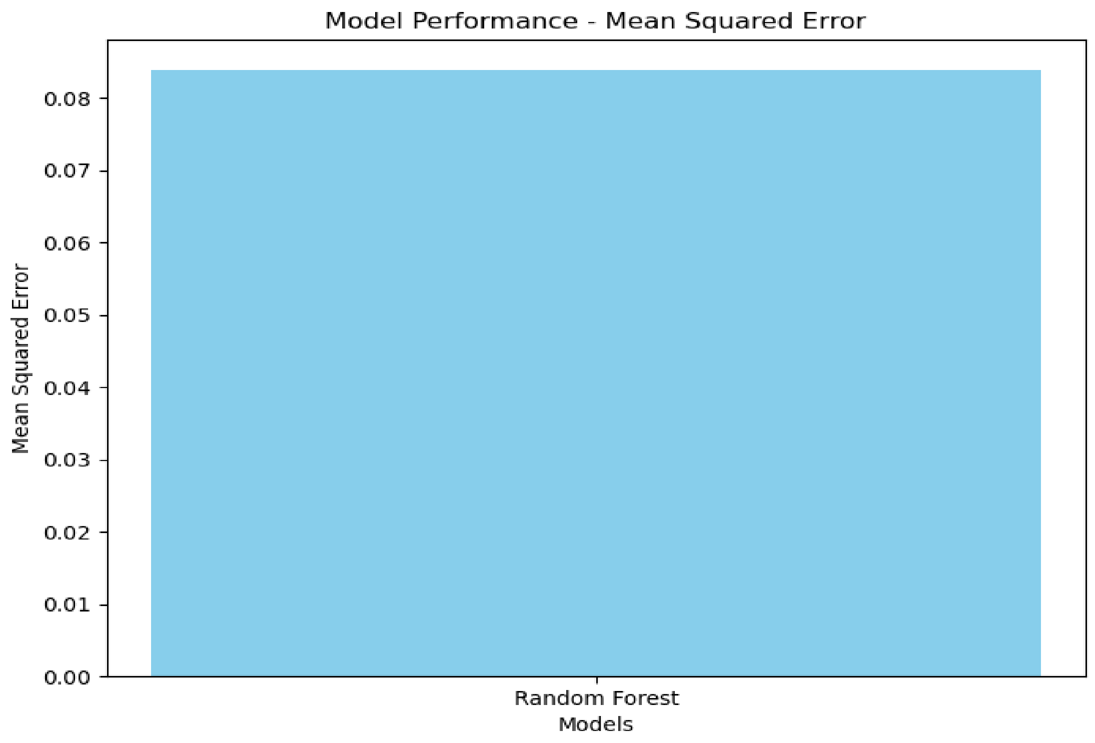
Here, the mean_squared_error function is used to compute the mean squared error by comparing the actual target values (y_val) from the validation set with the predicted values (predictions) generated by the model. The calculated MSE value represents the average squared difference between predicted and actual values.

print(f"Mean Squared Error on Validation Set: {mse}"):

This line prints out the computed mean squared error value. It provides insight into how well the model predictions align with the actual values in the validation set. In regression tasks, a lower MSE implies that the model's predictions are closer to the actual values, indicating better performance.

Mean Squared Error on Validation Set: 0.08719870332954016

A mean squared error (MSE) of 0.0872 suggests that, on average, the squared difference between the predicted values generated by the model and the actual target values in the validation set is approximately 0.0872. In regression tasks, a lower MSE indicates that the model's predictions are closer to the actual values, which generally signifies better performance



This graphical representation provides a visual understanding of the model's performance in predicting the target variable. Lower bars represent lower MSE values, signifying better performance in predicting the target variable

Higher bars, on the other hand, represent higher MSE values, indicating potentially poorer prediction performance. This plot is useful for comparing the predictive accuracy of different models or different versions of the same model.

Deployment of our project:

```
library(shiny)

library(randomForest)

library(caret)

library(ggplot2)

library(sparklyr)


# Initialize Spark connection

sc <- spark_connect(master = "local")


# Read data

filePath <- "genetic_markers_dataset.csv"

df <- spark_read_csv(sc, "df", filePath, header = TRUE, infer_schema = TRUE)


# Function to manually scale a column

scale_column <- function(df, column_name) {

  mean_val <- df %>% summarize(mean = mean(!!sym(column_name))) %>% collect() %>%
.[["mean"]]

  sd_val <- df %>% summarize(sd = sd(!!sym(column_name))) %>% collect() %>% .[["sd"]]


  scaled_column_name <- paste0(column_name, "_scaled")

  df %>% mutate(!!scaled_column_name := (!!sym(column_name) - mean_val) / sd_val)

}


# UI definition
```

```
ui <- fluidPage(
  titlePanel("Disease Likelihood Prediction with Random Forest"),
  sidebarLayout(
    sidebarPanel(
      fileInput('datafile', 'Upload CSV File', accept = c(".csv")),
      actionButton("train", "Train Model")
    ),
    mainPanel(
      DT::dataTableOutput('tableData')
    )
  )
)

# Server logic
server <- function(input, output, session) {
  observeEvent(input$goButton, {
    req(input$file1)

    trainedModel <- eventReactive(input$train, {
      data <- dataInput()
      if (is.null(data)) return(NULL)

      # Data scaling
      scaled_data <- scale_column(data, "feature_column_name")

      # Splitting data into features and target variable
      X <- subset(scaled_data, select = -Disease_Likelihood)
      y <- scaled_data$Disease_Likelihood

      # Model training logic
      set.seed(42)
      trainIndex <- caret::createDataPartition(y, p = 0.7, list = FALSE)
      X_train <- X[trainIndex, ]
      y_train <- y[trainIndex]
      X_val <- X[-trainIndex, ]
      y_val <- y[-trainIndex]

      model <- randomForest(x = X_train, y = y_train)
      saveRDS(model, "model.rds")
      list(model = model, X_val = X_val, y_val = y_val)
    })

    output$tableData <- DT::renderDataTable({
      data <- dataInput()
      if (is.null(data)) return()
      data
    })
  })
}

# Run the application
shinyApp(ui, server)
```

A brief understanding of deployment.

This code is for a Shiny web application that predicts disease likelihood based on genetic markers using a Random Forest model. Here's a simple breakdown:

Libraries: It starts by loading various libraries like shiny, randomForest, caret, ggplot2, and sparklyr.

Spark Connection: It initializes a connection to a local Spark instance (spark_connect) and reads a dataset (genetic_markers_dataset.csv) into a Spark DataFrame (df) using spark_read_csv.

Scaling Function: There's a function defined (scale_column) to manually scale a specified column in the dataset by subtracting the mean and dividing by the standard deviation.

UI Definition: It defines the user interface for the Shiny app. It has a title panel, a sidebar with an option to upload a CSV file and a button to train the model, and a main panel to display data in a table format.

Server Logic: The server function contains the core logic of the app.

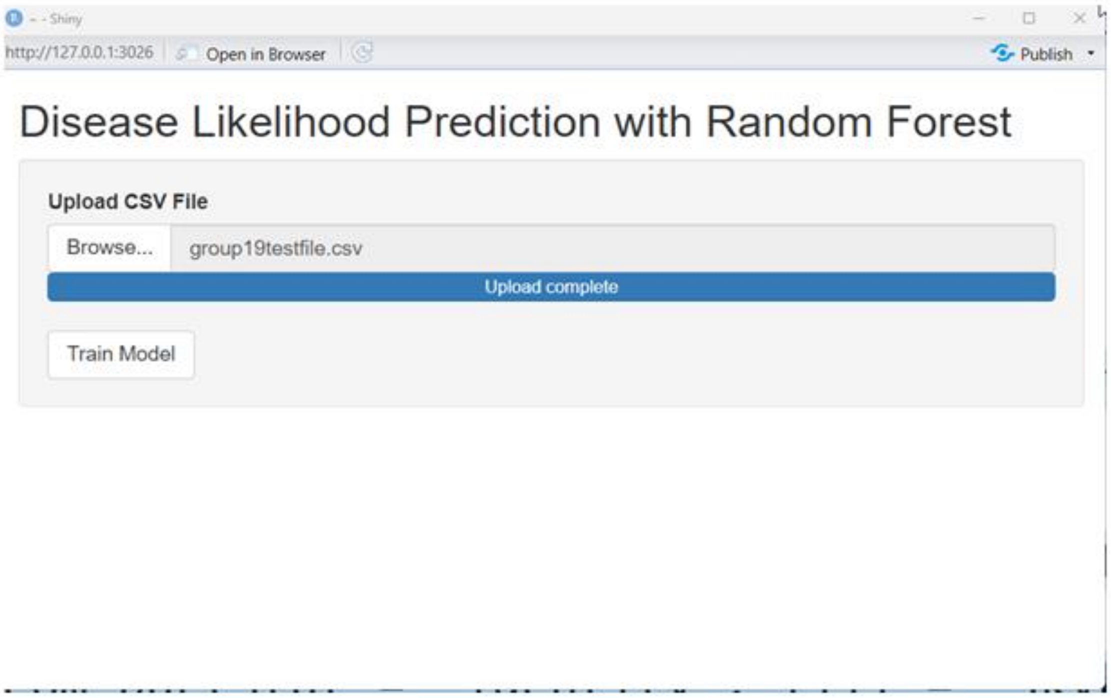
It observes the "train" button click and performs actions when triggered.

When the "train" button is clicked, it reads the uploaded data, scales it using the scale_column function, splits it into features and target variable, trains a Random Forest model using randomForest, saves the trained model as model.rds, and stores validation data.

The resulting trained model and data are stored as a reactive object trainedModel.

It also renders the uploaded data in a table using renderDataTable.

Shiny App Run: Finally, the shinyApp function runs the Shiny application, combining the UI and server logic.



Disease Likelihood Prediction with Random Forest

Upload CSV File