Supervised Learning Exercise: Data QC, Exploration, and Model Training

Objective: Understand the dataset's characteristics, clean it, explore its features, and train a basic supervised machine learning model.

Part 1: Data Quality Check and Exploration

```r
# 1. Define the list of required packages.
# We include packages for data manipulation, plotting, and all the machine learning models.
required_packages <- c(
  "tidyverse",    # For data manipulation and plotting (includes ggplot2, dplyr)
  "e1071",        # For the SVM model
  "corrplot",     # For visualizing correlation matrices
  "GGally",       # For creating pair plots
  "rpart",        # For fitting decision tree models
  "rpart.plot",   # For visualizing decision trees
  "randomForest", # For fitting random forest models
  "pROC",         # For creating and visualizing ROC curves
  "caret",        # For a streamlined machine learning workflow (tuning, CV)
  "xgboost"       # For the powerful gradient boosting model
)

# 2. Identify any packages that are not yet installed.
# The code checks your currently installed packages against our required list.
missing_packages <- required_packages[!(required_packages %in% installed.packages()[,"Package"])]

# 3. Install the missing packages.
# If the list of missing packages is not empty, this code will install them.
if(length(missing_packages) > 0) {
  cat("The following packages are missing and will be installed:\n")
  print(missing_packages)
  install.packages(missing_packages)
}

# 4. Load all the required packages for this session.
# We use `lapply` to apply the `library()` function to each package in our list.
# `suppressPackageStartupMessages` is used to keep the console output clean.
cat("\nLoading all required packages...\n")
```

0. Load libraries and data

```
##
## Loading all required packages...
```

```r
suppressPackageStartupMessages({
  lapply(required_packages, library, character.only = TRUE)
})
```

```
## [[1]]
##  [1] "lubridate" "forcats"   "stringr"   "dplyr"     "purrr"     "readr"
##  [7] "tidyr"     "tibble"    "ggplot2"   "tidyverse" "stats"     "graphics"
## [13] "grDevices" "utils"     "datasets"  "methods"   "base"
##
## [[2]]
##  [1] "e1071"     "lubridate" "forcats"   "stringr"   "dplyr"     "purrr"
##  [7] "readr"     "tidyr"     "tibble"    "ggplot2"   "tidyverse" "stats"
## [13] "graphics"  "grDevices" "utils"     "datasets"  "methods"   "base"
##
## [[3]]
##  [1] "corrplot"  "e1071"     "lubridate" "forcats"   "stringr"   "dplyr"
##  [7] "purrr"     "readr"     "tidyr"     "tibble"    "ggplot2"   "tidyverse"
## [13] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "methods"
## [19] "base"
##
## [[4]]
##  [1] "GGally"    "corrplot"  "e1071"     "lubridate" "forcats"   "stringr"
##  [7] "dplyr"     "purrr"     "readr"     "tidyr"     "tibble"    "ggplot2"
## [13] "tidyverse" "stats"     "graphics"  "grDevices" "utils"     "datasets"
## [19] "methods"   "base"
##
## [[5]]
##  [1] "rpart"     "GGally"    "corrplot"  "e1071"     "lubridate" "forcats"
##  [7] "stringr"   "dplyr"     "purrr"     "readr"     "tidyr"     "tibble"
## [13] "ggplot2"   "tidyverse" "stats"     "graphics"  "grDevices" "utils"
## [19] "datasets"  "methods"   "base"
##
## [[6]]
##  [1] "rpart.plot" "rpart"      "GGally"     "corrplot"   "e1071"
##  [6] "lubridate"  "forcats"    "stringr"    "dplyr"      "purrr"
## [11] "readr"      "tidyr"      "tibble"     "ggplot2"    "tidyverse"
## [16] "stats"      "graphics"   "grDevices"  "utils"      "datasets"
## [21] "methods"    "base"
##
## [[7]]
##  [1] "randomForest" "rpart.plot"   "rpart"        "GGally"       "corrplot"
##  [6] "e1071"        "lubridate"    "forcats"      "stringr"      "dplyr"
## [11] "purrr"        "readr"        "tidyr"        "tibble"       "ggplot2"
## [16] "tidyverse"    "stats"        "graphics"     "grDevices"    "utils"
## [21] "datasets"     "methods"      "base"
##
## [[8]]
##  [1] "pROC"         "randomForest" "rpart.plot"   "rpart"        "GGally"
##  [6] "corrplot"     "e1071"        "lubridate"    "forcats"      "stringr"
## [11] "dplyr"        "purrr"        "readr"        "tidyr"        "tibble"
## [16] "ggplot2"      "tidyverse"    "stats"        "graphics"     "grDevices"
## [21] "utils"        "datasets"     "methods"      "base"
##
## [[9]]
##  [1] "caret"        "lattice"      "pROC"         "randomForest" "rpart.plot"
##  [6] "rpart"        "GGally"       "corrplot"     "e1071"        "lubridate"
## [11] "forcats"      "stringr"      "dplyr"        "purrr"        "readr"
## [16] "tidyr"        "tibble"       "ggplot2"      "tidyverse"    "stats"
## [21] "graphics"     "grDevices"    "utils"        "datasets"     "methods"
## [26] "base"
##
## [[10]]
##  [1] "xgboost"      "caret"        "lattice"      "pROC"         "randomForest"
##  [6] "rpart.plot"   "rpart"        "GGally"       "corrplot"     "e1071"
## [11] "lubridate"    "forcats"      "stringr"      "dplyr"        "purrr"
## [16] "readr"        "tidyr"        "tibble"       "ggplot2"      "tidyverse"
## [21] "stats"        "graphics"     "grDevices"    "utils"        "datasets"
## [26] "methods"      "base"
```

```
cat("Setup complete. All packages are loaded.\n\n")
```

```
## Setup complete. All packages are loaded.
```

```
df <- read_csv("dataset_24082023.csv")
```

```
## Rows: 10000 Columns: 12
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr   (1): sex
## dbl  (10): subject_id, age, weight, height, cholesterol, blood_pressure, cal...
## date  (1): checkup_date
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
head(df)
```

```
## # A tibble: 6 x 12
##   subject_id   age sex   weight height cholesterol blood_pressure calorie_intake
##        <dbl> <dbl> <chr>  <dbl>  <dbl>       <dbl>          <dbl>          <dbl>
## 1          1    64 F       78.8   169.        206.          100.           2336.
## 2          2    67 M       68.6   173.        188.          106.           1794.
## 3          3    73 F       45.6   166.         NA           103.           1672.
## 4          4    20 M       59.9   166.        204.           69.5          1979.
## 5          5    23 M       87.7   169.        211.          103.           2135.
## 6          6    79 F       74.5   171.        199.          124.           1918.
## # i 4 more variables: exercise_frequency <dbl>, diabetes <dbl>,
## #   checkup_date <date>, bmi <dbl>
```

```
colnames(df)
```

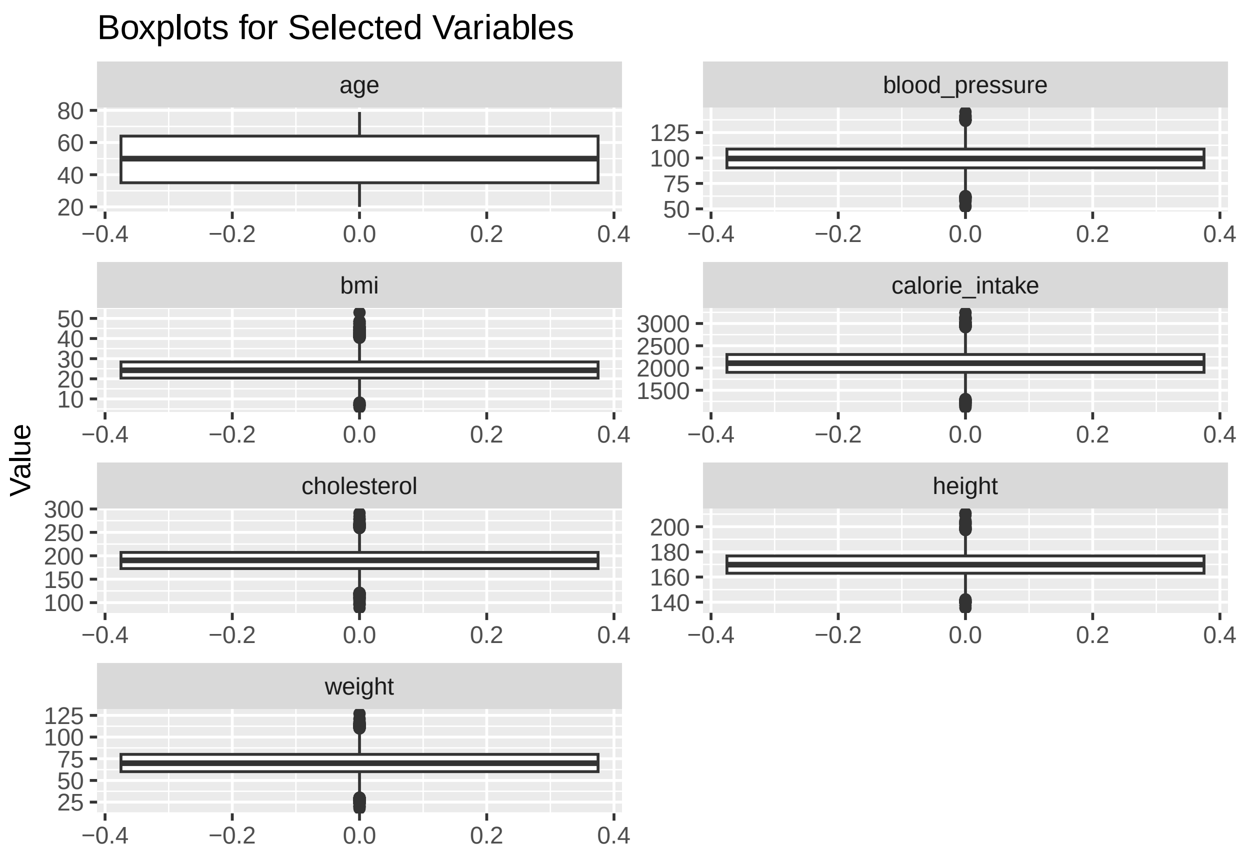
```
##  [1] "subject_id"         "age"                "sex"
##  [4] "weight"             "height"             "cholesterol"
##  [7] "blood_pressure"     "calorie_intake"     "exercise_frequency"
## [10] "diabetes"           "checkup_date"       "bmi"
```

1. Detect Outliers Outliers are data points that deviate significantly from other observations. They can arise due to variability in the data or errors. Handling outliers is crucial as they can skew the results and decrease the model's accuracy.

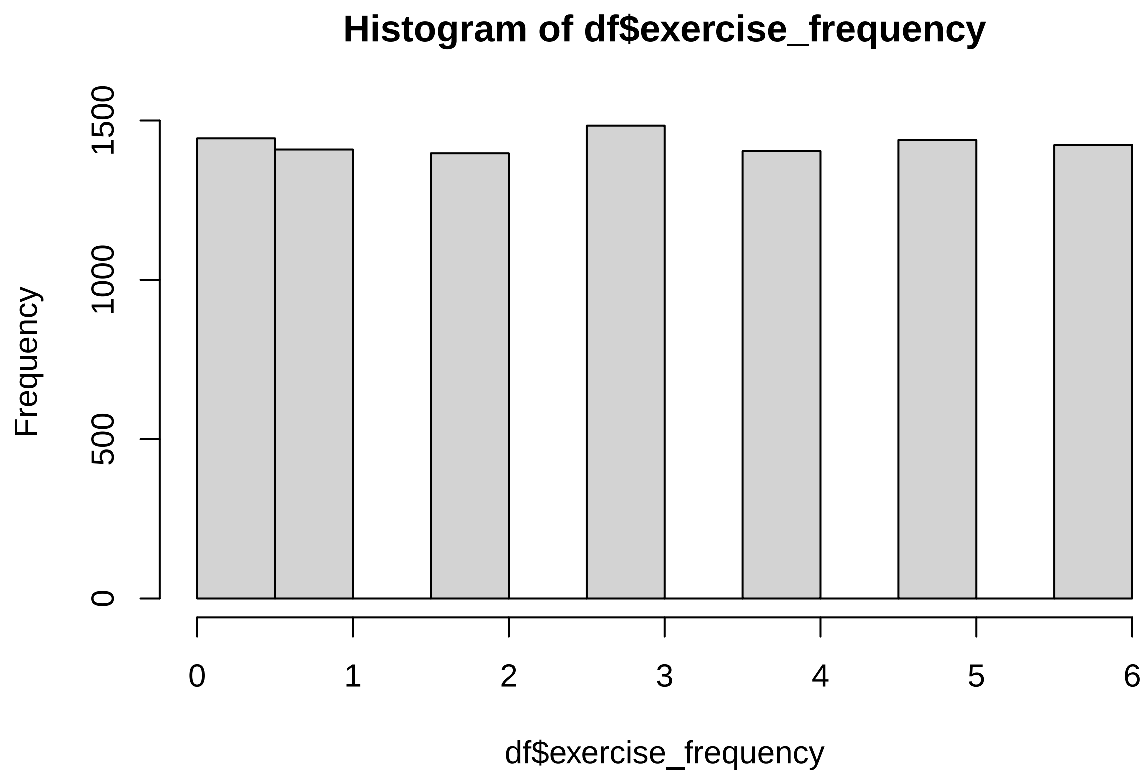
- **Visual Inspection:** Boxplots are a standard way to detect outliers in continuous data. The “whiskers” of a boxplot extend to 1.5 times the interquartile range (IQR) by default, and data points outside of this range are typically considered outliers.

```
# Convert the dataset from wide format to long format
df_long <- df %>%
  select(age, weight, height, cholesterol, blood_pressure, calorie_intake,bmi) %>%  # Selecting the rel
  pivot_longer(everything(), names_to="variable", values_to="value")

# Create a combined boxplot
ggplot(df_long, aes(y=value)) +
  geom_boxplot() +
  facet_wrap(~ variable, scales="free", ncol=2) +
  ggtitle("Boxplots for Selected Variables") +
  ylab("Value")
```



```
hist(df$exercise_frequency)
```



Here are the boxplots of some continuous features grouped by the diabetes status.

Age: There don't appear to be any outliers for age in either group. Cholesterol: There are some potential outliers in both diabetes and non-diabetes groups, especially on the higher end of the cholesterol scale. BMI: We can see potential outliers in both groups, mainly in the higher BMI range. Blood Pressure: There are potential outliers in the higher blood pressure range for both groups.

- **Z-Score:** The Z-score represents how many standard deviations a data point is from the mean. A common rule of thumb is that a data point with a Z-score greater than 3 or less than -3 is considered an outlier.

Steps:

1. **Calculate the Z-score**:
   - For each data point, subtract the mean and then divide by the standard deviation.

   $$Z = \frac{X - \mu}{\sigma}$$

   where X is the data point, μ is the mean, and σ is the standard deviation.
2. **Identify Outliers**:
   - Data points with a Z-score > 3 or $<$ -3 are typically considered outliers.

```r
# List of variables to check for outliers
variables <- c("age", "weight", "height", "cholesterol", "blood_pressure", "calorie_intake","bmi")

# Function to calculate Z-scores and identify outliers, ignoring missing values
identify_outliers <- function(var) {
  mean_val <- mean(df[[var]], na.rm = TRUE)
  sd_val <- sd(df[[var]], na.rm = TRUE)

  z_scores <- (df[[var]] - mean_val) / sd_val
  outliers <- ifelse(abs(z_scores) > 3, 1, 0)

  return(list(z_scores = z_scores, outliers = outliers))
}

# Apply the function to each variable
results <- lapply(variables, identify_outliers)

# Extract Z-scores and outliers for each variable and store them in the dataframe
for (i in seq_along(variables)) {
  var <- variables[i]
  df[paste0(var, "_z")] <- results[[i]]$z_scores
  df[paste0("outliers_", var)] <- results[[i]]$outliers
}
```

Based on the Z-score method, we identified potential outliers in the following feature:

```r
# Print out the number of outliers for each variable
sapply(variables, function(var) sum(df[[paste0("outliers_", var)]],na.rm=TRUE))
```

```
##            age         weight         height    cholesterol blood_pressure
##              0             26             21             30             13
## calorie_intake            bmi
##             30             40
```

Having identified these potential outliers, the next step is to decide how to handle them. There are a few strategies:

Deletion: Remove the outliers. This is straightforward but can lead to loss of data. **Capping:** Cap the outliers to a certain maximum or minimum value. Imputation: Replace the outliers with another value, like the mean or median of the feature.

We'll proceed by deleting the identified outliers:

```r
# Remove rows with identified outliers, but keep NA values
df_cleaned <- df %>%
  filter(rowSums(select(., starts_with("outliers_")), na.rm = TRUE) == 0)

# Optionally, drop the columns that were used to mark outliers and Z-scores
df_cleaned <- df_cleaned %>%
  select(-starts_with("outliers_"), -ends_with("_z"))

# Check the dimensions of the cleaned data
dim(df_cleaned)
```

```
## [1] 9844   12
```

The dataset's shape before and after removing outliers:

```r
# Check the dimensions of the raw data
dim(df)
```

```
## [1] 10000    26
```

```r
# Check the dimensions of the cleaned data
dim(df_cleaned)
```

```
## [1] 9844   12
```

2. Identify Missing Data Before analyzing, it's essential to check for missing data. Missing values can introduce bias or reduce the power of a model.

```r
# Checking for missing values in the dataset
missing_data_per_column <- sapply(df_cleaned, function(col) sum(is.na(col)))
missing_data_per_column
```

```
##         subject_id                age                sex             weight 
##                  0                  0                  0                  0 
##             height        cholesterol     blood_pressure     calorie_intake 
##                  0                 99                 98                  0 
## exercise_frequency           diabetes       checkup_date                bmi 
##                  0                  0                  0                  0
```

3. Handle Missing Data There are various techniques to handle missing data:

- **Deletion**: Remove rows with missing values. This is not always recommended as it can result in significant data loss, especially if many rows have missing values.
- **Mean/Median Imputation**: Replace missing values with the mean or median of the column. This method is simple and works well when the proportion of missing data is low.
- **Mode Imputation**: For categorical data, replace missing values with the mode (most frequent value) of the column.
- **Model-based Imputation**: Use models like k-NN or regression to predict and impute missing values.
- **Multiple Imputation**: Impute missing values multiple times to create multiple datasets and then average the results.
- **Forward Fill/Backward Fill:** Uses the previous or next known value to fill the missing value.

For this exercise, we'll use the simple mean imputation method.Let's go ahead and replace the missing values in the 'cholesterol' and 'blood_pressure' columns with their respective mean values:

```r
# Calculate the mean for 'cholesterol' and 'blood_pressure', ignoring NA values
mean_cholesterol <- mean(df_cleaned$cholesterol, na.rm = TRUE)
mean_blood_pressure <- mean(df_cleaned$blood_pressure, na.rm = TRUE)

# Replace NA values in 'cholesterol' and 'blood_pressure' with their respective means
df_cleaned$cholesterol[is.na(df_cleaned$cholesterol)] <- mean_cholesterol
df_cleaned$blood_pressure[is.na(df_cleaned$blood_pressure)] <- mean_blood_pressure
```

Verify the Imputation

Upon implementing the mean imputation, we verify to ensure that there are no more missing values in our dataset.

```r
sum(is.na(df_cleaned))
```

```
## [1] 0
```

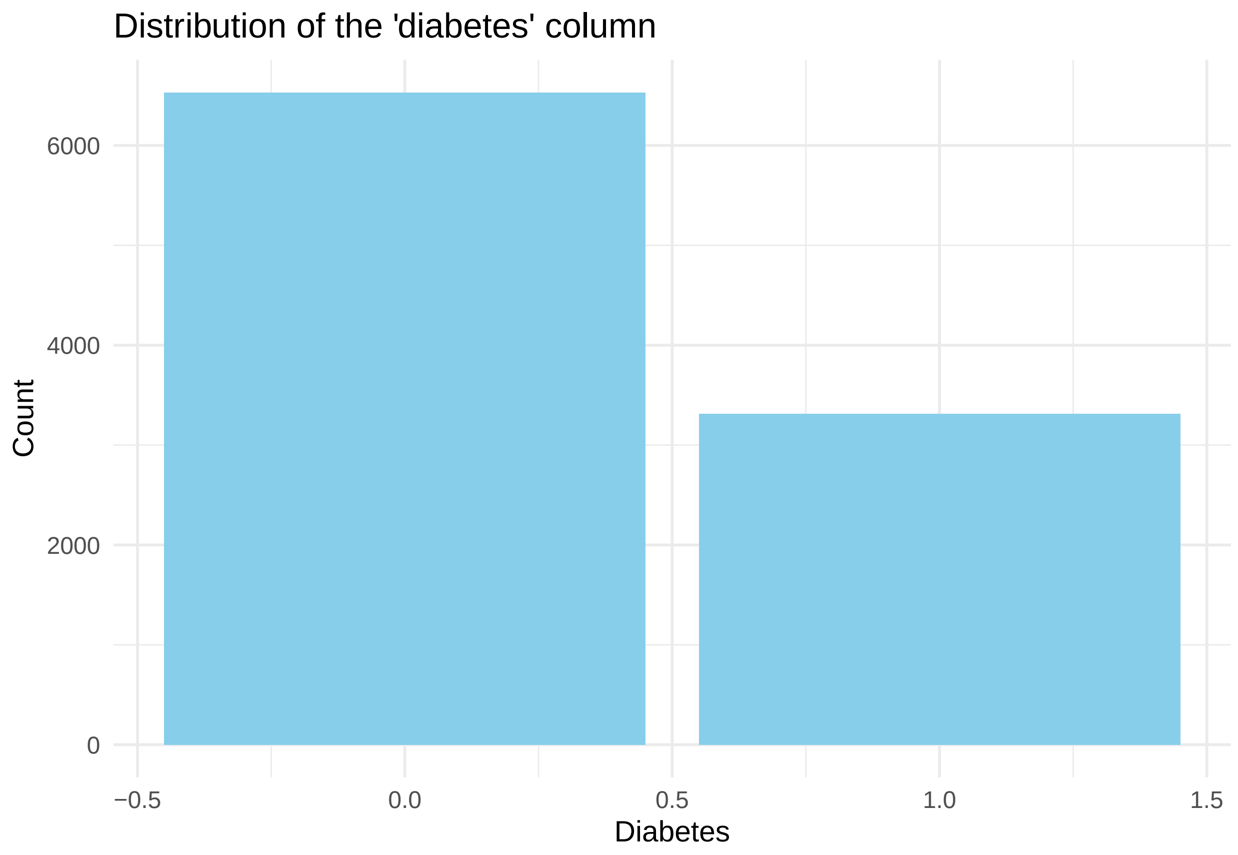
The result confirms that the 'cholesterol' and 'blood_pressure' columns no longer contain any missing values.

Part 2. Exploratory Data Analysis (EDA) It's about understanding the data distributions, relationships, and patterns:

- **Distribution of Target Variable:** Important to understand the class distribution, especially for classification problems.
- **Feature Relationships:** Scatter plots, correlation matrices, and pair plots can help visualize relationships.

1 Visualizing the Distribution of Target Variable It's always a good starting point to understand the distribution of the target variable. In our case, this is the 'diabetes' column. We'll visualize the distribution to see the balance between classes.

```r
# Plot the distribution of the 'diabetes' column
ggplot(df_cleaned, aes(x=diabetes)) +
  geom_bar(aes(y=..count..), fill="skyblue") +
  ggtitle("Distribution of the 'diabetes' column") +
  xlab("Diabetes") +
  ylab("Count") +
  theme_minimal()
```



Here's the distribution of the 'diabetes' variable:

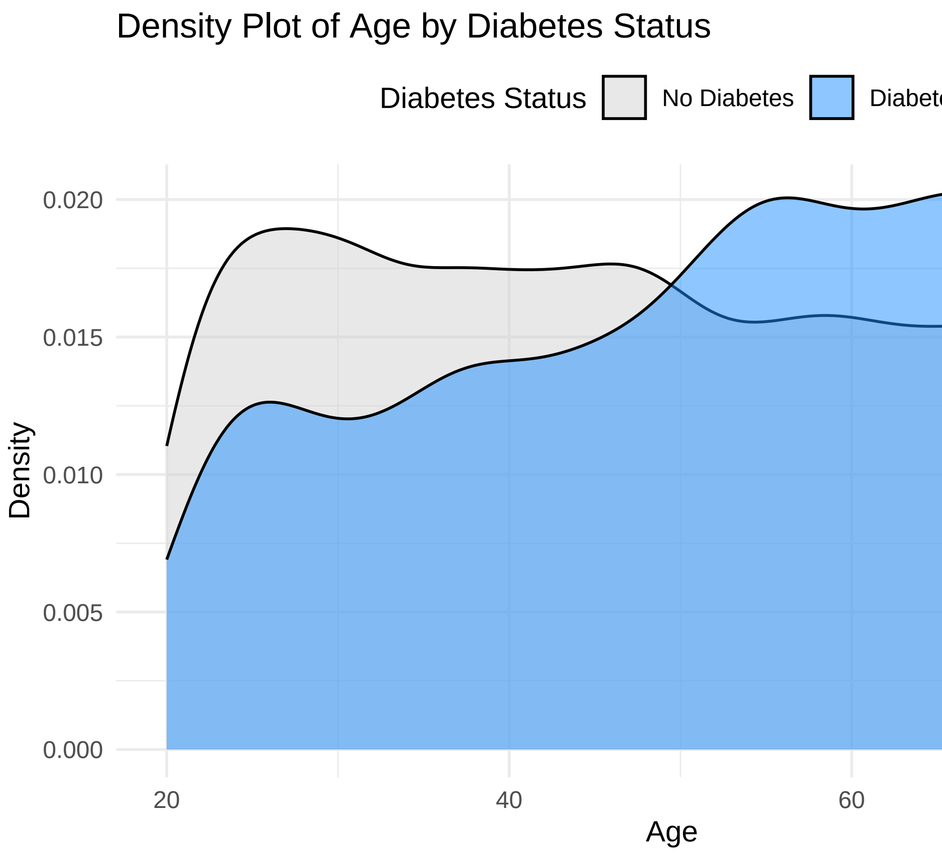
0 (No Diabetes): Represents subjects without diabetes.

1 (Diabetes): Represents subjects diagnosed with diabetes.

From the plot, we can see that there's a slightly higher count of subjects without diabetes compared to those with diabetes, but the classes are reasonably balanced.

```r
# Convert the 'diabetes' column to a factor
df_cleaned$diabetes <- factor(df_cleaned$diabetes, levels = c(0, 1), labels = c("No Diabetes", "Diabete

# Density Plot for Age vs. Diabetes Status
ggplot(df_cleaned, aes(x=age, fill=diabetes)) +
  geom_density(alpha=0.5) +
  ggtitle("Density Plot of Age by Diabetes Status") +
  xlab("Age") +
  ylab("Density") +
  scale_fill_manual(values=c("lightgray", "dodgerblue"), name="Diabetes Status", labels=c("No Diabetes"
  theme_minimal() +
  theme(legend.position="top")
```



2 Visualization for Age vs. Diabetes

The above plot illustrates the age distribution based on diabetes status:

Age vs. No Diabetes (0): This represents the age distribution of subjects without diabetes. **Age vs. Diabetes (1)**: This represents the age distribution of subjects with diabetes. From the plot, we can observe:

Younger subjects (around the age of 20-30) seem to have a lower prevalence of diabetes. The likelihood of having diabetes appears to increase with age, especially for subjects in the 50-70 age range. This visualization gives an idea of how the risk of a particular condition (in this case, diabetes) might change across different age groups.

3 Explore Relationships Among Features Beyond examining each feature's relationship with the target variable, it's also valuable to understand how the features relate to each other. This can provide insights into potential multicollinearity, underlying patterns, or areas of interest for further investigation.

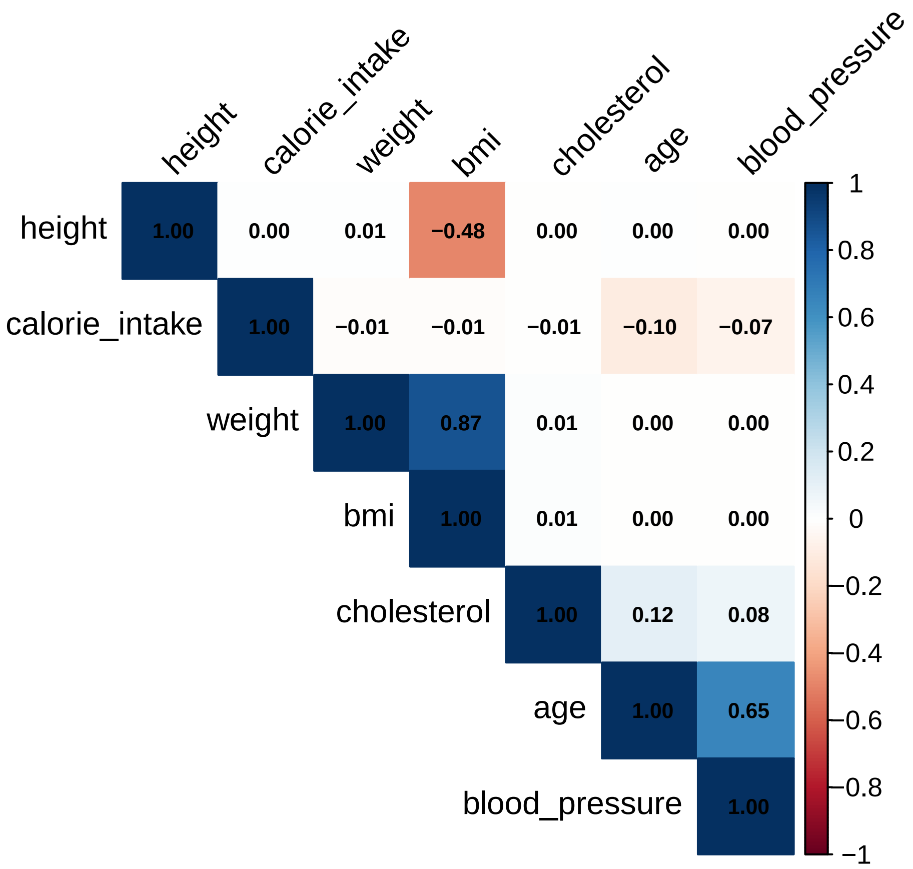
a. **Correlation Matrix:** One effective way to gauge linear relationships between continuous features is through a correlation matrix, which provides correlation coefficients (r-values) for each pair of features. This matrix can be visualized using a heatmap to make patterns more discernible.

Here's the heatmap representation of the correlation matrix for the continuous features:

```r
# Select continuous variables
continuous_vars <- df_cleaned %>%
  select(age, weight, height, cholesterol, blood_pressure, calorie_intake, bmi)

# Compute the correlation matrix
cor_matrix <- cor(continuous_vars, use="complete.obs")  # 'complete.obs' ensures that NAs are excluded
```

```r
# Plot the heatmap of the correlation matrix
corrplot(cor_matrix, method="color", type="upper", order="hclust",
         tl.col="black", tl.srt=45,addCoef.col="black", # Add correlation coefficients
         number.cex=0.7)
```



The color spectrum represents the range of correlation values, from -1 (perfect negative correlation) to 1 (perfect positive correlation). A value close to 0 suggests a weak linear relationship between the two variables. Annotations on the heatmap indicate the exact correlation coefficient between each pair of features. From the heatmap:

Age & Blood Pressure: There seems to be a positive correlation, indicating that as age increases, blood pressure tends to increase.

BMI & Weight: As expected, there's a strong positive correlation, since BMI is calculated using weight and height.

Understanding these correlations can be crucial when building linear models, as multicollinearity (high correlation between predictor variables) can destabilize the model. It's also insightful for feature engineering and selection.

b. **Pair Plots Pair plots** allow us to visualize relationships between multiple features at once. They can be especially informative when trying to discern patterns or clusters within the data.

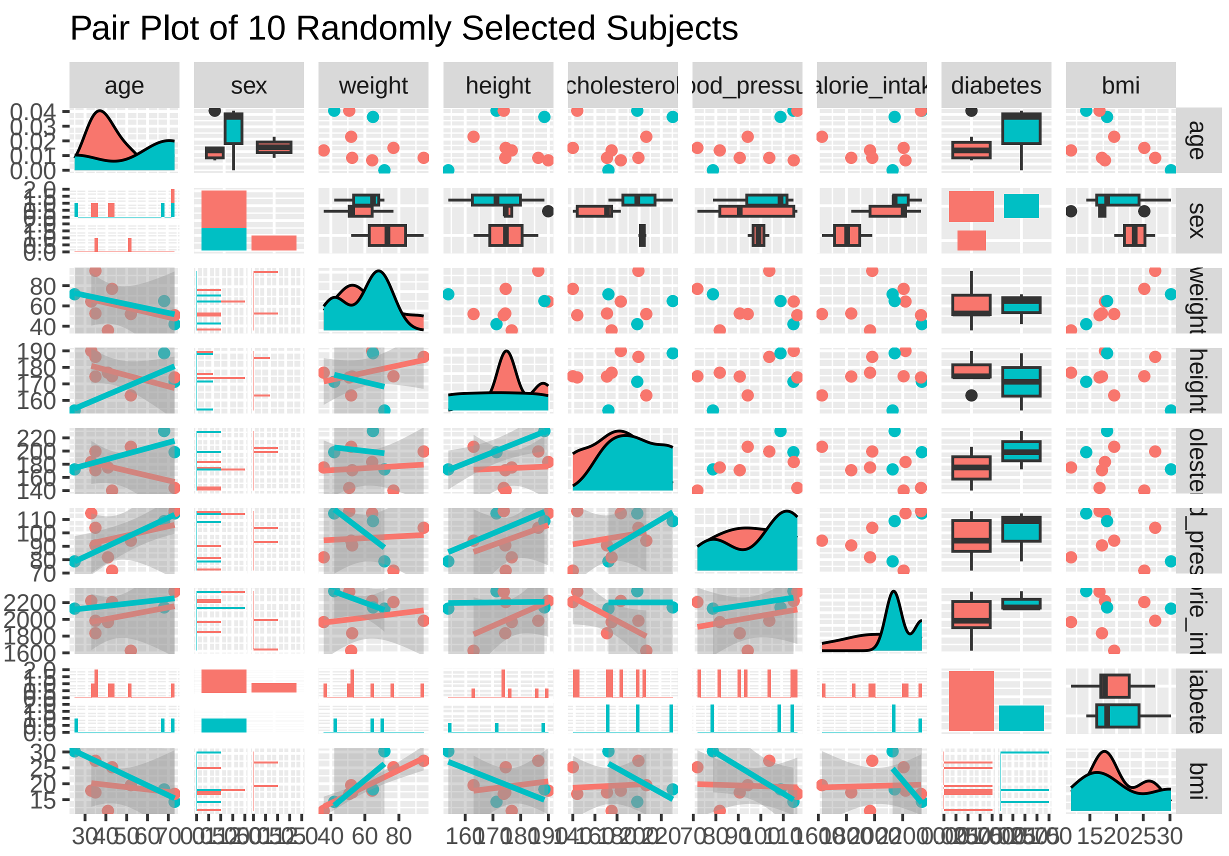
Let's assume that we can't plot the whole dataset due to the size of the table. We already have ~10K subjects, it is already quite computationally intense. In such case we can sample randomly subset of the subjects:

```
# Number of subjects to randomly select
N <- 10  # You can adjust this number as needed

# Randomly select N subjects
selected_data <- df_cleaned %>%
  select(age, sex, weight, height, cholesterol, blood_pressure, calorie_intake, diabetes,bmi) %>%
  sample_n(N)

# Create a pair plot for the continuous variables of the selected subjects
pair_plot <- ggpairs(selected_data,
                     aes(color=diabetes),  # Color by diabetes status
                     title=paste("Pair Plot of", N, "Randomly Selected Subjects"),
                     upper=list(continuous="points"),
                     lower=list(continuous="smooth"))

print(pair_plot)
```



The diagonal histograms represent the distribution of individual features. The scatter plots off the diagonal show the relationships between pairs of features. Data points are colored based on diabetes status, allowing us to discern potential patterns between features and diabetes outcomes. From the plots, we can make some observations:

BMI vs. Age: There's a slight trend indicating an increase in BMI with age. Also, higher BMI values seem more prevalent among subjects with diabetes.

Cholesterol vs. Age: Higher cholesterol levels appear more common in older individuals, and these higher levels also seem correlated with diabetes.

Blood Pressure vs. Age: Blood pressure seems to increase with age, and higher values are slightly more common among diabetic subjects. These visualizations provide with a comprehensive understanding of how the variables relate to each other and to the target variable. They are essential tools for data exploration and hypothesis generation.

Part 3: Machine Learning Model Training

1. Feature Engineering and Selection

Transforming raw data into a format that's more suitable for modeling.

- **Feature Scaling:**

When features in a dataset have different scales, certain algorithms, especially those that rely on distances or gradients (like k-NN, SVM, or gradient descent optimization algorithms), can perform poorly or take longer to converge. To overcome this, it's common to scale features so they have a similar scale.

There are two common methods of feature scaling:

1. **Min-Max Scaling (Normalization):** This scales features to lie between a given minimum and maximum value, often between zero and one. The formula for this is:

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

2. **Standard Scaling (Standardization):** This scales features to have a mean of 0 and a standard deviation of 1. The formula for this is:

$$X_{\text{std}} = \frac{X - \text{mean}(X)}{\text{std}(X)}$$

For this exercise, let's use Standard Scaling, as it's often the preferred method when there's no specific reason to prefer a fixed range.

```r
# Extract numerical columns
numerical_cols <- df_cleaned %>% select(where(is.numeric)) %>% colnames()

# Exclude non numerical variables from scaling if it's in the list
numerical_cols <- setdiff(numerical_cols,
                          c("subject_id","sex","diabetes","checkup_date","exercise_frequency"))

# Apply Standard Scaling to numerical columns
df_cleaned[numerical_cols] <- lapply(df_cleaned[numerical_cols], function(col) {
  (col - mean(col, na.rm = TRUE)) / sd(col, na.rm = TRUE)
})
```

In this code:

1. We first identify the numerical columns in the dataset.
2. We exclude categorical columns from scaling.
3. We apply Standard Scaling to each of the numerical columns using the `lapply` function.

4. We update the original dataframe with the scaled columns.

- **Categorical to Numerical:** Machine learning algorithms require numerical input, so categorical variables need to be converted to a numerical format. One of the most common methods for this conversion is called **one-hot encoding** (or dummy encoding). In this method, each category of a categorical variable is converted into a new binary column (0 or 1).

  For instance, consider a categorical variable "Color" with values "Red", "Green", and "Blue". One-hot encoding would transform this single column into three columns: "Color_Red", "Color_Green", and "Color_Blue". If a row originally had the value "Red" for the "Color" column, it would now have a 1 in the "Color_Red" column and 0s in the other two columns.

  Let's convert any categorical variables in the dataset to a numerical format using one-hot encoding:

```
# 1. Identify categorical columns
categorical_cols <- df_cleaned %>% select_if(function(col) is.factor(col) | is.character(col)) %>% colna


categorical_cols
```

```
## [1] "sex"      "diabetes"
```

```
# 0 add exercise_frequency as categorical
categorical_cols <- c(categorical_cols,"exercise_frequency")

# 1. Create a helper function for one-hot encoding
one_hot_encode <- function(data, col_name) {
  # Check if column exists
  if (!col_name %in% names(data)) {
    stop(paste("Column", col_name, "doesn't exist in the data."))
  }

  # Get unique values
  unique_vals <- unique(data[[col_name]])

  # Generate new column names based on unique values, excluding the first unique value
  new_cols <- paste0(col_name, "_", unique_vals[-1])

  # One-hot encode, skipping the first unique value
  for (val in unique_vals[-1]) {
    data[paste0(col_name, "_", val)] <- ifelse(data[[col_name]] == val, 1, 0)
  }

  # Drop the original column
  data[[col_name]] <- NULL

  return(data)
}


# 2. Check the number of unique levels or classes in each categorical column
cat_levels <- sapply(df_cleaned %>% select(all_of(categorical_cols)), function(col) length(unique(col)))
print(cat_levels)
```

```
##                 sex          diabetes exercise_frequency
##                   2                 2                  7
```

```
# 3. Decide on the encoding based on the number of unique levels
for (col in categorical_cols) {
  # Binary encoding
  if (length(unique(df_cleaned %>% pull(all_of(col)))) == 2) {
    levels <- unique(df_cleaned[[col]])
    df_cleaned[[col]] <- ifelse(df_cleaned[[col]] == levels[1], 1, 0)
  }
  # One-hot encoding
  else {
    df_cleaned <- one_hot_encode(df_cleaned, col)
  }
}
```

After executing this code, the features in `df_cleaned` will be standardized, making them more suitable for many machine learning algorithms.

```
# Check the first few rows of the scaled data
head(df_cleaned)
```

```
## # A tibble: 6 x 17
##   subject_id    age   sex  weight  height cholesterol blood_pressure
##        <dbl>  <dbl> <dbl>   <dbl>   <dbl>       <dbl>          <dbl>
## 1          1  0.837     1  0.601  -0.0781       0.640         0.0643
## 2          2  1.01      0 -0.0954  0.325       -0.101         0.524
## 3          3  1.36      1 -1.66   -0.440        0             0.283
## 4          4 -1.71      0 -0.689  -0.369        0.561        -2.29
## 5          5 -1.53      0  1.20   -0.124        0.859         0.235
## 6          6  1.70      1  0.306   0.0875       0.362         1.83
## # i 10 more variables: calorie_intake <dbl>, diabetes <dbl>,
## #   checkup_date <date>, bmi <dbl>, exercise_frequency_0 <dbl>,
## #   exercise_frequency_5 <dbl>, exercise_frequency_3 <dbl>,
## #   exercise_frequency_1 <dbl>, exercise_frequency_2 <dbl>,
## #   exercise_frequency_6 <dbl>
```

Before moving to the model training let's save the cleaned data, so we don't need to do this again:

```
# Save the training set
write.csv(df_cleaned, "data_cleaned.csv", row.names = FALSE,append = FALSE)
```

2. Data Splitting To assess the performance of our machine learning models, we need to split the dataset into a training set and a test set. A common practice is to use 80% of the data for training and 20% for testing.

Partitioning the data into training and test sets:

```
# Set a seed for reproducibility
set.seed(1234)

# Calculate the number of positive and negative samples needed for the training set
```

```r
num_positive_train <- sum(df_cleaned$diabetes == 1) * 0.8
num_negative_train <- sum(df_cleaned$diabetes == 0) * 0.8

# Get indices of positive and negative samples
positive_indices <- which(df_cleaned$diabetes == 1)
negative_indices <- which(df_cleaned$diabetes == 0)

# Randomly sample from positive and negative indices for the training set
train_positive_indices <- sample(positive_indices, size = floor(num_positive_train))
train_negative_indices <- sample(negative_indices, size = floor(num_negative_train))

# Combine the sampled indices to form the complete training index
train_index <- c(train_positive_indices, train_negative_indices)

# Split the data into training and test sets
train_set <- df_cleaned[train_index, ]
test_set <- df_cleaned[-train_index, ]

# Check the dimensions of the training and test sets
cat("Training set dimensions:", dim(train_set), "\n")
```

```
## Training set dimensions: 7874 17
```

```r
cat("Test set dimensions:", dim(test_set), "\n")
```

```
## Test set dimensions: 1970 17
```

The dataset has been successfully split:

Training Set: … subjects (features: …)

Test Set: … subjects (features: …)

With our data prepared, we can move on to training machine learning models.

3. Model Training We'll start by training a few commonly used classifiers for binary classification problems. Given our task, the following models are suitable:

Logistic Regression, Decision Trees, Random Forests, Support Vector Machines (SVM).

Metric/Algorithm	Logistic Regression	Decision Trees	Random Forests	Support Vector Machines (SVM)
Model Type	Probabilistic	Non-parametric	Ensemble	Maximum Margin
Output	Probability	Class/Value	Class/Value	Class/Value
Interpretability	High	High	Moderate	Low
Handling Missing Data	No (requires imputation)	Some implementations can	Yes (depends on implementation)	No (requires imputation)
Training Speed	Fast	Fast-Moderate	Moderate-Slow	Slow (especially for large datasets)
Prediction Speed	Fast	Fast	Fast	Moderate-Slow

Metric/Algorithm	Logistic Regression	Decision Trees	Random Forests	Support Vector Machines (SVM)
Parameter Tuning Needed	Moderate	Moderate	High	High
Regularization Options	Yes (e.g., L1, L2)	No	No (uses bagging)	Yes (e.g., C, kernel parameters)
Sensitivity to Outliers	Moderate	Yes	Less than decision trees	High (in non-linear kernel)
Scalability	Scalable	Scalable	Scalable but requires more resources	Not scalable for very large datasets
Common Use Cases	Binary/Multiclass classification	Classification, Regression	Classification, Regression	Classification, Regression, especially when classes are not linearly separable

Notes:

- **Interpretability**: Refers to how easy it is to understand the predictions made by the model.
- **Training Speed**: Refers to how quickly the algorithm can train on a dataset.
- **Prediction Speed**: Refers to how quickly the algorithm can make predictions once trained.
- **Parameter Tuning Needed**: Refers to how much tuning is typically required to achieve optimal performance.
- **Regularization Options**: Refers to the availability and types of regularization methods to prevent overfitting.
- **Sensitivity to Outliers**: Refers to how much outliers can affect the model's performance.
- **Scalability**: Refers to how well the algorithm can handle large datasets.

For each model, we'll train it using the training set and then compute its training accuracy.

Using the training data to train the model:

```r
# 1. Logistic Regression
logistic_model <- glm(diabetes ~ ., family = binomial(link = "logit"), data = train_set)
logistic_pred <- predict(logistic_model, train_set, type = "response")
logistic_accuracy <- mean(ifelse(logistic_pred > 0.5, 1, 0) == train_set$diabetes)

# 2. Decision Trees
tree_model <- rpart(diabetes ~ ., data = train_set, method = "class")
tree_pred <- predict(tree_model, train_set, type = "class")
tree_accuracy <- mean(tree_pred == train_set$diabetes)

# Ensure 'diabetes' is treated as a factor
train_set$diabetes <- as.factor(train_set$diabetes)

# 3. Random Forests
forest_model <- randomForest(diabetes ~ ., data = train_set)
```

```
forest_pred <- predict(forest_model, train_set)
forest_accuracy <- mean(forest_pred == train_set$diabetes)

# 4. Support Vector Machines (SVM)
svm_model <- svm(diabetes ~ ., data = train_set, kernel = "linear", type = "C-classification",probabili
svm_pred <- predict(svm_model, train_set)
svm_accuracy <- mean(svm_pred == train_set$diabetes)
```

After training each model, we'll compute its training accuracy:

```
cat("Training Accuracy for Logistic Regression:", logistic_accuracy, "\n")
```

```
## Training Accuracy for Logistic Regression: 0.8139446
```

```
cat("Training Accuracy for Decision Tree:", tree_accuracy, "\n")
```

```
## Training Accuracy for Decision Tree: 0.9001778
```

```
cat("Training Accuracy for Random Forest:", forest_accuracy, "\n")
```

```
## Training Accuracy for Random Forest: 1
```

```
cat("Training Accuracy for SVM:", svm_accuracy, "\n")
```

```
## Training Accuracy for SVM: 0.7993396
```

4. Model Evaluation

- **Compute the accuracy, precision, recall, and F1-score for the test dataset predictions.**

1. **Accuracy**:
   - **Definition**: It's the ratio of correctly predicted instances to the total instances in the data set.
   - **Formula**:
   $$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$
   - **Interpretation**: While accuracy gives a general overview of model performance, it might not be the best metric, especially for imbalanced datasets. In such cases, a model might predict the majority class for all inputs and still achieve a high accuracy.
2. **Precision**:
   - **Definition**: It measures the accuracy of positive predictions.
   - **Formula**:
   $$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$
   - **Interpretation**: High precision indicates that the false positive rate is low. For instance, in medical testing, precision would indicate how many of the positive test results are actually correct.
3. **Recall (or Sensitivity or True Positive Rate)**:
   - **Definition**: It measures the fraction of the actual positives that were correctly identified.

- **Formula**:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- **Interpretation**: A high recall indicates that most of the positive instances were captured by the model. In contexts where missing a positive instance is crucial (like detecting a disease), recall is especially important.

4. **F1-Score**:

- **Definition**: It's the harmonic mean of precision and recall. The harmonic mean is used here because it gives equal weight to both precision and recall and is a more robust metric when there are huge differences between them.
- **Formula**:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Interpretation**: F1-score is especially useful when the class distribution is imbalanced. It tries to find the balance between precision and recall. A perfect F1-score of 1 indicates perfect precision and recall, while a score of 0 indicates that either the precision or the recall is zero.

When evaluating a model, it's important to consider all these metrics collectively, especially in contexts where certain types of errors might have bigger consequences than others.

```r
# Function to compute metrics from confusion matrix
compute_metrics <- function(cm) {
  TP <- cm[2, 2]
  TN <- cm[1, 1]
  FP <- cm[1, 2]
  FN <- cm[2, 1]

  Accuracy <- (TP + TN) / (TP + TN + FP + FN)
  Precision <- TP / (TP + FP)
  Recall <- TP / (TP + FN)
  F1_Score <- 2 * (Precision * Recall) / (Precision + Recall)

  return(list(Accuracy = Accuracy, Precision = Precision, Recall = Recall, F1_Score = F1_Score))
}


# Ensure that 'diabetes' in the test set is treated as a factor
test_set$diabetes <- as.factor(test_set$diabetes)

# Predictions on test data
logistic_test_pred <- predict(logistic_model, newdata = test_set, type = "response")
logistic_test_pred <- ifelse(logistic_test_pred > 0.5, 1, 0)

tree_test_pred <- predict(tree_model, newdata = test_set, type = "class")
forest_test_pred <- predict(forest_model, newdata = test_set)
svm_test_pred <- predict(svm_model, newdata = test_set)


# For each model's prediction, ensure factor levels match those in the test set
logistic_test_pred <- factor(logistic_test_pred, levels = levels(test_set$diabetes))
tree_test_pred <- factor(tree_test_pred, levels = levels(test_set$diabetes))
forest_test_pred <- factor(forest_test_pred, levels = levels(test_set$diabetes))
svm_test_pred <- factor(svm_test_pred, levels = levels(test_set$diabetes))
```

```r
# Now, compute metrics for each model as before
models <- list(Logistic = logistic_test_pred,
               Tree = tree_test_pred,
               Forest = forest_test_pred,
               SVM = svm_test_pred)


# Initialize a data frame to store the metrics
metrics_df <- data.frame(Model = character(),
                         Accuracy = numeric(),
                         Precision = numeric(),
                         Recall = numeric(),
                         F1_Score = numeric(),
                         stringsAsFactors = FALSE)

# Populate the data frame with metrics for each model
for (model_name in names(models)) {
  pred_values <- models[[model_name]]
  cm <- table(pred_values, test_set$diabetes)
  metrics <- compute_metrics(cm)

  metrics_df <- rbind(metrics_df, data.frame(
    Model = model_name,
    Accuracy = metrics$Accuracy,
    Precision = metrics$Precision,
    Recall = metrics$Recall,
    F1_Score = metrics$F1_Score
  ))
}

rownames(metrics_df) <- NULL
print(metrics_df)
```

```
##      Model  Accuracy Precision    Recall  F1_Score
## 1 Logistic 0.8005076 0.8875287 0.8250356 0.8551419
## 2     Tree 0.8802030 0.9502678 0.8789809 0.9132353
## 3   Forest 0.8837563 0.9517980 0.8822695 0.9157159
## 4      SVM 0.7837563 0.8714614 0.8153185 0.8424556
```

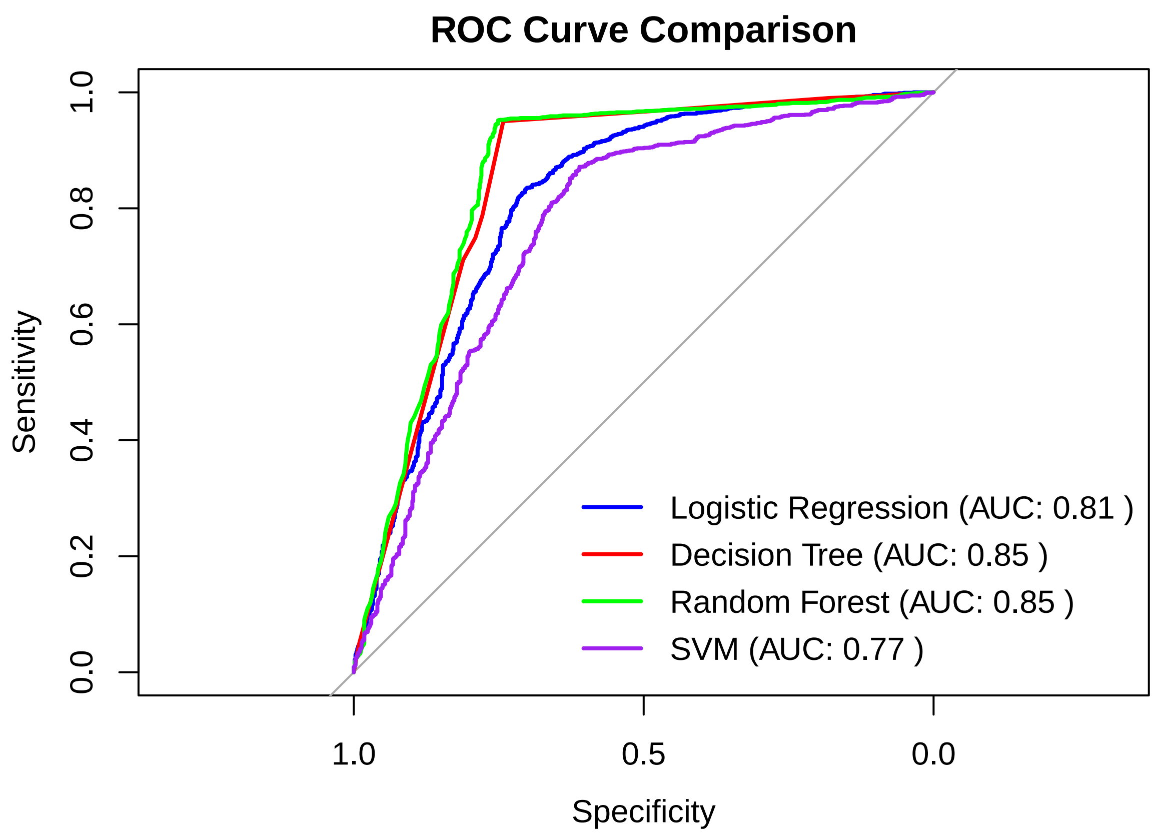
- **ROC Curve and AUC** The Receiver Operating Characteristic (ROC) curve is a graphical representation of a classifier's performance across all thresholds. The Area Under the Curve (AUC) provides a single value summary of the model performance, with 1.0 being a perfect classifier and 0.5 representing a model that performs no better than random guessing.

```r
# Extract predicted probabilities for each model
logistic_probs <- predict(logistic_model, newdata = test_set, type = "response")
tree_probs <- predict(tree_model, newdata = test_set, type = "prob")[,2]
forest_probs <- predict(forest_model, newdata = test_set, type = "prob")[,2]
svm_probs <- attr(predict(svm_model, newdata = test_set, probability = TRUE), "probabilities")[,2]
```

```r
# Create ROC curves for each model
roc_obj_logistic <- roc(test_set$diabetes, logistic_probs)
roc_obj_tree <- roc(test_set$diabetes, tree_probs)
roc_obj_forest <- roc(test_set$diabetes, forest_probs)
roc_obj_svm <- roc(test_set$diabetes, svm_probs)

# Plot the ROC curves
plot(roc_obj_logistic, col = "blue", lwd = 2, main = "ROC Curve Comparison")
lines(roc_obj_tree, col = "red", lwd = 2)
lines(roc_obj_forest, col = "green", lwd = 2)
lines(roc_obj_svm, col = "purple", lwd = 2)

# Add legend with AUC values
legend("bottomright",
       legend = c(paste("Logistic Regression (AUC:", round(auc(roc_obj_logistic), 2), ")"),
                  paste("Decision Tree (AUC:", round(auc(roc_obj_tree), 2), ")"),
                  paste("Random Forest (AUC:", round(auc(roc_obj_forest), 2), ")"),
                  paste("SVM (AUC:", round(auc(roc_obj_svm), 2), ")")),
       col = c("blue", "red", "green", "purple"),
       lwd = 2, bty = "n")
```



- **Confusion Matrix** Here are the confusion matrices for each model:

```r
# Predict class labels using each model on the test data
logistic_preds <- ifelse(predict(logistic_model, newdata = test_set, type = "response") > 0.5, 1, 0)
tree_preds <- predict(tree_model, newdata = test_set, type = "class")
forest_preds <- predict(forest_model, newdata = test_set, type = "class")
svm_preds <- predict(svm_model, newdata = test_set)

# Convert predictions and true labels to factor
logistic_preds <- as.factor(logistic_preds)
tree_preds <- as.factor(tree_preds)
forest_preds <- as.factor(forest_preds)
svm_preds <- as.factor(svm_preds)
true_labels <- as.factor(test_set$diabetes)

# Generate confusion matrices using the table function
cm_logistic <- table(logistic_preds, true_labels)
cm_tree <- table(tree_preds, true_labels)
cm_forest <- table(forest_preds, true_labels)
cm_svm <- table(svm_preds, true_labels)

# Print confusion matrices
cat("\nConfusion Matrix for Logistic Regression:\n")
```

```
##
## Confusion Matrix for Logistic Regression:
```

```r
print(cm_logistic)
```

```
##                 true_labels
## logistic_preds    0    1
##               0  417  147
##               1  246 1160
```

```r
cat("\nConfusion Matrix for Decision Tree:\n")
```

```
##
## Confusion Matrix for Decision Tree:
```

```r
print(cm_tree)
```

```
##             true_labels
## tree_preds    0    1
##           0  492   65
##           1  171 1242
```

```r
cat("\nConfusion Matrix for Random Forest:\n")
```

```
##
## Confusion Matrix for Random Forest:
```

```r
print(cm_forest)
```

```
##              true_labels
## forest_preds    0    1
##            0  497   63
##            1  166 1244
```

```r
cat("\nConfusion Matrix for SVM:\n")
```

```
##
## Confusion Matrix for SVM:
```

```r
print(cm_svm)
```

```
##          true_labels
## svm_preds    0    1
##         0  405  168
##         1  258 1139
```

Rows represent the actual classes, while columns represent the predicted classes. The top-left and bottom-right values (diagonal) represent correct predictions (True Negatives and True Positives respectively). The top-right and bottom-left values represent incorrect predictions (False Positives and False Negatives).

The confusion matrix provides a comprehensive view of how each model performs in terms of Type I and Type II errors. It's especially valuable when the costs of false positives and false negatives differ significantly.

Part 4: Summary and conclusion The table you provided contains evaluation metrics (Accuracy, Precision, Recall, and F1_Score) for four different models: Logistic Regression, Decision Tree, Random Forest, and SVM. Let's break down and interpret the results for each model:

1. **Logistic Regression**:
   - **Accuracy (0.8005)**: About 80% of the predictions made by the model are correct. This is a decent accuracy, but it's always essential to compare with a naive/baseline model.
   - **Precision (0.8875)**: Of the subjects the model predicted to have diabetes, about 88.75% actually had diabetes.
   - **Recall (0.8250)**: Of all the subjects who actually had diabetes, the model identified about 82.5% of them.
   - **F1_Score (0.8551)**: The harmonic mean of precision and recall is about 85.5%, indicating a balanced performance between precision and recall.
2. **Decision Tree**:
   - Higher accuracy, precision, and F1_Score compared to Logistic Regression. This indicates that the Decision Tree model performs better than the Logistic model on this dataset.
3. **Random Forest**:
   - The Random Forest model has the highest accuracy among all models and is very close to the Decision Tree in terms of precision, recall, and F1_Score. Given that Random Forest is an ensemble of Decision Trees, this is expected. It usually performs better or at least as well as a single Decision Tree.
4. **SVM**:

- The SVM has the lowest accuracy among the models, but its precision, recall, and F1_Score are reasonably high, indicating that it's still a competent model.

Overall Insights:

- **Best Overall Model**: The Random Forest appears to be the best model in terms of accuracy.
- **Precision**: All models have high precision, indicating that when they predict a positive class (diabetes), they are usually correct.
- **Recall**: Recall is also relatively high for all models, but the Decision Tree and Random Forest perform slightly better.
- **F1_Score**: Random Forest and Decision Tree have the highest F1 scores, indicating a good balance between precision and recall.

Recommendations:

- If the primary goal is to capture as many true positive cases as possible, even at the risk of increasing false positives, then one might prioritize Recall. In such a case, the Random Forest or Decision Tree would be the top choice.
- If the primary goal is to be very sure about the positive predictions made, then one might prioritize Precision. Here, again, the Random Forest or Decision Tree would be preferred.
- If there's a need for a balance between precision and recall, the F1_Score is a good metric to consider. The Random Forest and Decision Tree both perform well in this aspect.

Lastly, it's essential to consider the context and the cost associated with false positives and false negatives. For instance, in medical diagnostics, a false negative might have more severe consequences than a false positive. Depending on the context, you might prioritize one metric over another.