University of Tehran
College of Engineering
Department of Mechanical Engineering

Title:

2nd Homework (Machine Learning)

Course:

Artificial Intelligence

Author:

Hatef Mohammadi Alashti

810601123

Course Instructor:

Dr. Shariat Panahi

May 2024

University of Tehran
College of Engineering
Department of Mechanical Engineering

1 - Regression Methods

The goal of this section is to develop and train a model that can predict a car's price using 24 numerical or categorical features.

A) Studying Raw Data

The first step is to read the CSV file.

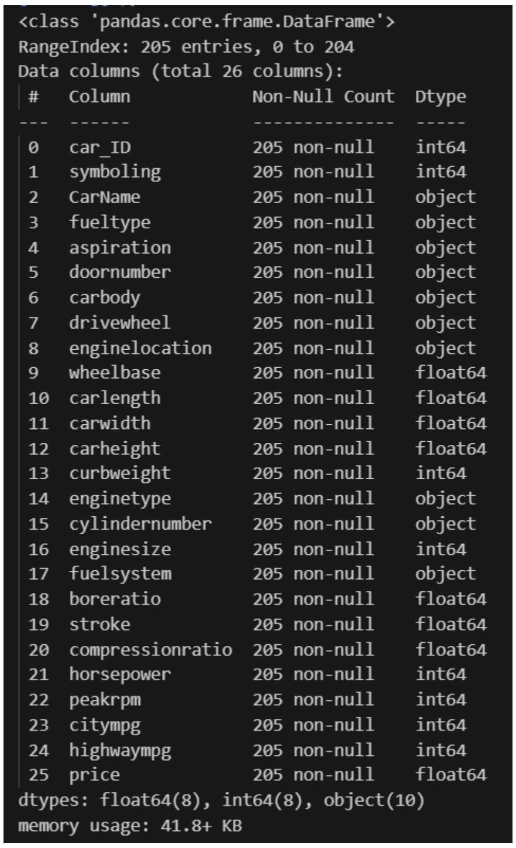
```
# Load the DataFrame from the .CSV File
file_path = 'D:\\Uni\\Semester 04\\Artificial Intelligence\\HW\\HW#2\\CarPrice.csv'
car_data = pd.read_csv(file_path)
```

A-1) Data Framework

To obtain an overview of the data structure, the “info” function was used.

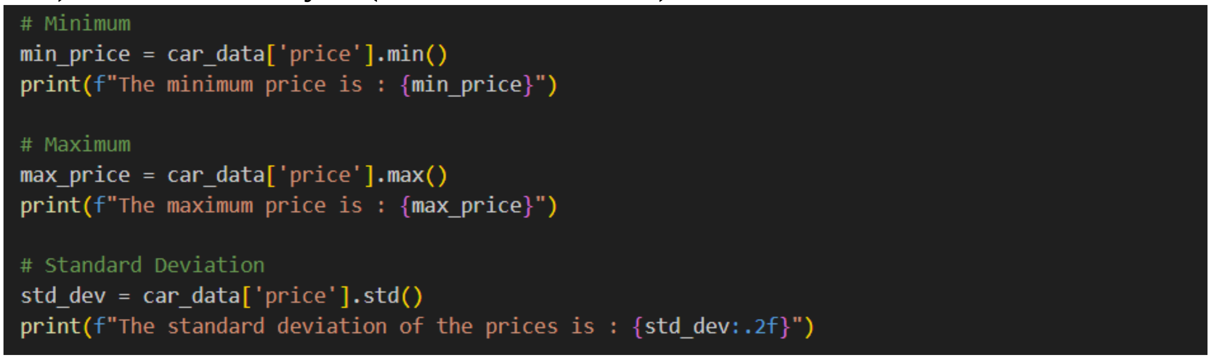
```
# a-1) DataFrame, Info Method
car_data.info()
```

Output: We can identify numerical (int64 and float64), and categorical (object) features.



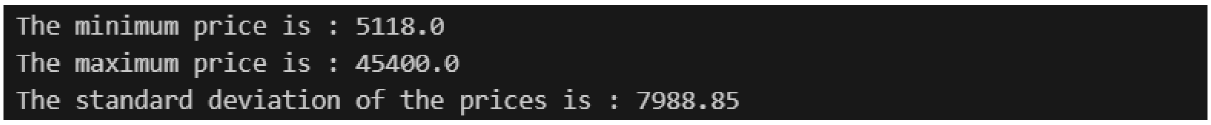
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 26 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   car_ID            205 non-null    int64  
 1   symboling         205 non-null    int64  
 2   CarName           205 non-null    object 
 3   fueltype          205 non-null    object 
 4   aspiration        205 non-null    object 
 5   doornumber        205 non-null    object 
 6   carbody           205 non-null    object 
 7   drivewheel        205 non-null    object 
 8   enginelocation    205 non-null    object 
 9   wheelbase         205 non-null    float64
 10  carlength         205 non-null    float64
 11  carwidth          205 non-null    float64
 12  carheight         205 non-null    float64
 13  curbweight        205 non-null    int64  
 14  enginetype        205 non-null    object 
 15  cylindernumber    205 non-null    object 
 16  enginesize        205 non-null    int64  
 17  fuelsystem        205 non-null    object 
 18  boreratio         205 non-null    float64
 19  stroke            205 non-null    float64
 20  compressionratio  205 non-null    float64
 21  horsepower        205 non-null    int64  
 22  peakrpm           205 non-null    int64  
 23  citympg           205 non-null    int64  
 24  highwaympg        205 non-null    int64  
 25  price             205 non-null    float64
dtypes: float64(8), int64(8), object(10)
memory usage: 41.8+ KB
```

A-2) Statistical Analysis (Min, Max, Std Dev)



```python
# Minimum
min_price = car_data['price'].min()
print(f"The minimum price is : {min_price}")

# Maximum
max_price = car_data['price'].max()
print(f"The maximum price is : {max_price}")

# Standard Deviation
std_dev = car_data['price'].std()
print(f"The standard deviation of the prices is : {std_dev:.2f}")
```

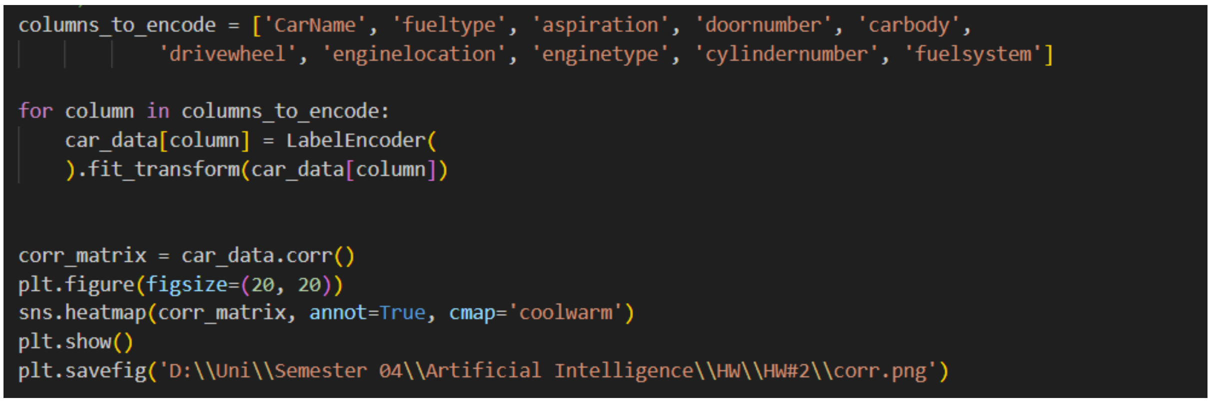
Output:



```
The minimum price is : 5118.0
The maximum price is : 45400.0
The standard deviation of the prices is : 7988.85
```

A-3) Correlation Matrix

At first, we assign numbers to categorical data to have a homogeneous data frame.



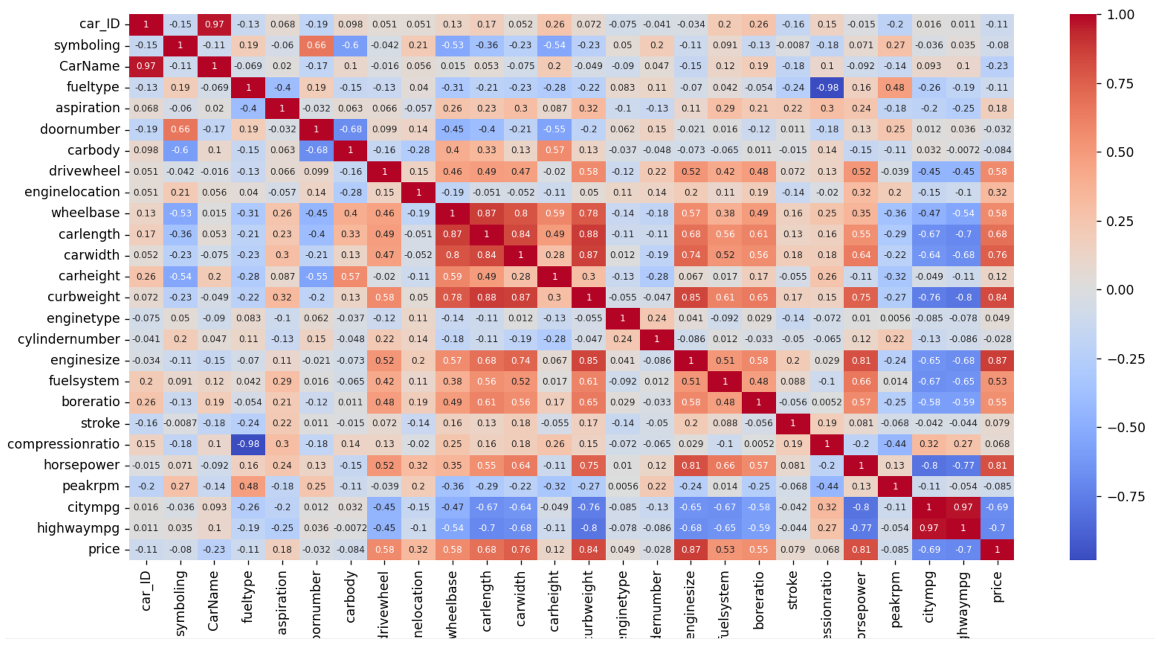
```python
columns_to_encode = ['CarName', 'fueltype', 'aspiration', 'doornumber', 'carbody',
                     'drivewheel', 'enginelocation', 'enginetype', 'cylindernumber', 'fuelsystem']

for column in columns_to_encode:
    car_data[column] = LabelEncoder(
    ).fit_transform(car_data[column])


corr_matrix = car_data.corr()
plt.figure(figsize=(20, 20))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.show()
plt.savefig('D:\\Uni\\Semester 04\\Artificial Intelligence\\HW\\HW#2\\corr.png')
```

Output:

From the figure below, the most important features (to the price) can be selected:

Enginsize (0.87), curbweight (0.84), horsepower (0.81), carwidth (0.76), highwaympg (0.70), citympg (0.69), and carlength (0.68).

Also, it can be seen that the correlation factors between highwaympg and citympg, and between fueltype and compressionratio are 0.97, and -0.98 respectively. This means that they are strongly colinear, so it is justifiable to keep only one of each pair.



B) Preprocess

In this section, we aim to neglect as much data as possible to reduce the calculation load. Firstly, we only keep features that have correlation factors of over 0.2 to the price.

```python
# b) preprocess
corr_price = corr_matrix['price'].abs()
high_corr_features = corr_price[corr_price >= 0.2].index.tolist()
```

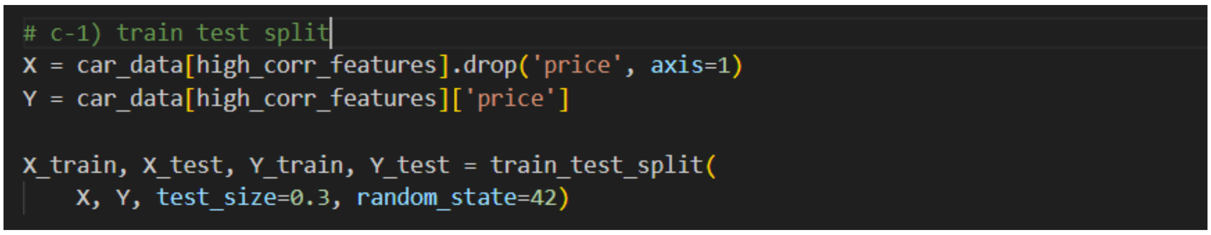
Then, for pairs that have high colinearity, we only keep one of each pair. We only have to remove one from highwaympg and citympg, as the other pair (fueltype and compressionration) did not make the list to this point due to having a correlation factor lower than 0.2.

```python
items_to_remove = 'highwaympg'
high_corr_features = [
    item for item in high_corr_features if item != items_to_remove]
```

C) Selection, Training, and Evaluation

C-1) Train and Test Split

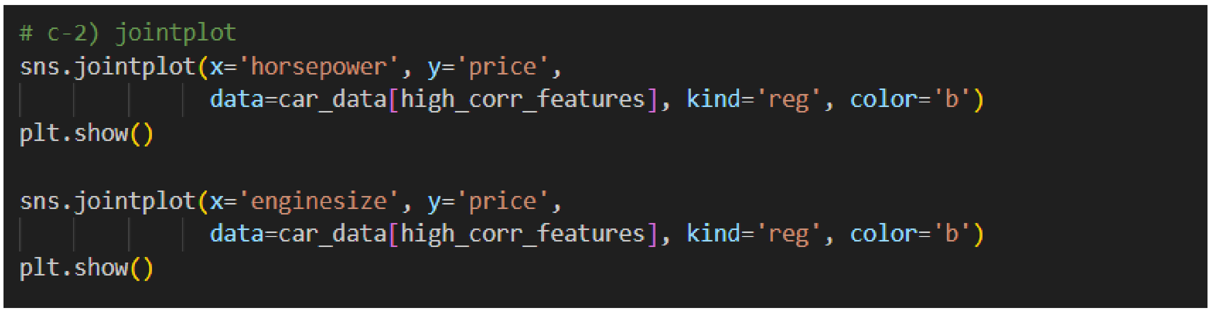
We have been asked to split the data frame into test (30%) and train (70%) sets. To do this, first, we only keep the features (we exclude price). Then, from the scikit-learn module, we implement the train_test_split function with test size = 0.3 and random_state = 42.



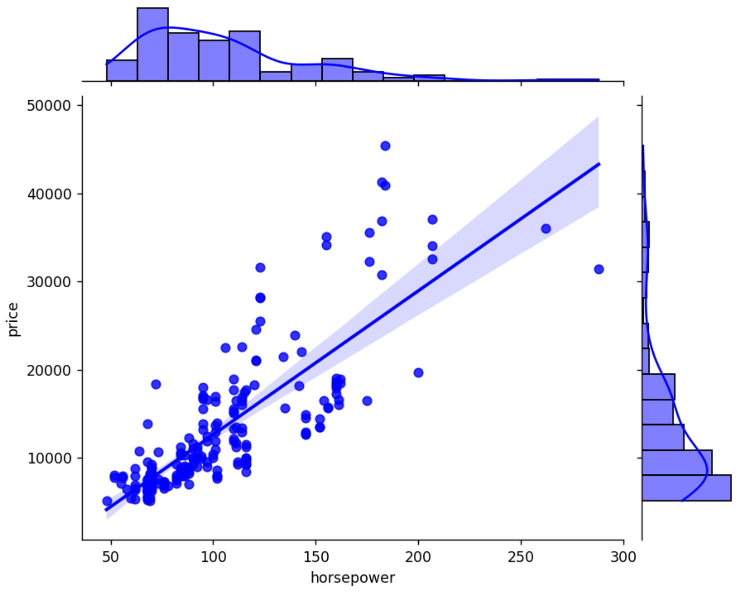
```
# c-1) train test split
X = car_data[high_corr_features].drop('price', axis=1)
Y = car_data[high_corr_features]['price']

X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.3, random_state=42)
```

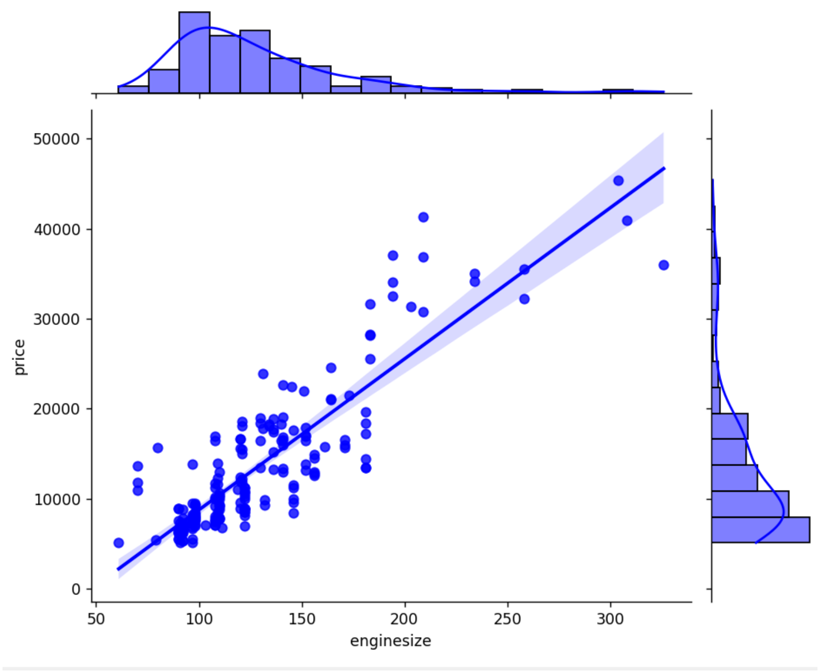
C-2) Jointplot (Horsepower and Enginesize)

In this stage, we are inclined to ensure whether horsepower and enginesize are suitable to be implemented in a linear regression model. To do so, we are asked to use jointplot finction from the seaborn module.



```
# c-2) jointplot
sns.jointplot(x='horsepower', y='price',
              data=car_data[high_corr_features], kind='reg', color='b')
plt.show()

sns.jointplot(x='enginesize', y='price',
              data=car_data[high_corr_features], kind='reg', color='b')
plt.show()
```

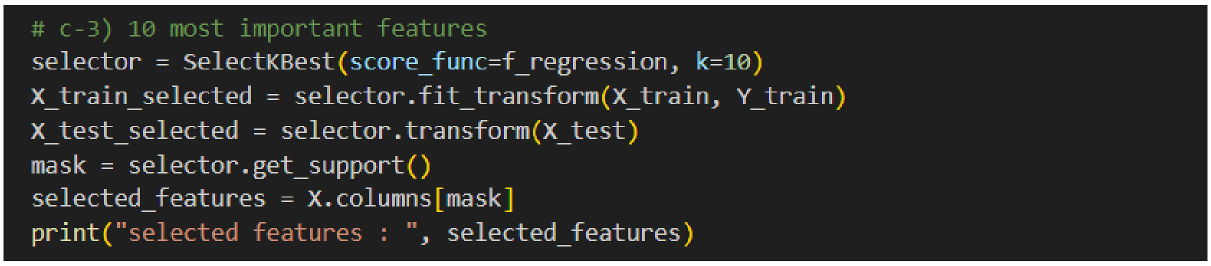




It can easily be deduced that both these features have a fair linearity with the price. Therefore, they have no problem to be taken into a linear regression model.

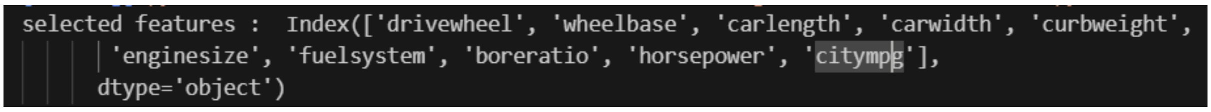
C-3) Selection of Best Features

Now, we select 10 of the most impactful features on the price by using the SelectKBest function from the sci-kit-learn module. We set the score function to be f_regression.



```
# c-3) 10 most important features
selector = SelectKBest(score_func=f_regression, k=10)
X_train_selected = selector.fit_transform(X_train, Y_train)
X_test_selected = selector.transform(X_test)
mask = selector.get_support()
selected_features = X.columns[mask]
print("selected features : ", selected_features)
```

Output:



```
selected features :  Index(['drivewheel', 'wheelbase', 'carlength', 'carwidth', 'curbweight',
       'enginesize', 'fuelsystem', 'boreratio', 'horsepower', 'citympg'],
      dtype='object')
```

C-4, C-5, C-6) Train and Evaluate

This is the section in which we train the four models in question. First, we need to initialize the models. We set the alpha parameter to be one for both the Lasso and the Ridge models. Also, we set the kernel parameter to linear for the Support Vector Regression (SVR) model.

```python
# c-4, 5, 6
# Initialize Models
linear_model = LinearRegression()
lasso_model = Lasso(alpha=1.0)
ridge_model = Ridge(alpha=1.0)
svr_model = SVR(kernel='linear')
```

Then, we define a function that takes the model and the data sets as the input. Next, it fits the model according to the train and test sets. In addition, the evaluation of the model (RMSE and R^2 score) is calculated.

```python
def train_and_evaluate(model, X_train, X_test, Y_train, Y_test):
    model.fit(X_train, Y_train)
    Y_pred = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(Y_test, Y_pred))
    r2 = r2_score(Y_test, Y_pred)
    return rmse, r2
```

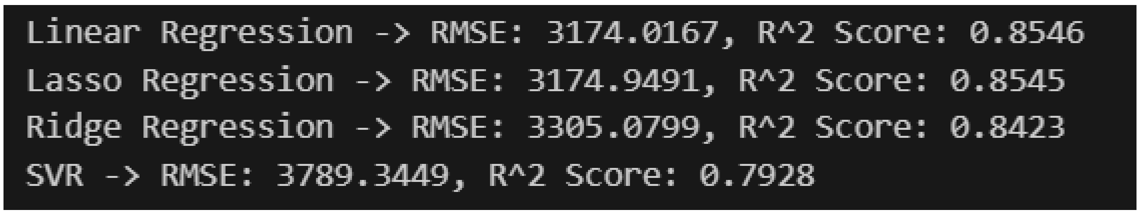
Resultantly, we set a dictionary of the models to be implemented in a for loop to be trained and evaluated.

```python
models = {
    'Linear Regression': linear_model,
    'Lasso Regression': lasso_model,
    'Ridge Regression': ridge_model,
    'SVR': svr_model
}
```

For the last section, in a for loop that runs through the aforementioned dictionary of the models, we train and evaluate them. Ultimately, we print the RMSE and R^2 score of the models.

```python
results = {}
for name, model in models.items():
    rmse, r2 = train_and_evaluate(model, X_train, X_test, Y_train, Y_test)
    results[name] = {'RMSE': rmse, 'R^2 Score': r2}

for model in results:
    print(
        f"{model} -> RMSE: {results[model]['RMSE']:.4f}, R^2 Score: {results[model]['R^2 Score']:.4f}")
```

Output:



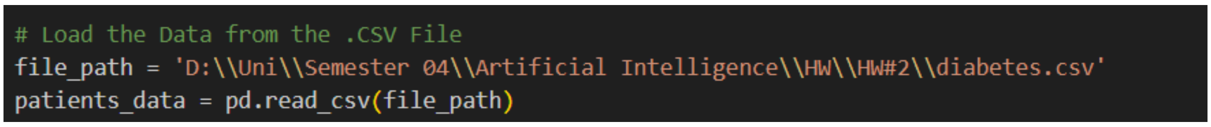
```
Linear Regression -> RMSE: 3174.0167, R^2 Score: 0.8546
Lasso Regression -> RMSE: 3174.9491, R^2 Score: 0.8545
Ridge Regression -> RMSE: 3305.0799, R^2 Score: 0.8423
SVR -> RMSE: 3789.3449, R^2 Score: 0.7928
```

2 - Classification

The second half of the homework is designated to develop and train a model to predict whether one has diabetes based on 8 features.

A) Studying Raw Data

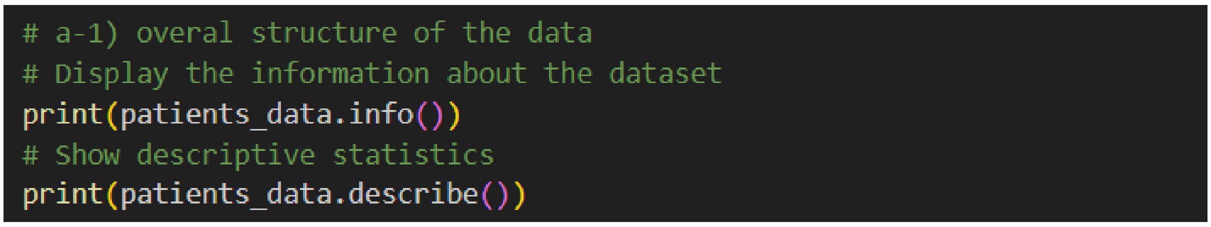
The first step is to read the spreadsheet file.



```
# Load the Data from the .CSV File
file_path = 'D:\\Uni\\Semester 04\\Artificial Intelligence\\HW\\HW#2\\diabetes.csv'
patients_data = pd.read_csv(file_path)
```

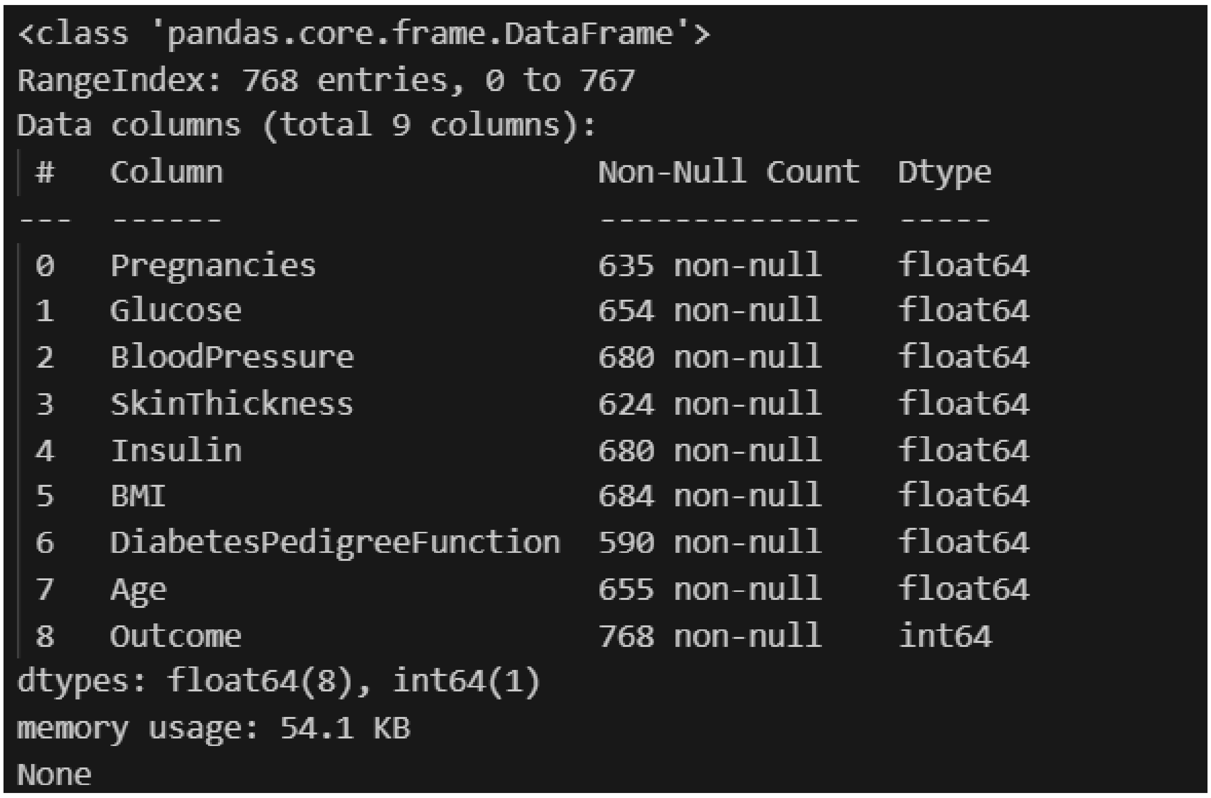
A-1) Data Framework

To get an overview of the data frame, we use info and describe functions.



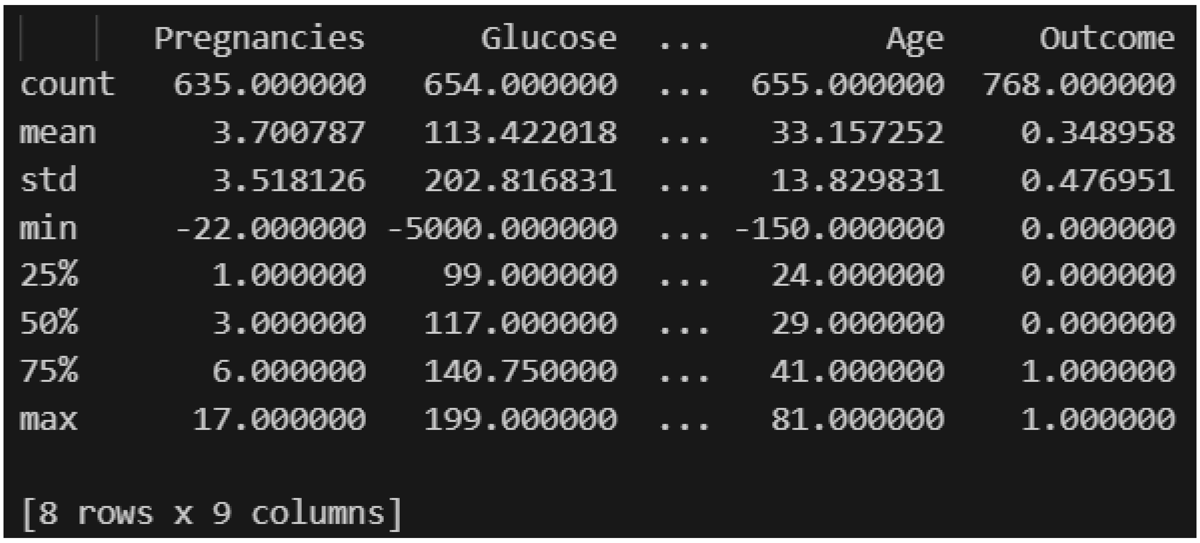
```
# a-1) overal structure of the data
# Display the information about the dataset
print(patients_data.info())
# Show descriptive statistics
print(patients_data.describe())
```

Info function's Output:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               635 non-null    float64
 1   Glucose                   654 non-null    float64
 2   BloodPressure             680 non-null    float64
 3   SkinThickness             624 non-null    float64
 4   Insulin                   680 non-null    float64
 5   BMI                       684 non-null    float64
 6   DiabetesPedigreeFunction  590 non-null    float64
 7   Age                       655 non-null    float64
 8   Outcome                   768 non-null    int64
dtypes: float64(8), int64(1)
memory usage: 54.1 KB
None
```

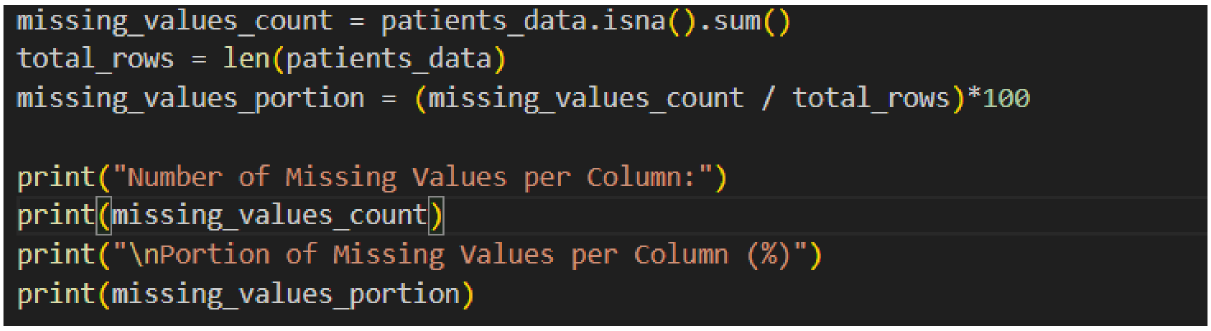
Describe Function's Output:



```
       Pregnancies       Glucose  ...         Age     Outcome
count   635.000000    654.000000  ...  655.000000  768.000000
mean      3.700787    113.422018  ...   33.157252    0.348958
std       3.518126    202.816831  ...   13.829831    0.476951
min     -22.000000  -5000.000000  ... -150.000000    0.000000
25%       1.000000     99.000000  ...   24.000000    0.000000
50%       3.000000    117.000000  ...   29.000000    0.000000
75%       6.000000    140.750000  ...   41.000000    1.000000
max      17.000000    199.000000  ...   81.000000    1.000000

[8 rows x 9 columns]
```

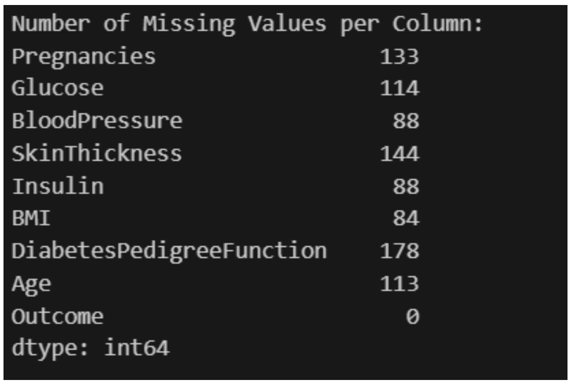
A-2) Missing Values

We used the sum function to calculate the number of empty cells in each row.

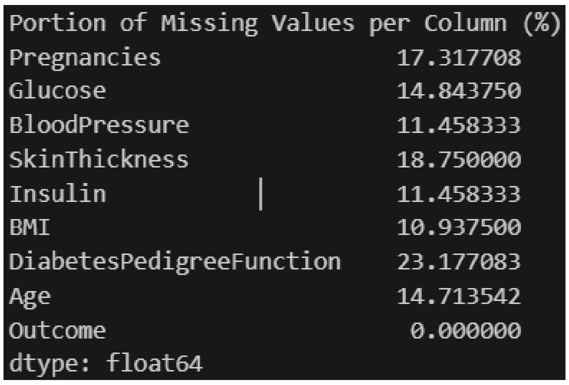


```python
missing_values_count = patients_data.isna().sum()
total_rows = len(patients_data)
missing_values_portion = (missing_values_count / total_rows)*100

print("Number of Missing Values per Column:")
print(missing_values_count)
print("\nPortion of Missing Values per Column (%)")
print(missing_values_portion)
```

Output:



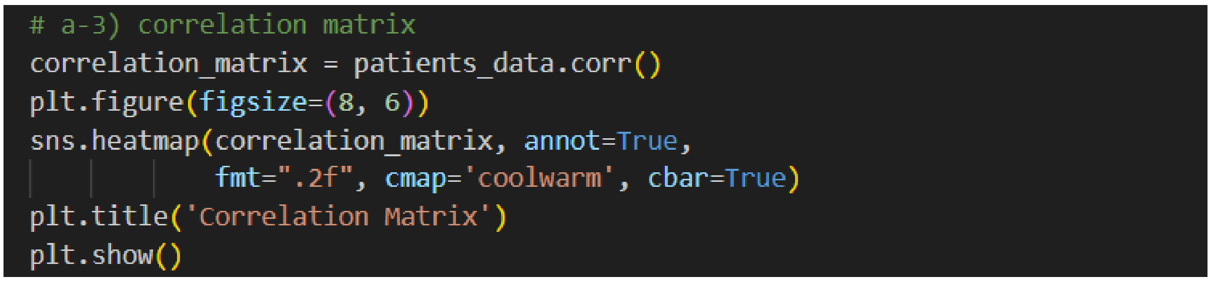
```
Number of Missing Values per Column:
Pregnancies                 133
Glucose                     114
BloodPressure                88
SkinThickness               144
Insulin                      88
BMI                          84
DiabetesPedigreeFunction    178
Age                         113
Outcome                       0
dtype: int64
```



```
Portion of Missing Values per Column (%)
Pregnancies                 17.317708
Glucose                     14.843750
BloodPressure               11.458333
SkinThickness               18.750000
Insulin                     11.458333
BMI                         10.937500
DiabetesPedigreeFunction    23.177083
Age                         14.713542
Outcome                      0.000000
dtype: float64
```

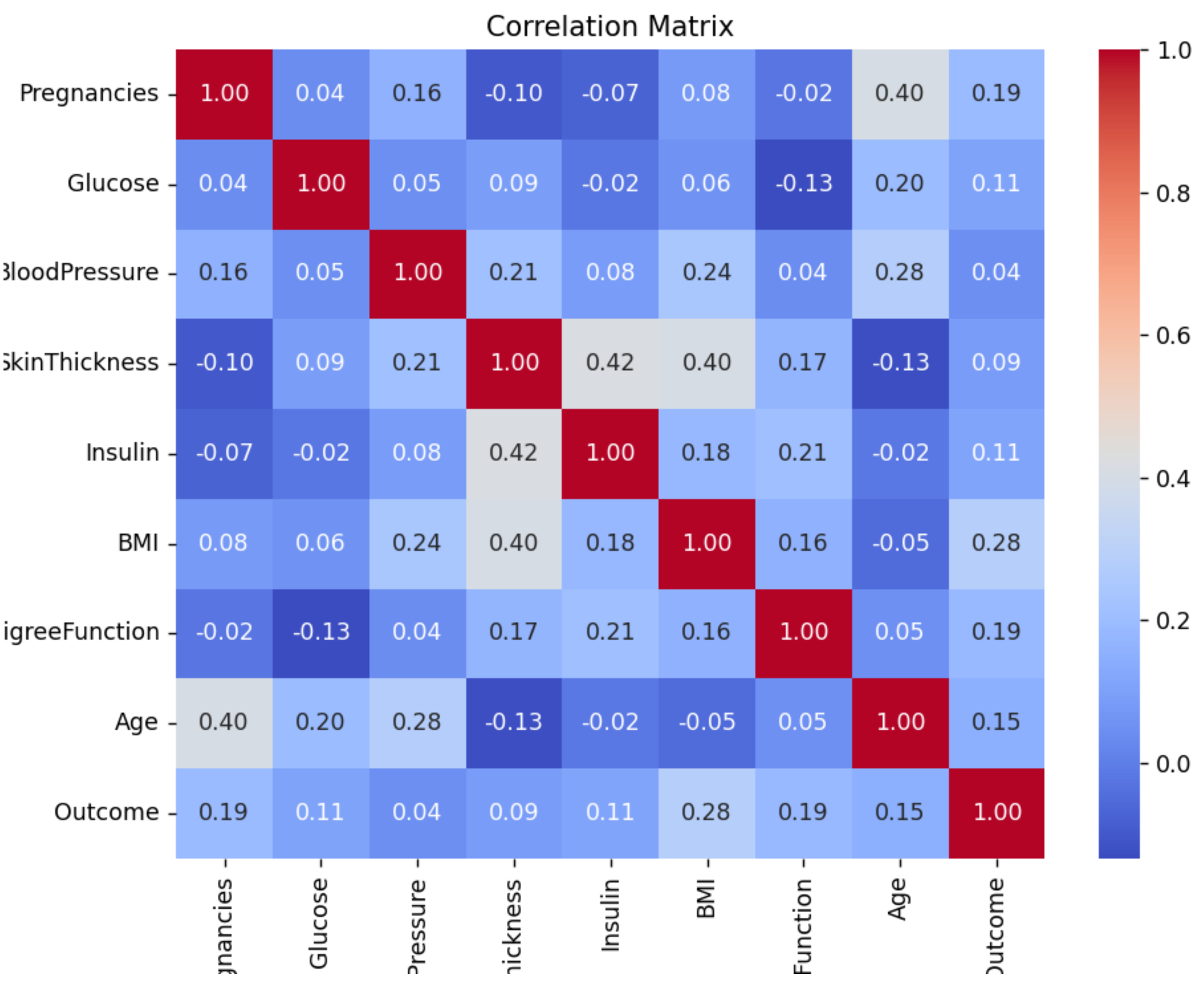
A-3) Correlation Matrix

Then, we calculated the correlation factor and showed its figure.



```
# a-3) correlation matrix
correlation_matrix = patients_data.corr()
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True,
            fmt=".2f", cmap='coolwarm', cbar=True)
plt.title('Correlation Matrix')
plt.show()
```

Output:



It is apparent that BMI (0.28), DiabetesPedigreeFunction (0.19), number of pregnancies (0.19), and Age (0.15) are of the most importance among other features.

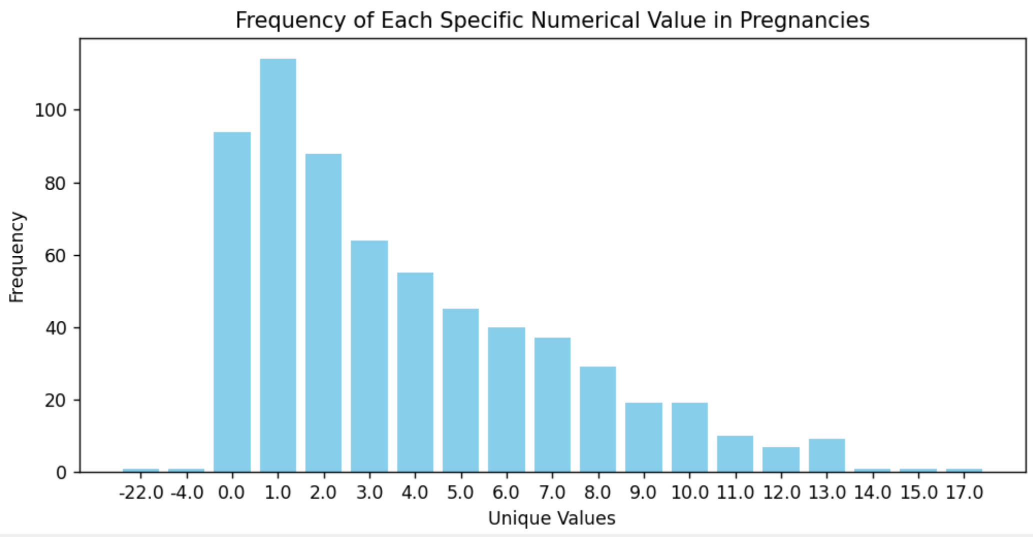
A-4) Number of Observations

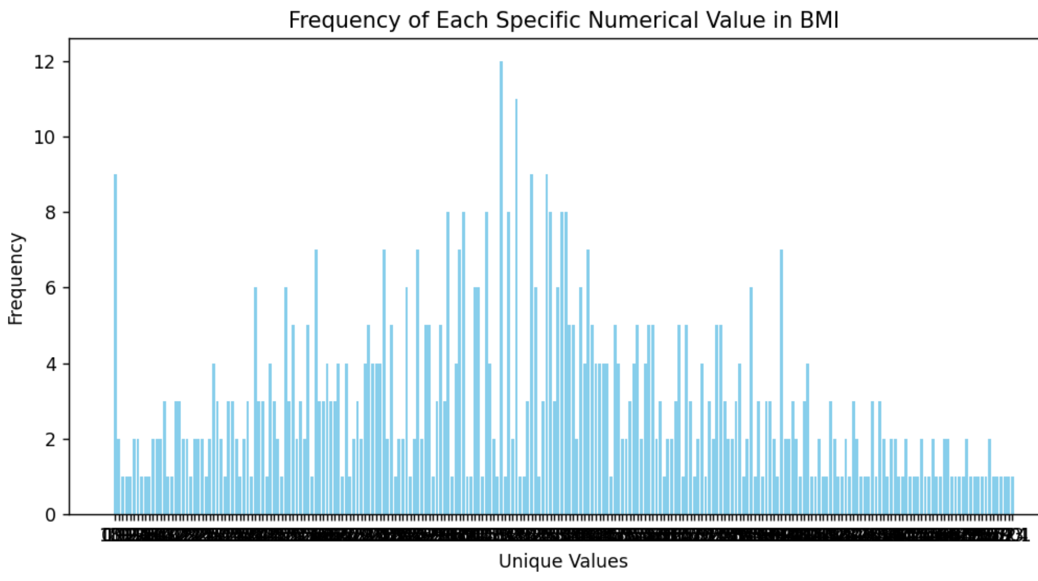
In question, it is stated to show the number of observations per each specific numerical value for four columns with the most importance. Since we need to obtain the number of frequencies for each unique number, the histogram of the data frame would not be of any help. Therefore, we use the value_counts function instead.

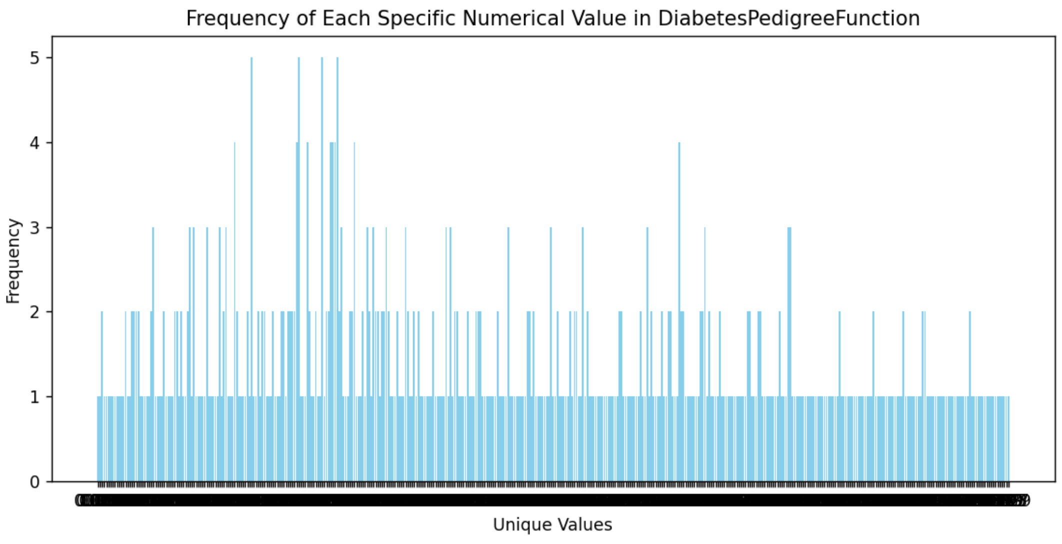
```
# a-4) Observations
# Specifing the columns of interest
cols = ['Pregnancies', 'BMI', 'DiabetesPedigreeFunction', 'Age']

# Creating figures for each column
for column in columns_of_interest:
    plt.figure()
    value_counts = patients_data[column].value_counts().sort_index()

    plt.bar(value_counts.index.astype(str),
            value_counts.values, color='skyblue')
    plt.title(f'Frequency of Each Specific Numerical Value in {column}')
    plt.xlabel('Unique Values')
    plt.ylabel('Frequency')
    plt.show()
```

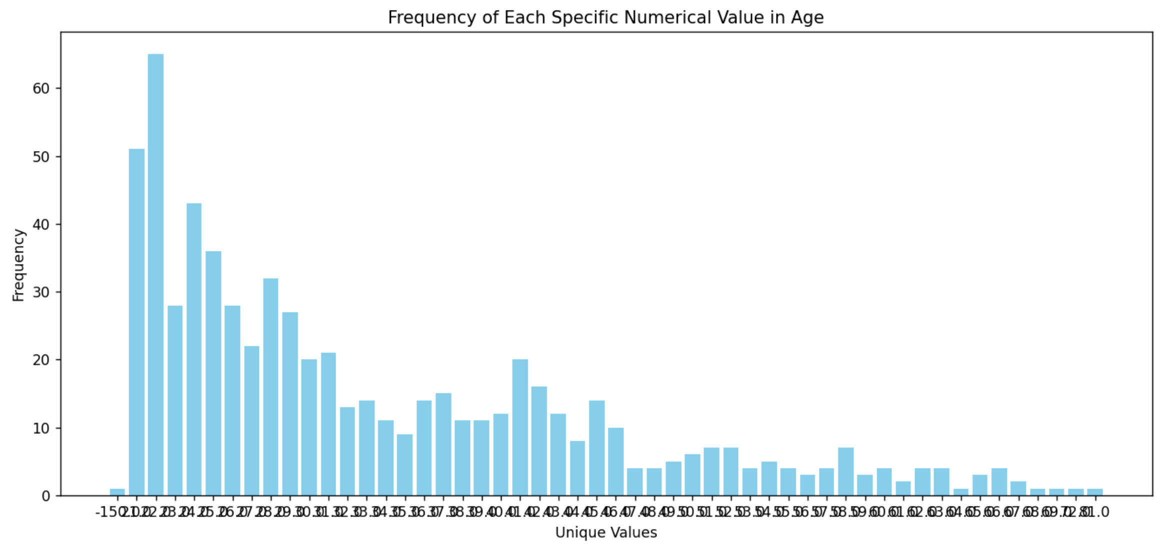
Since we wanted to obtain the number of repetitions for each number (not numbers in a specific range as in histogram), and there are numerous different values in the data frame, the x label of the plots may be not feasibly readable. This issue would not be problematic owing to the accessibility to the list of all the frequencies, in case of need.

Outputs:



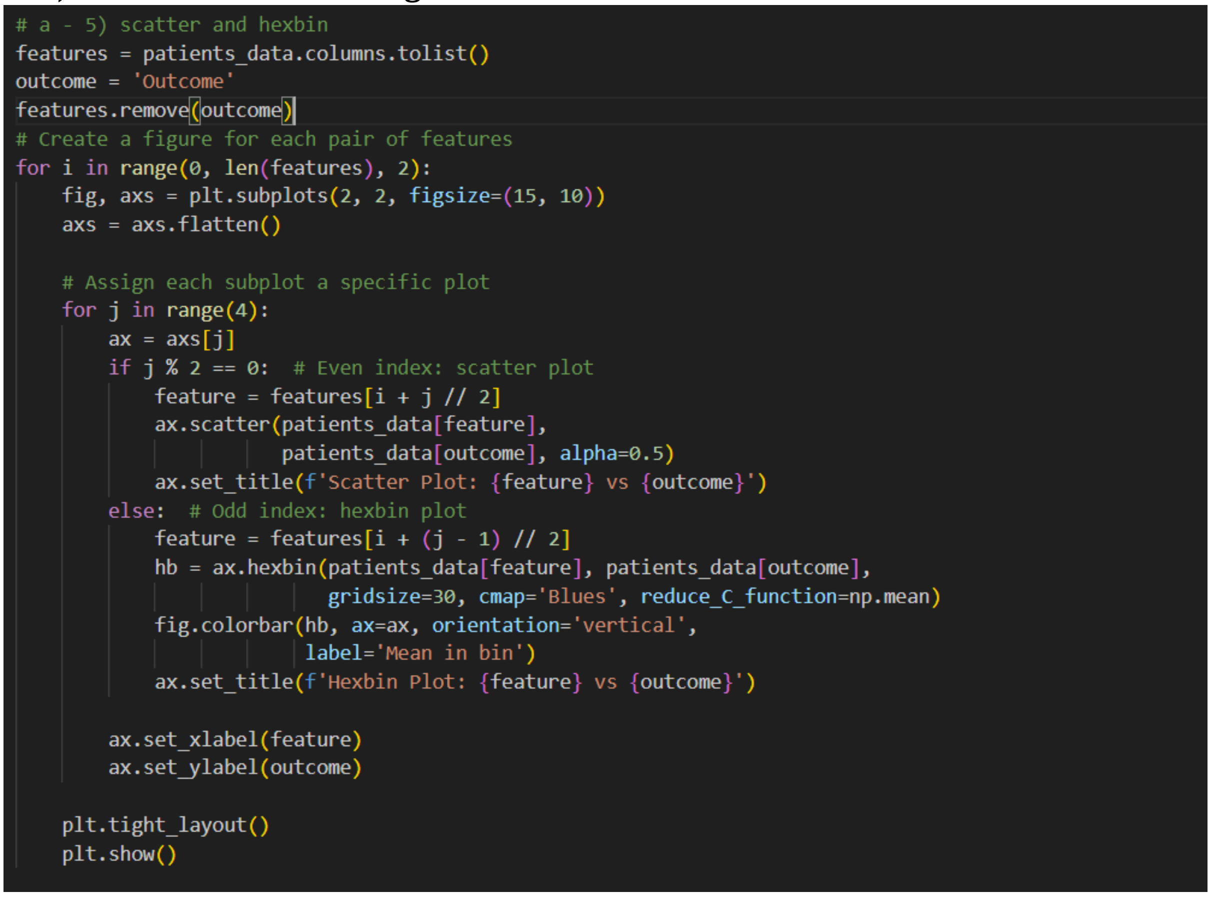

Frequency of Each Specific Numerical Value in BMI
Frequency
Unique Values
0
2
4
6
8
10
12


Frequency of Each Specific Numerical Value in DiabetesPedigreeFunction
Frequency
Unique Values
0
1
2
3
4
5



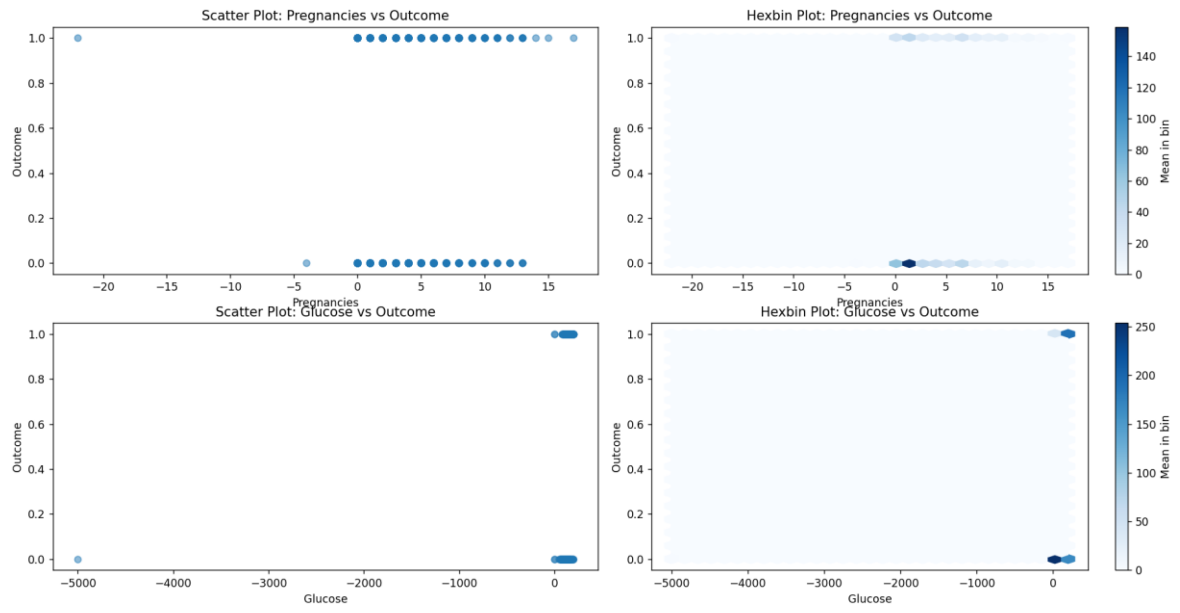
Note: The objective of this section was rather vague owing to stating “the number of observations per specific values”. We reported the number of frequencies for each different value in the data frame. If the objective was to show the histogram figures of the selected features, they are exhibited in section B-2.

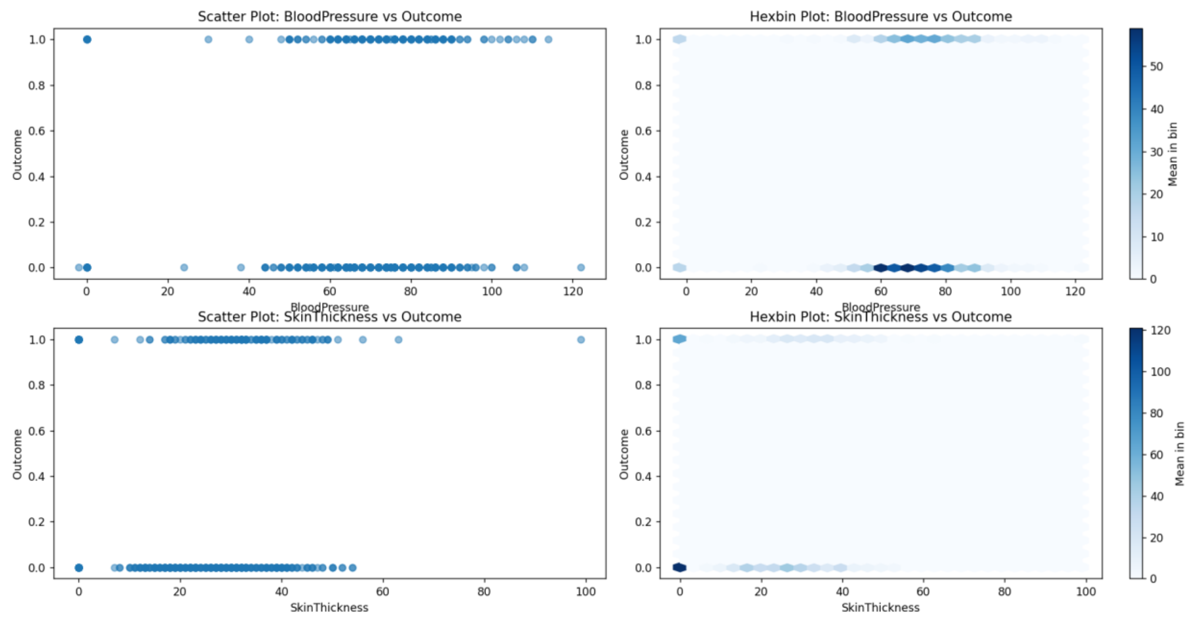
A-5) Scatter and Hexbin Figures

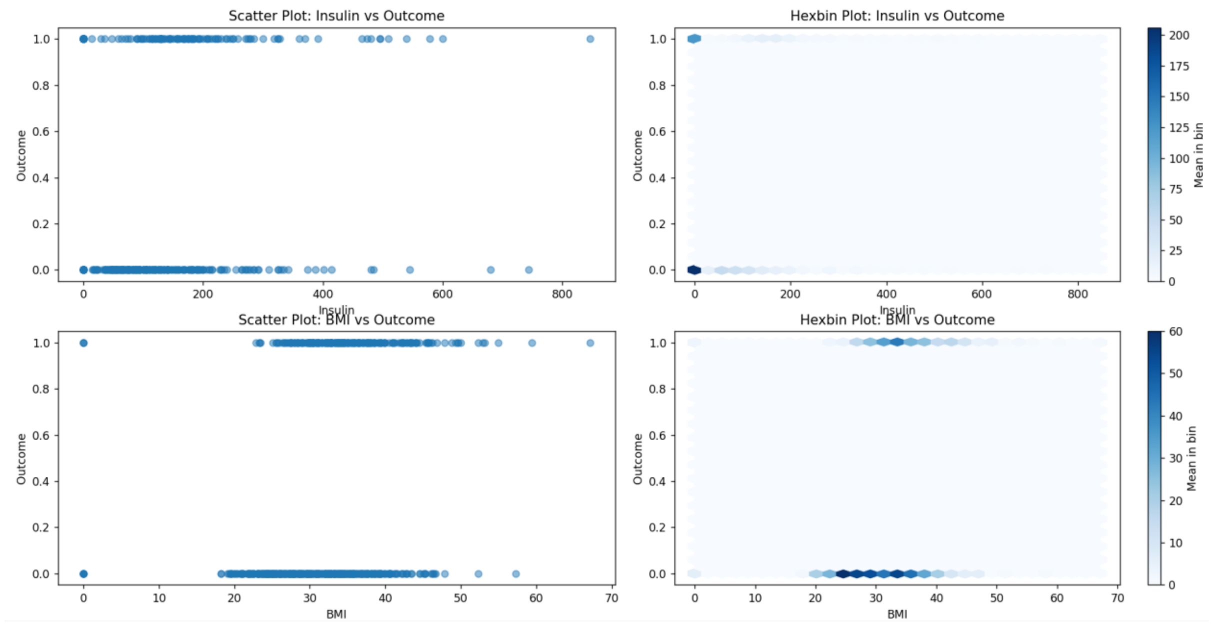


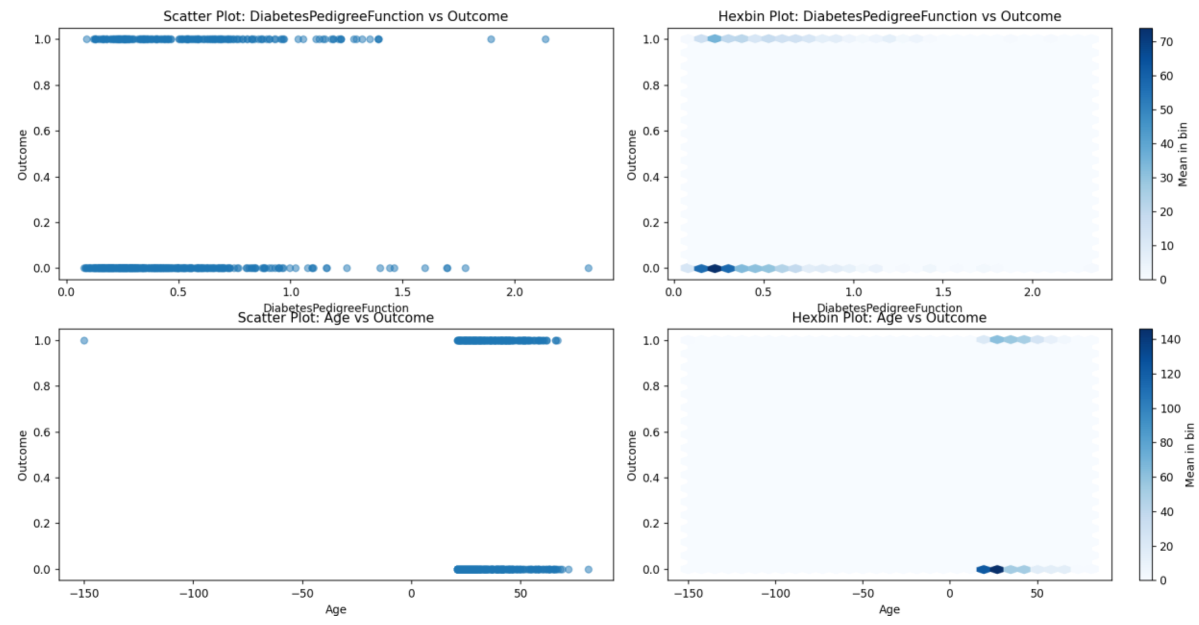
```
# a - 5) scatter and hexbin
features = patients_data.columns.tolist()
outcome = 'Outcome'
features.remove(outcome)
# Create a figure for each pair of features
for i in range(0, len(features), 2):
    fig, axs = plt.subplots(2, 2, figsize=(15, 10))
    axs = axs.flatten()

    # Assign each subplot a specific plot
    for j in range(4):
        ax = axs[j]
        if j % 2 == 0:  # Even index: scatter plot
            feature = features[i + j // 2]
            ax.scatter(patients_data[feature],
                       patients_data[outcome], alpha=0.5)
            ax.set_title(f'Scatter Plot: {feature} vs {outcome}')
        else:  # Odd index: hexbin plot
            feature = features[i + (j - 1) // 2]
            hb = ax.hexbin(patients_data[feature], patients_data[outcome],
                           gridsize=30, cmap='Blues', reduce_C_function=np.mean)
            fig.colorbar(hb, ax=ax, orientation='vertical',
                         label='Mean in bin')
            ax.set_title(f'Hexbin Plot: {feature} vs {outcome}')

        ax.set_xlabel(feature)
        ax.set_ylabel(outcome)

    plt.tight_layout()
    plt.show()
```

Outcome:


Scatter Plot: Pregnancies vs Outcome
Hexbin Plot: Pregnancies vs Outcome
Scatter Plot: Glucose vs Outcome
Hexbin Plot: Glucose vs Outcome
Outcome
Pregnancies
Glucose
Mean in bin


Scatter Plot: BloodPressure vs Outcome
Hexbin Plot: BloodPressure vs Outcome
Scatter Plot: SkinThickness vs Outcome
Hexbin Plot: SkinThickness vs Outcome
Outcome
BloodPressure
SkinThickness
Mean in bin

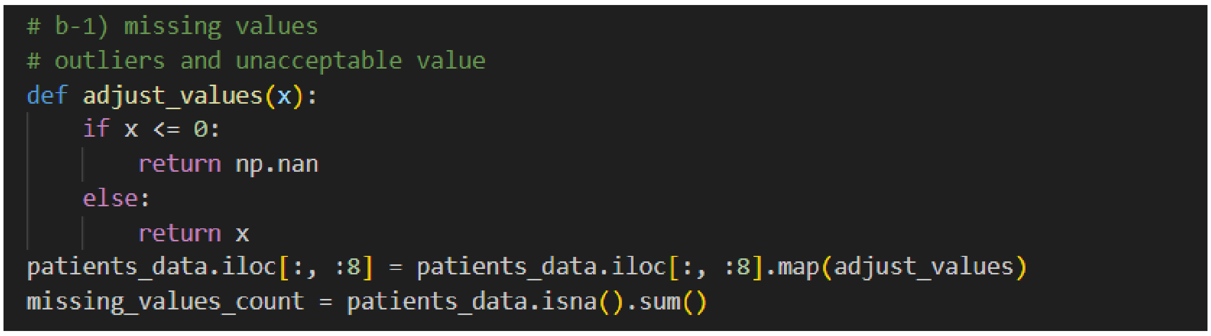

Scatter Plot: Insulin vs Outcome
Hexbin Plot: Insulin vs Outcome
Scatter Plot: BMI vs Outcome
Hexbin Plot: BMI vs Outcome
Outcome
Insulin
BMI
Mean in bin


Scatter Plot: DiabetesPedigreeFunction vs Outcome
Hexbin Plot: DiabetesPedigreeFunction vs Outcome
Scatter Plot: Age vs Outcome
Hexbin Plot: Age vs Outcome
Outcome
DiabetesPedigreeFunction
Age
Mean in bin

B) Preprocessing

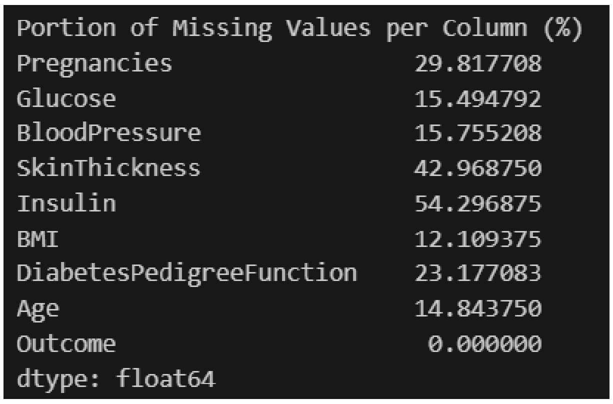
B-1) Missing Values

As can be seen from the previous visualizations, some data should be removed. For all columns, negative and zero values cannot be accepted. Hence, we remove them by assigning NaN values to them.



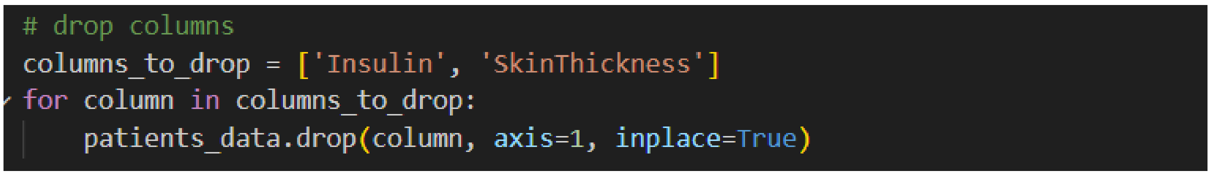
```python
# b-1) missing values
# outliers and unacceptable value
def adjust_values(x):
    if x <= 0:
        return np.nan
    else:
        return x
patients_data.iloc[:, :8] = patients_data.iloc[:, :8].map(adjust_values)
missing_values_count = patients_data.isna().sum()
```

Once again, like before, we calculate the number of missing values.



```
Portion of Missing Values per Column (%)
Pregnancies                 29.817708
Glucose                     15.494792
BloodPressure               15.755208
SkinThickness               42.968750
Insulin                     54.296875
BMI                         12.109375
DiabetesPedigreeFunction    23.177083
Age                         14.843750
Outcome                      0.000000
dtype: float64
```

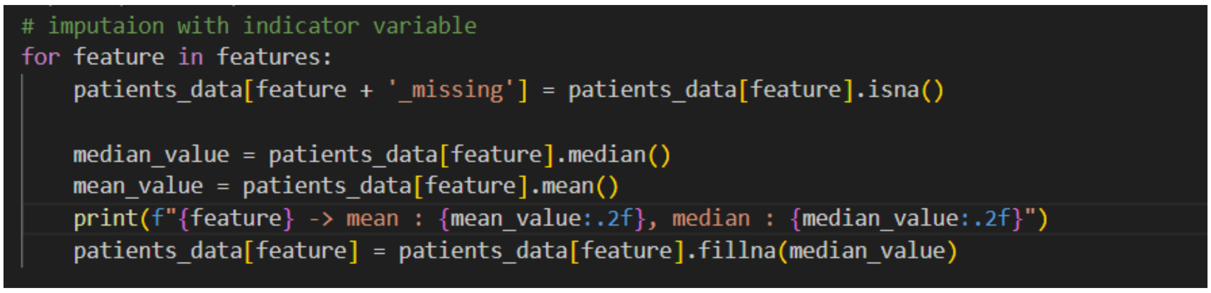
It is safe to say that perhaps, the SkinThickness and Insulin columns cannot help much with training our models due to the high portion of missing values. As a result, we remove them from our data frame.



```python
# drop columns
columns_to_drop = ['Insulin', 'SkinThickness']
for column in columns_to_drop:
    patients_data.drop(column, axis=1, inplace=True)
```

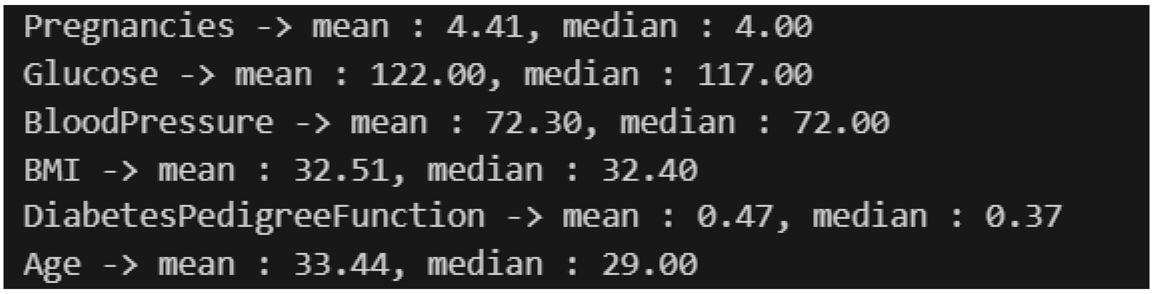
Now, that we have our data frame (containing 6 features) with some missing values, we opt for the imputation method with an indicator variable to mark values that did not exist before.

After, adding new columns for missing values, we have to assign a numerical datum to each empty cell. The two most popular options are the mean and the median of each column. Here we compared these two values.



```python
# imputaion with indicator variable
for feature in features:
    patients_data[feature + '_missing'] = patients_data[feature].isna()

    median_value = patients_data[feature].median()
    mean_value = patients_data[feature].mean()
    print(f"{feature} -> mean : {mean_value:.2f}, median : {median_value:.2f}")
    patients_data[feature] = patients_data[feature].fillna(median_value)
```

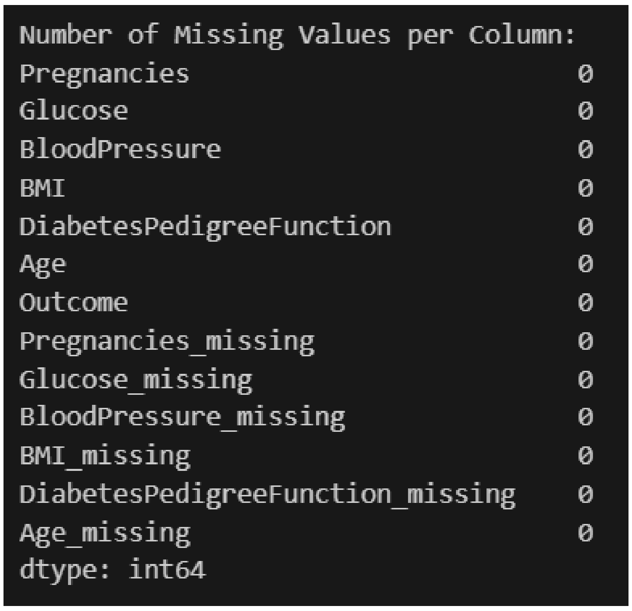
Comparison between median and mean:



```
Pregnancies -> mean : 4.41, median : 4.00
Glucose -> mean : 122.00, median : 117.00
BloodPressure -> mean : 72.30, median : 72.00
BMI -> mean : 32.51, median : 32.40
DiabetesPedigreeFunction -> mean : 0.47, median : 0.37
Age -> mean : 33.44, median : 29.00
```

Since no meaningful difference between these two statistical measurements was seen, we can be assured that assigning the median (or mean) of each column to its empty values probably would not cause any inconsistencies.

At this stage, to check our work, we counted the number of missing values. As expected, no column has any missing value anymore.



```
Number of Missing Values per Column:
Pregnancies                         0
Glucose                             0
BloodPressure                       0
BMI                                 0
DiabetesPedigreeFunction            0
Age                                 0
Outcome                             0
Pregnancies_missing                 0
Glucose_missing                     0
BloodPressure_missing               0
BMI_missing                         0
DiabetesPedigreeFunction_missing    0
Age_missing                         0
dtype: int64
```

B-2) Normalizing, and Standardizing

Normalizing and standardizing are two common preprocessing steps to prepare numerical data for analysis, especially when using machine learning algorithms that are sensitive to the scale of input data.

Standardizing, often called z-score normalization, transforms the data to have zero mean and a standard deviation of one. It's computed using the formula:

$$z = \frac{x - \mu}{\sigma}$$

where x is the original value, μ is the mean of the feature, and σ is the standard deviation. Standardizing does not bind values to a specific range, which might be necessary for some algorithms or methodologies.

Normalizing, often referred to as min-max scaling, rescales the data to a fixed range, typically 0 to 1, using the formula:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Where x_{min} and x_{max} are the minimum and maximum values of the feature, respectively. This transformation is sensitive to outliers because an outlier can compress the majority of the data into a very narrow range in the transformed space.

Standardizing is more commonly used in scenarios where the data needs to be normalized around the mean, removing the bias of different means and variances in the data. It’s beneficial for methods that assume data is normally distributed within each feature and work well even if the data isn't strictly normal.

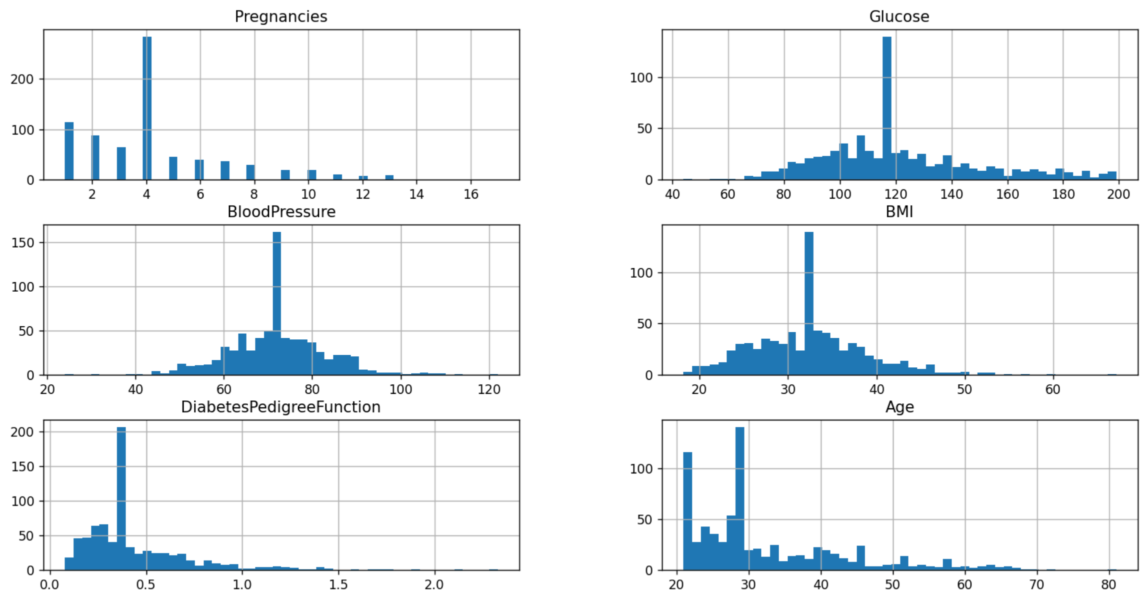
Normalizing is particularly useful when the data needs to be bound within a range, such as with neural networks that use Sigmoid or Tanh activation functions, and for algorithms that compute distances between data points.

To determine whether our data frame needs standardizing, normalizing, or neither, we have to examine the data distribution. If features are on different scales, standardizing can help equalize the influence of each feature. If outliers are a concern and should not disproportionately influence the model, we should consider robust scaling methods (like using median and quantile range instead of mean and standard deviation in standardization).

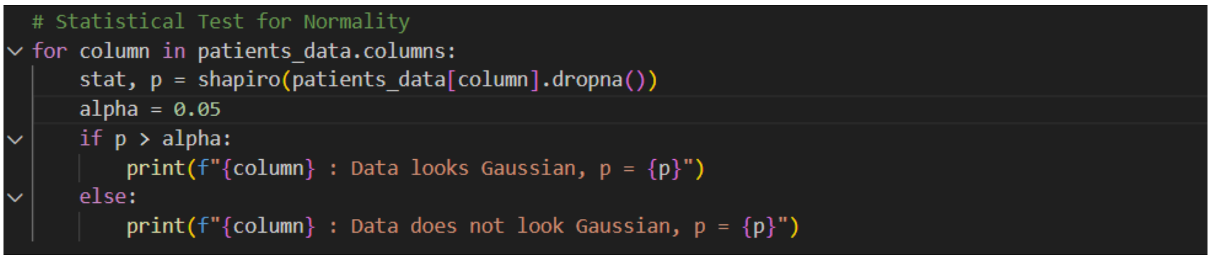
To do so, the first thing to investigate should be the histogram of the data set.

```
# b - 2)
# Data Distribution
patients_data[features].hist(bins=50, figsize=(20, 15))
plt.show()
```

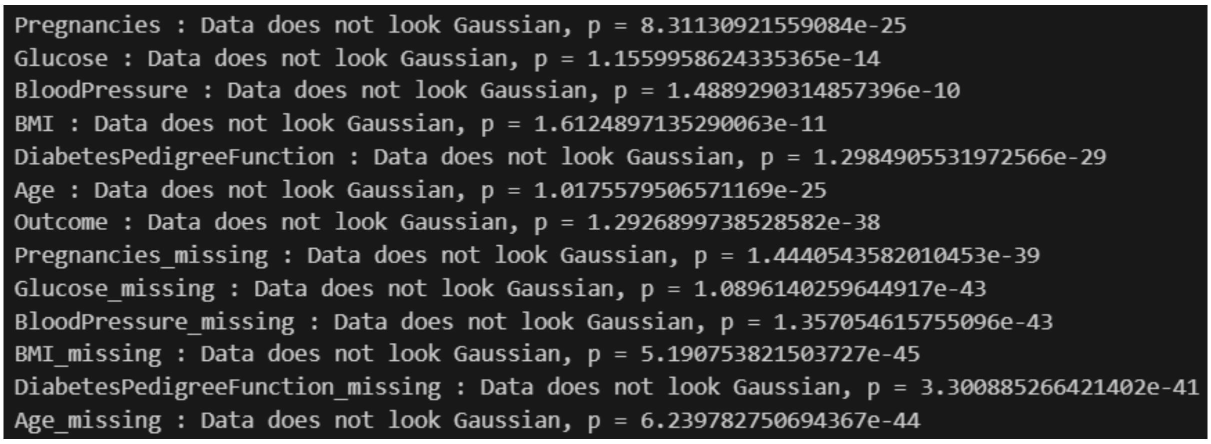
Output:



One of the most popular tests for normality of the data is the Shapiro-Wilk test. This test is used to assess whether a set of data is normally distributed. Also, this test is sensitive to NaN values. We set the significance level threshold at 5%. If the p-value of a data set is greater than the set threshold, we will have a rather Gaussian distribution.

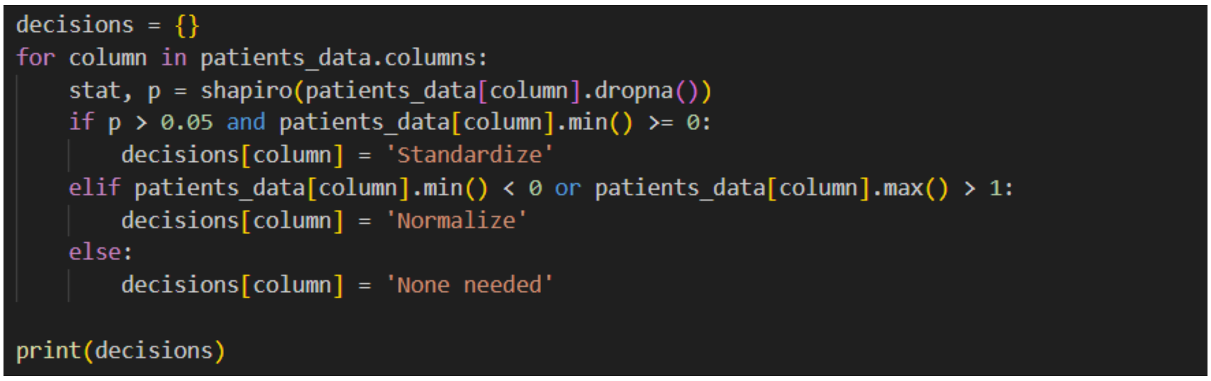

```python
# Statistical Test for Normality
for column in patients_data.columns:
    stat, p = shapiro(patients_data[column].dropna())
    alpha = 0.05
    if p > alpha:
        print(f"{column} : Data looks Gaussian, p = {p}")
    else:
        print(f"{column} : Data does not look Gaussian, p = {p}")
```

Output:


```
Pregnancies : Data does not look Gaussian, p = 8.31130921559084e-25
Glucose : Data does not look Gaussian, p = 1.1559958624335365e-14
BloodPressure : Data does not look Gaussian, p = 1.4889290314857396e-10
BMI : Data does not look Gaussian, p = 1.6124897135290063e-11
DiabetesPedigreeFunction : Data does not look Gaussian, p = 1.2984905531972566e-29
Age : Data does not look Gaussian, p = 1.0175579506571169e-25
Outcome : Data does not look Gaussian, p = 1.2926899738528582e-38
Pregnancies_missing : Data does not look Gaussian, p = 1.4440543582010453e-39
Glucose_missing : Data does not look Gaussian, p = 1.0896140259644917e-43
BloodPressure_missing : Data does not look Gaussian, p = 1.3570546157755096e-43
BMI_missing : Data does not look Gaussian, p = 5.190753821503727e-45
DiabetesPedigreeFunction_missing : Data does not look Gaussian, p = 3.300885266421402e-41
Age_missing : Data does not look Gaussian, p = 6.239782750694367e-44
```

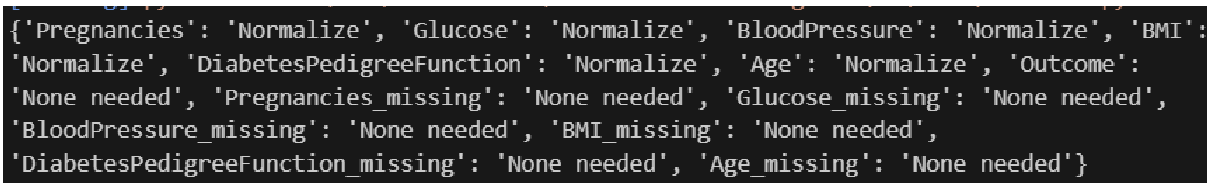
This test alone is not sufficient to deduct if normalizing or standardizing is needed. However, if we look at our results, it can be easily interpreted that the scale of the values of each feature differs significantly. Resultantly, normalizing our data can be a big help.

Also, in the code below, we determined which set needs to be normalized. We expect the distribution of the missing indicator columns to need nothing, as they only contain True (1) and False (0).



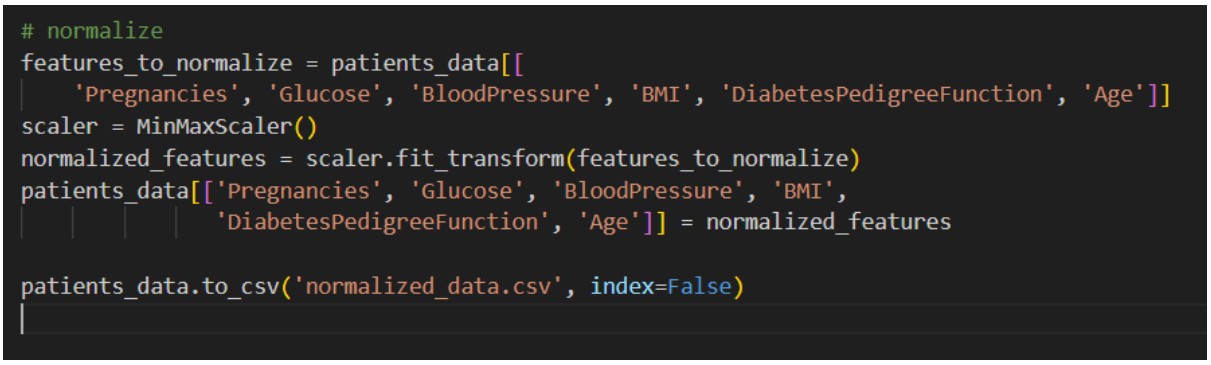
```
decisions = {}
for column in patients_data.columns:
    stat, p = shapiro(patients_data[column].dropna())
    if p > 0.05 and patients_data[column].min() >= 0:
        decisions[column] = 'Standardize'
    elif patients_data[column].min() < 0 or patients_data[column].max() > 1:
        decisions[column] = 'Normalize'
    else:
        decisions[column] = 'None needed'

print(decisions)
```

Output:



```
{'Pregnancies': 'Normalize', 'Glucose': 'Normalize', 'BloodPressure': 'Normalize', 'BMI':
'Normalize', 'DiabetesPedigreeFunction': 'Normalize', 'Age': 'Normalize', 'Outcome':
'None needed', 'Pregnancies_missing': 'None needed', 'Glucose_missing': 'None needed',
'BloodPressure_missing': 'None needed', 'BMI_missing': 'None needed',
'DiabetesPedigreeFunction_missing': 'None needed', 'Age_missing': 'None needed'}
```

Hence, we need to normalize all the features (6 columns) except the indicators.



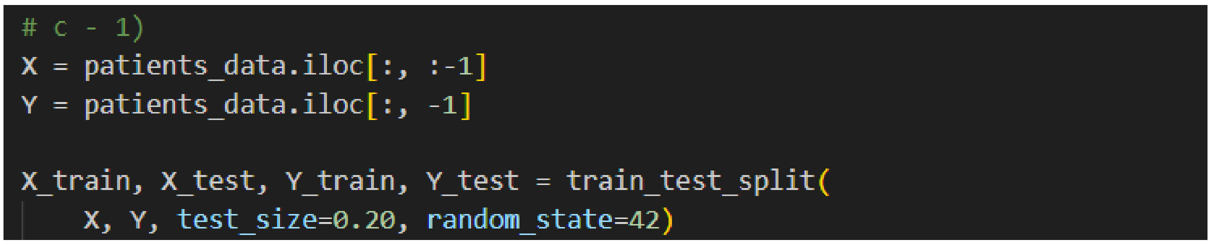
```
# normalize
features_to_normalize = patients_data[[
    'Pregnancies', 'Glucose', 'BloodPressure', 'BMI', 'DiabetesPedigreeFunction', 'Age']]
scaler = MinMaxScaler()
normalized_features = scaler.fit_transform(features_to_normalize)
patients_data[['Pregnancies', 'Glucose', 'BloodPressure', 'BMI',
               'DiabetesPedigreeFunction', 'Age']] = normalized_features

patients_data.to_csv('normalized_data.csv', index=False)
```

At this stage, our data frame only has values between 0 and 1.

C) Selection, Training, and Evaluation

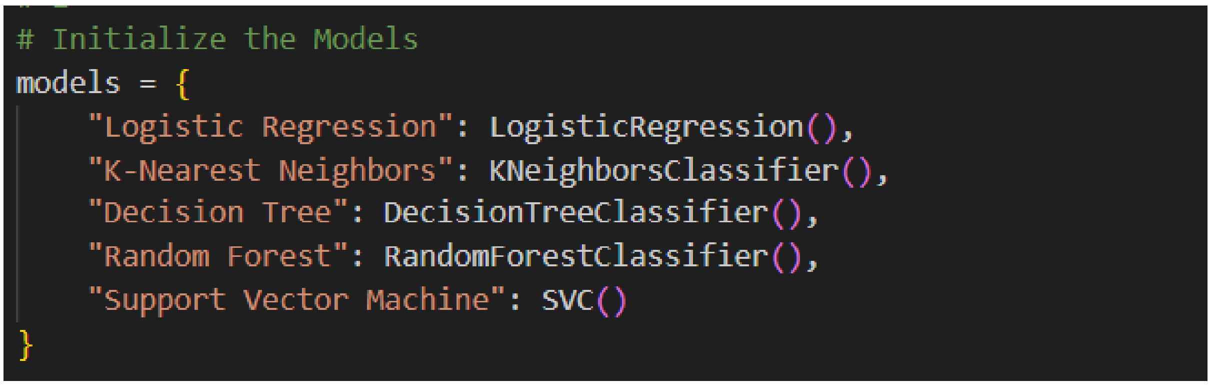
C-1, C-2) Split and Train

In this section we are asked to randomly split our data frame into a train (80%) and test (20%) set. Then by using the sci-kit-learn module train 5 models: Logistic Regression (LR), K-Nearest-Neighbors (KNN), Decision Tree (DT), Random Forest (RF), and Support Vector Machine (SVM). First, we split the data:



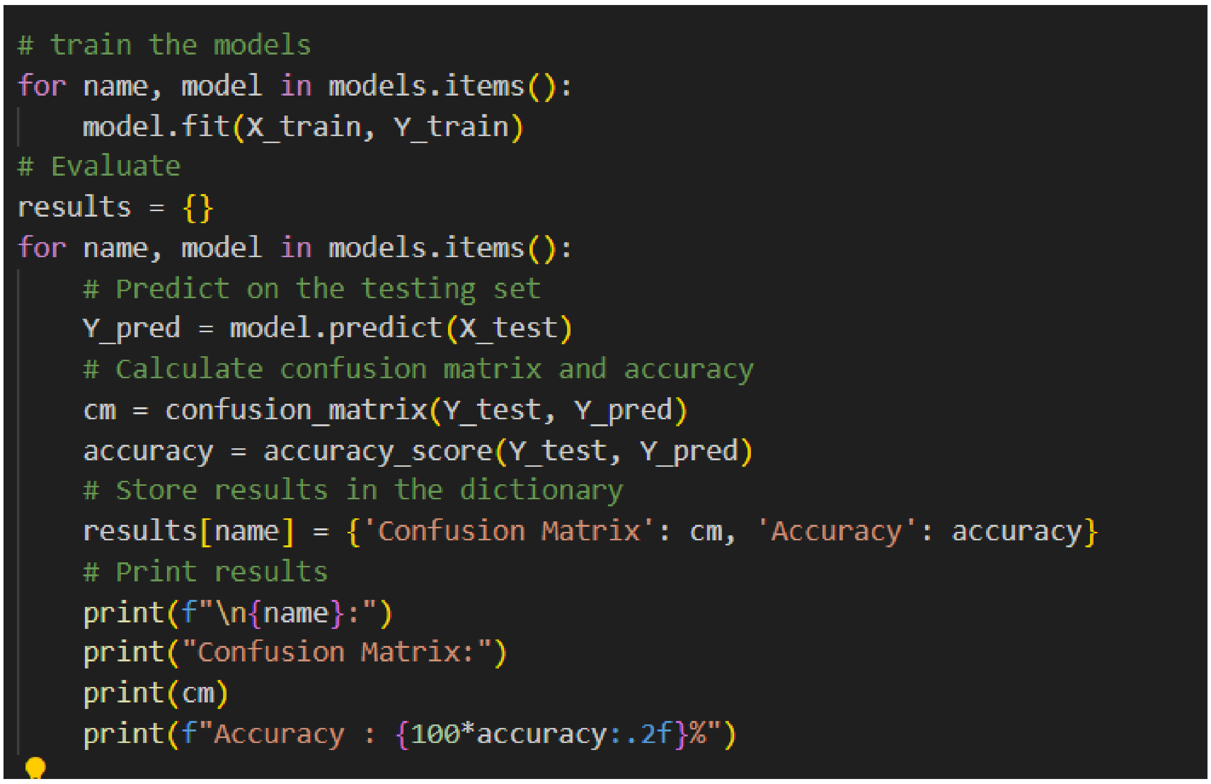
```python
# c - 1)
X = patients_data.iloc[:, :-1]
Y = patients_data.iloc[:, -1]

X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.20, random_state=42)
```

Then we initialize the models into a dictionary to loop through:



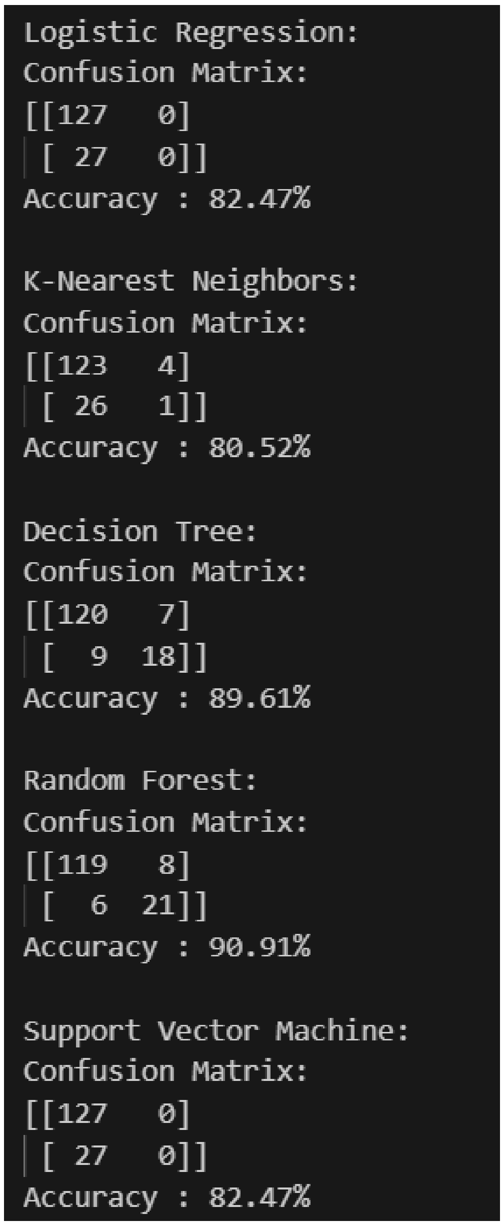
```python
# Initialize the Models
models = {
    "Logistic Regression": LogisticRegression(),
    "K-Nearest Neighbors": KNeighborsClassifier(),
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "Support Vector Machine": SVC()
}
```

Then we train and evaluate the models. Also, we calculate the confusion matrix and accuracy for each model.



```python
# train the models
for name, model in models.items():
    model.fit(X_train, Y_train)
# Evaluate
results = {}
for name, model in models.items():
    # Predict on the testing set
    Y_pred = model.predict(X_test)
    # Calculate confusion matrix and accuracy
    cm = confusion_matrix(Y_test, Y_pred)
    accuracy = accuracy_score(Y_test, Y_pred)
    # Store results in the dictionary
    results[name] = {'Confusion Matrix': cm, 'Accuracy': accuracy}
    # Print results
    print(f"\n{name}:")
    print("Confusion Matrix:")
    print(cm)
    print(f"Accuracy : {100*accuracy:.2f}%")
```

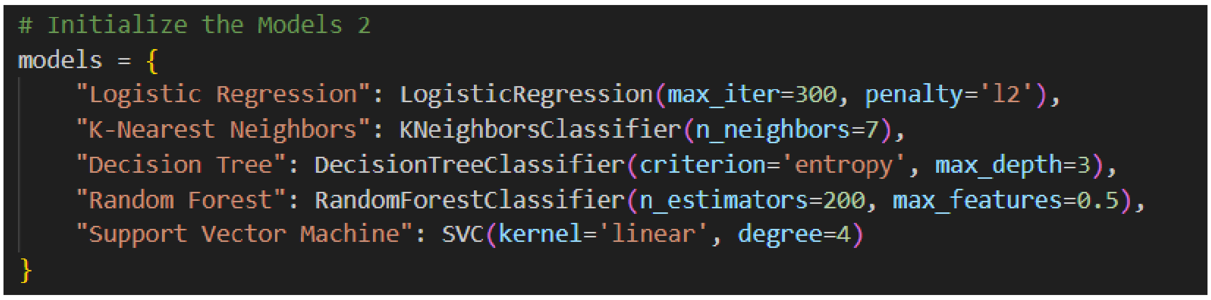
Output:



```
Logistic Regression:
Confusion Matrix:
[[127   0]
 [ 27   0]]
Accuracy : 82.47%

K-Nearest Neighbors:
Confusion Matrix:
[[123   4]
 [ 26   1]]
Accuracy : 80.52%

Decision Tree:
Confusion Matrix:
[[120   7]
 [  9  18]]
Accuracy : 89.61%

Random Forest:
Confusion Matrix:
[[119   8]
 [  6  21]]
Accuracy : 90.91%

Support Vector Machine:
Confusion Matrix:
[[127   0]
 [ 27   0]]
Accuracy : 82.47%
```

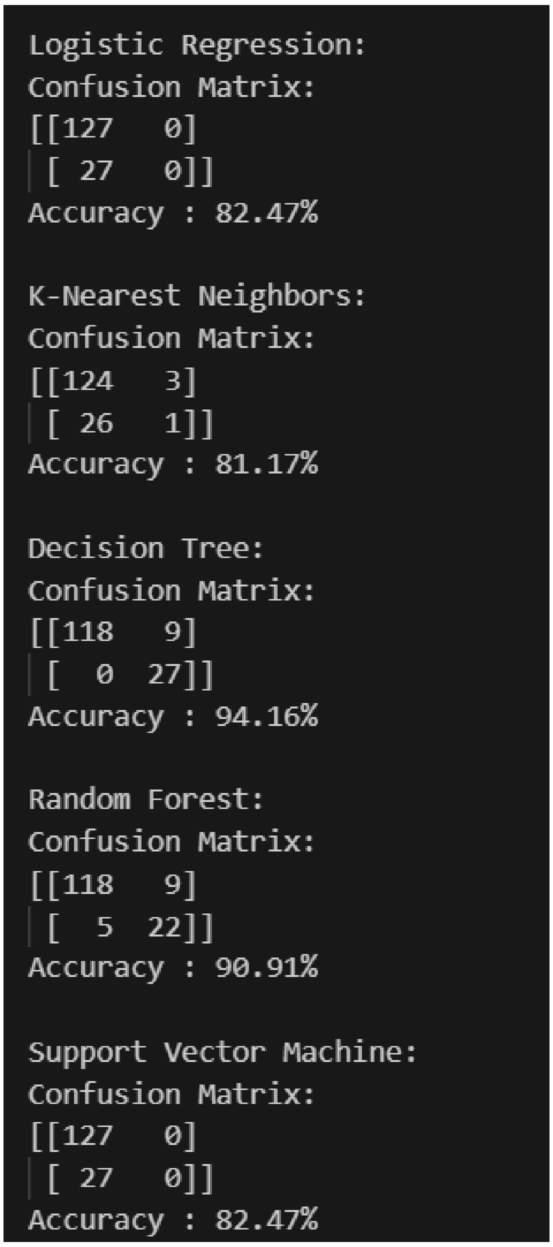
C-3) Change Hyperparameters and Optimization

First, we changed two parameters of each model, except for the KNN model in which we were asked to only change the number of neighbors. The only difference between this section and the former one is in the initializing part:



```
# Initialize the Models 2
models = {
    "Logistic Regression": LogisticRegression(max_iter=300, penalty='l2'),
    "K-Nearest Neighbors": KNeighborsClassifier(n_neighbors=7),
    "Decision Tree": DecisionTreeClassifier(criterion='entropy', max_depth=3),
    "Random Forest": RandomForestClassifier(n_estimators=200, max_features=0.5),
    "Support Vector Machine": SVC(kernel='linear', degree=4)
}
```

The output differed as well, as we expected:



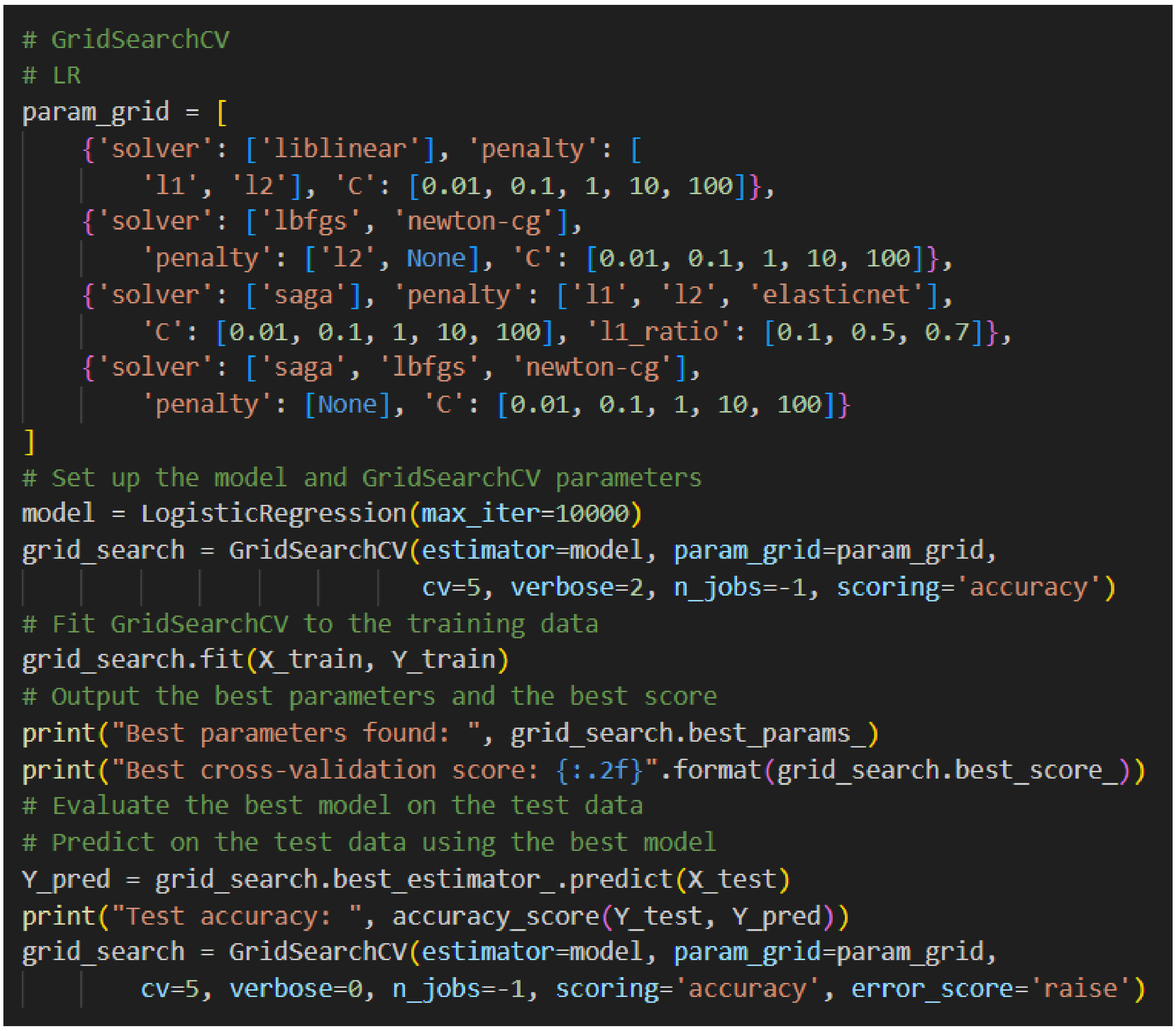
```
Logistic Regression:
Confusion Matrix:
[[127   0]
 [ 27   0]]
Accuracy : 82.47%

K-Nearest Neighbors:
Confusion Matrix:
[[124   3]
 [ 26   1]]
Accuracy : 81.17%

Decision Tree:
Confusion Matrix:
[[118   9]
 [  0  27]]
Accuracy : 94.16%

Random Forest:
Confusion Matrix:
[[118   9]
 [  5  22]]
Accuracy : 90.91%

Support Vector Machine:
Confusion Matrix:
[[127   0]
 [ 27   0]]
Accuracy : 82.47%
```

Hence, our models experienced no changes in accuracy for 3 of the models (LR, Rf, and SVM). However, the other two (KNN, and DT) got slightly more accurate.

Now, we go through the process of optimizing.

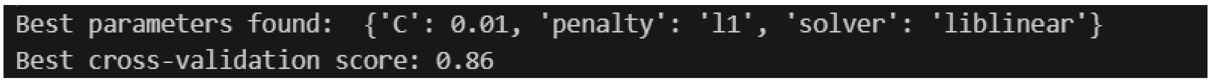
We have done this section for each model separately.

Linear Regression:



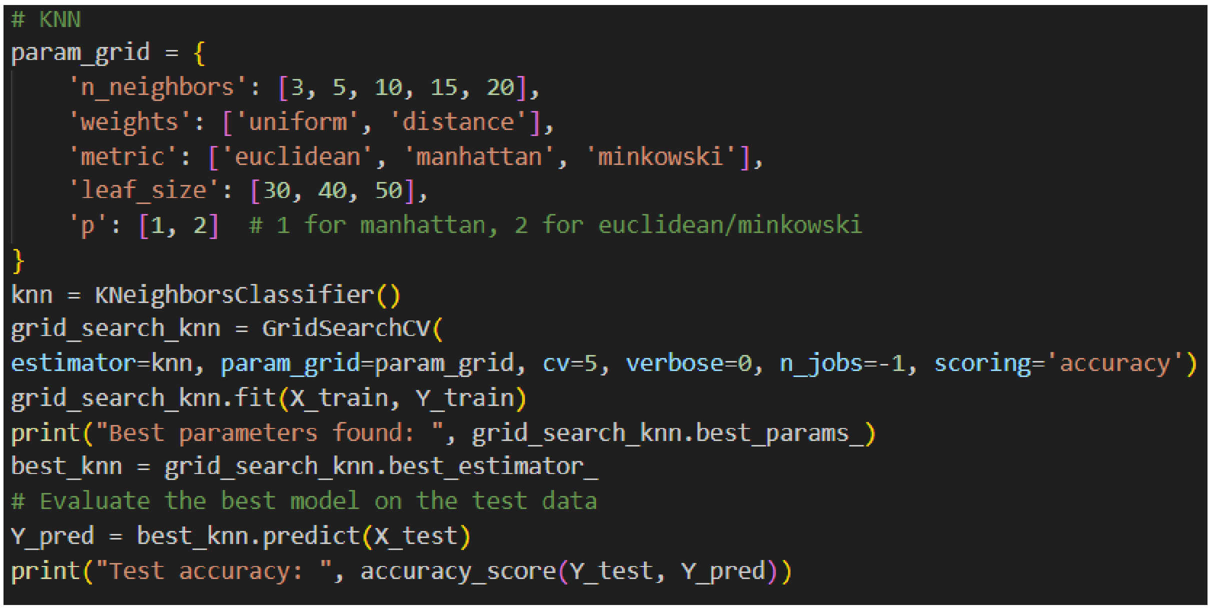
```
# GridSearchCV
# LR
param_grid = [
    {'solver': ['liblinear'], 'penalty': [
        'l1', 'l2'], 'C': [0.01, 0.1, 1, 10, 100]},
    {'solver': ['lbfgs', 'newton-cg'],
        'penalty': ['l2', None], 'C': [0.01, 0.1, 1, 10, 100]},
    {'solver': ['saga'], 'penalty': ['l1', 'l2', 'elasticnet'],
        'C': [0.01, 0.1, 1, 10, 100], 'l1_ratio': [0.1, 0.5, 0.7]},
    {'solver': ['saga', 'lbfgs', 'newton-cg'],
        'penalty': [None], 'C': [0.01, 0.1, 1, 10, 100]}
]
# Set up the model and GridSearchCV parameters
model = LogisticRegression(max_iter=10000)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
                           cv=5, verbose=2, n_jobs=-1, scoring='accuracy')
# Fit GridSearchCV to the training data
grid_search.fit(X_train, Y_train)
# Output the best parameters and the best score
print("Best parameters found: ", grid_search.best_params_)
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
# Evaluate the best model on the test data
# Predict on the test data using the best model
Y_pred = grid_search.best_estimator_.predict(X_test)
print("Test accuracy: ", accuracy_score(Y_test, Y_pred))
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
        cv=5, verbose=0, n_jobs=-1, scoring='accuracy', error_score='raise')
```

Output:



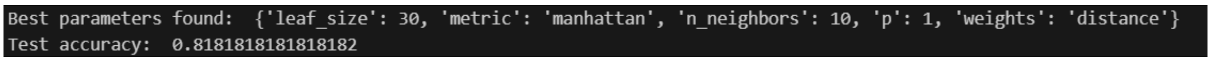
```
Best parameters found:  {'C': 0.01, 'penalty': 'l1', 'solver': 'liblinear'}
Best cross-validation score: 0.86
```

K-Nearest-Neighbor:



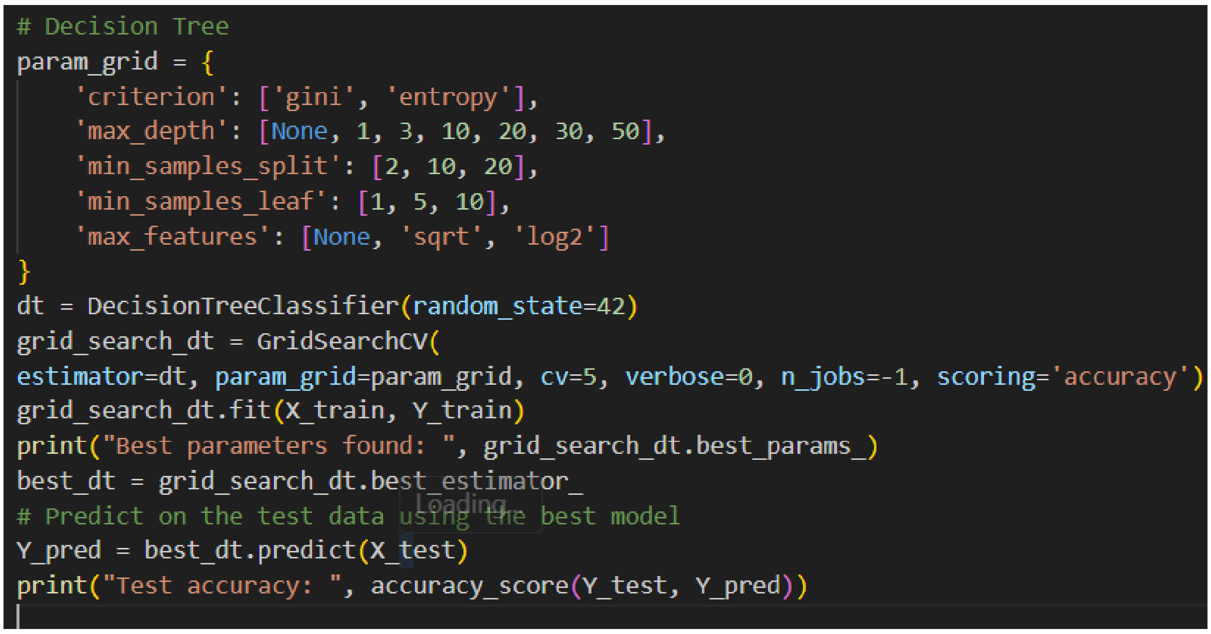
```python
# KNN
param_grid = {
    'n_neighbors': [3, 5, 10, 15, 20],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski'],
    'leaf_size': [30, 40, 50],
    'p': [1, 2]  # 1 for manhattan, 2 for euclidean/minkowski
}
knn = KNeighborsClassifier()
grid_search_knn = GridSearchCV(
estimator=knn, param_grid=param_grid, cv=5, verbose=0, n_jobs=-1, scoring='accuracy')
grid_search_knn.fit(X_train, Y_train)
print("Best parameters found: ", grid_search_knn.best_params_)
best_knn = grid_search_knn.best_estimator_
# Evaluate the best model on the test data
Y_pred = best_knn.predict(X_test)
print("Test accuracy: ", accuracy_score(Y_test, Y_pred))
```

Output:



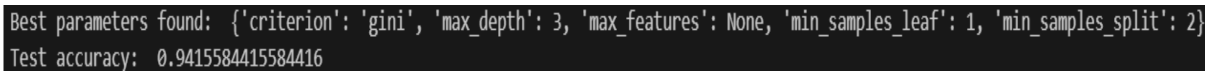
```
Best parameters found:  {'leaf_size': 30, 'metric': 'manhattan', 'n_neighbors': 10, 'p': 1, 'weights': 'distance'}
Test accuracy:  0.8181818181818182
```

Decision Tree:



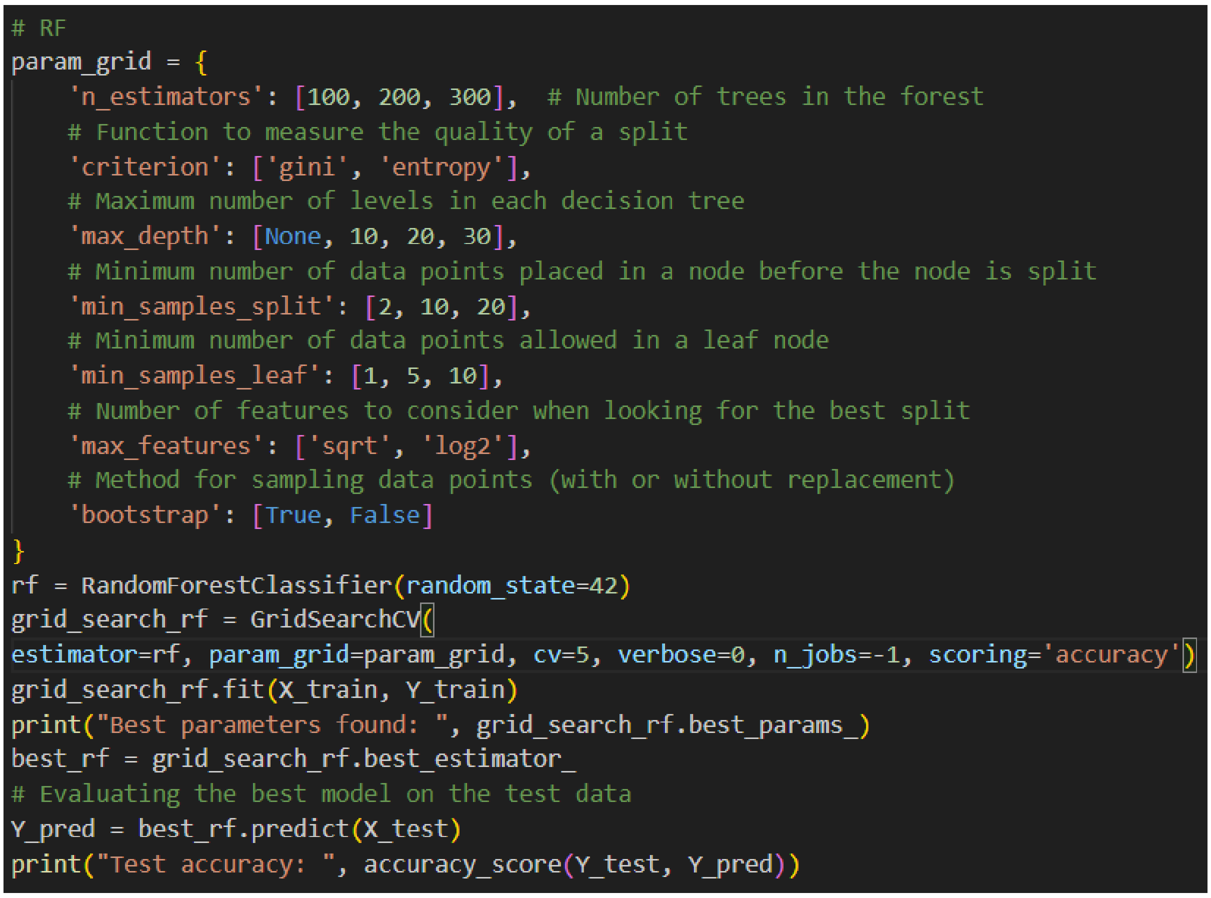
```python
# Decision Tree
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 1, 3, 10, 20, 30, 50],
    'min_samples_split': [2, 10, 20],
    'min_samples_leaf': [1, 5, 10],
    'max_features': [None, 'sqrt', 'log2']
}
dt = DecisionTreeClassifier(random_state=42)
grid_search_dt = GridSearchCV(
estimator=dt, param_grid=param_grid, cv=5, verbose=0, n_jobs=-1, scoring='accuracy')
grid_search_dt.fit(X_train, Y_train)
print("Best parameters found: ", grid_search_dt.best_params_)
best_dt = grid_search_dt.best_estimator_
# Predict on the test data using the best model
Y_pred = best_dt.predict(X_test)
print("Test accuracy: ", accuracy_score(Y_test, Y_pred))
```

Output:



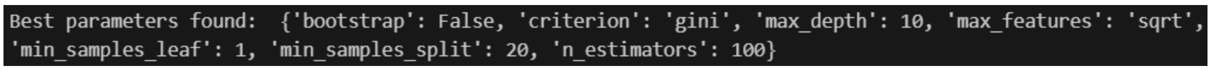
```
Best parameters found:  {'criterion': 'gini', 'max_depth': 3, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
Test accuracy:  0.9415584415584416
```

Random Forest:



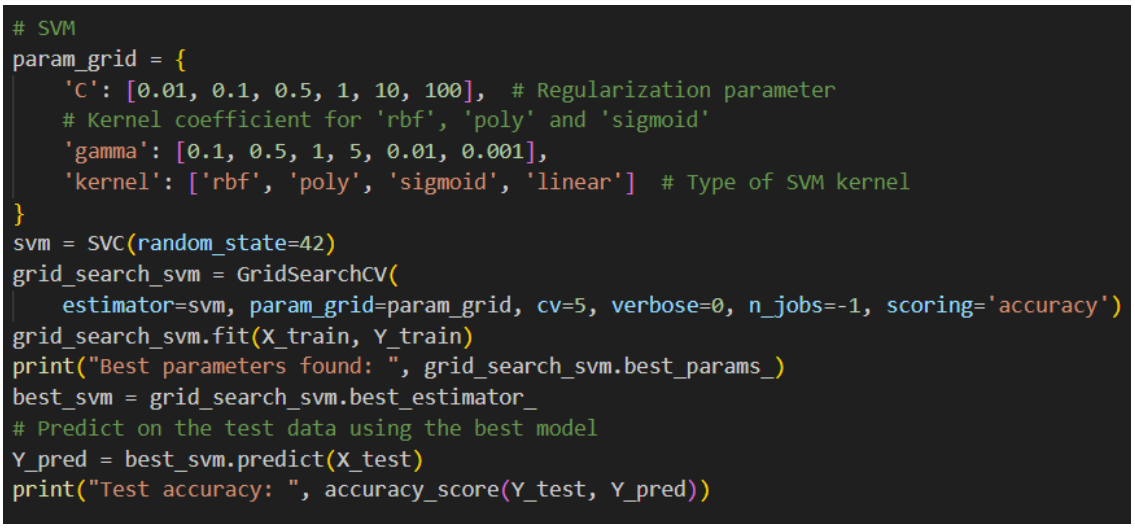
```
# RF
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    # Function to measure the quality of a split
    'criterion': ['gini', 'entropy'],
    # Maximum number of levels in each decision tree
    'max_depth': [None, 10, 20, 30],
    # Minimum number of data points placed in a node before the node is split
    'min_samples_split': [2, 10, 20],
    # Minimum number of data points allowed in a leaf node
    'min_samples_leaf': [1, 5, 10],
    # Number of features to consider when looking for the best split
    'max_features': ['sqrt', 'log2'],
    # Method for sampling data points (with or without replacement)
    'bootstrap': [True, False]
}
rf = RandomForestClassifier(random_state=42)
grid_search_rf = GridSearchCV(
estimator=rf, param_grid=param_grid, cv=5, verbose=0, n_jobs=-1, scoring='accuracy')
grid_search_rf.fit(X_train, Y_train)
print("Best parameters found: ", grid_search_rf.best_params_)
best_rf = grid_search_rf.best_estimator_
# Evaluating the best model on the test data
Y_pred = best_rf.predict(X_test)
print("Test accuracy: ", accuracy_score(Y_test, Y_pred))
```

Output:



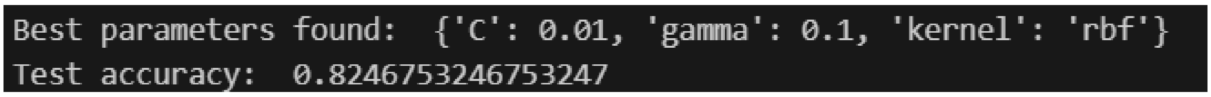
```
Best parameters found:  {'bootstrap': False, 'criterion': 'gini', 'max_depth': 10, 'max_features': 'sqrt',
'min_samples_leaf': 1, 'min_samples_split': 20, 'n_estimators': 100}
```

SVM:



```
# SVM
param_grid = {
    'C': [0.01, 0.1, 0.5, 1, 10, 100],  # Regularization parameter
    # Kernel coefficient for 'rbf', 'poly' and 'sigmoid'
    'gamma': [0.1, 0.5, 1, 5, 0.01, 0.001],
    'kernel': ['rbf', 'poly', 'sigmoid', 'linear']  # Type of SVM kernel
}
svm = SVC(random_state=42)
grid_search_svm = GridSearchCV(
    estimator=svm, param_grid=param_grid, cv=5, verbose=0, n_jobs=-1, scoring='accuracy')
grid_search_svm.fit(X_train, Y_train)
print("Best parameters found: ", grid_search_svm.best_params_)
best_svm = grid_search_svm.best_estimator_
# Predict on the test data using the best model
Y_pred = best_svm.predict(X_test)
print("Test accuracy: ", accuracy_score(Y_test, Y_pred))
```

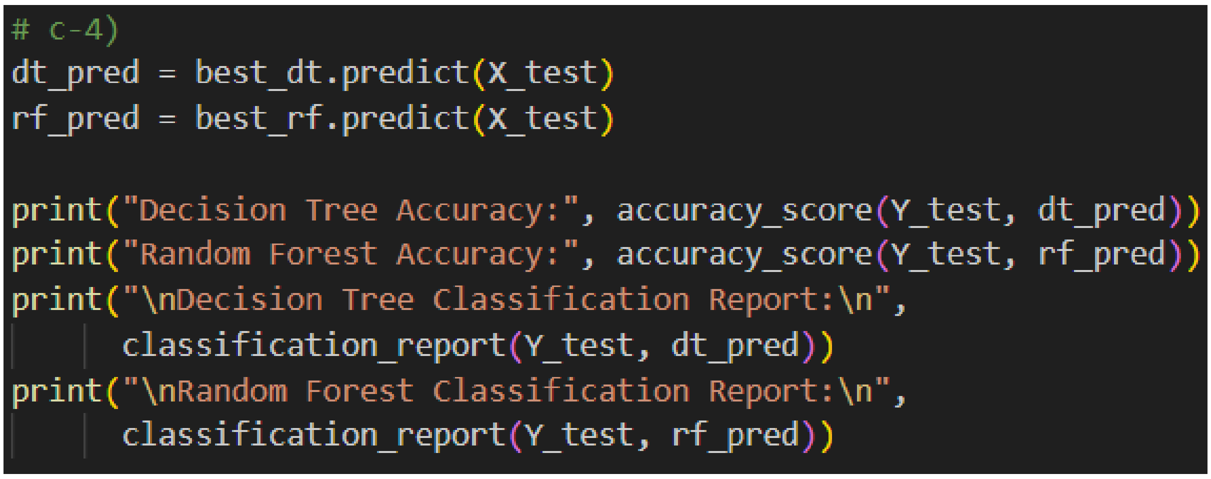
Output:



```
Best parameters found:  {'C': 0.01, 'gamma': 0.1, 'kernel': 'rbf'}
Test accuracy:  0.8246753246753247
```

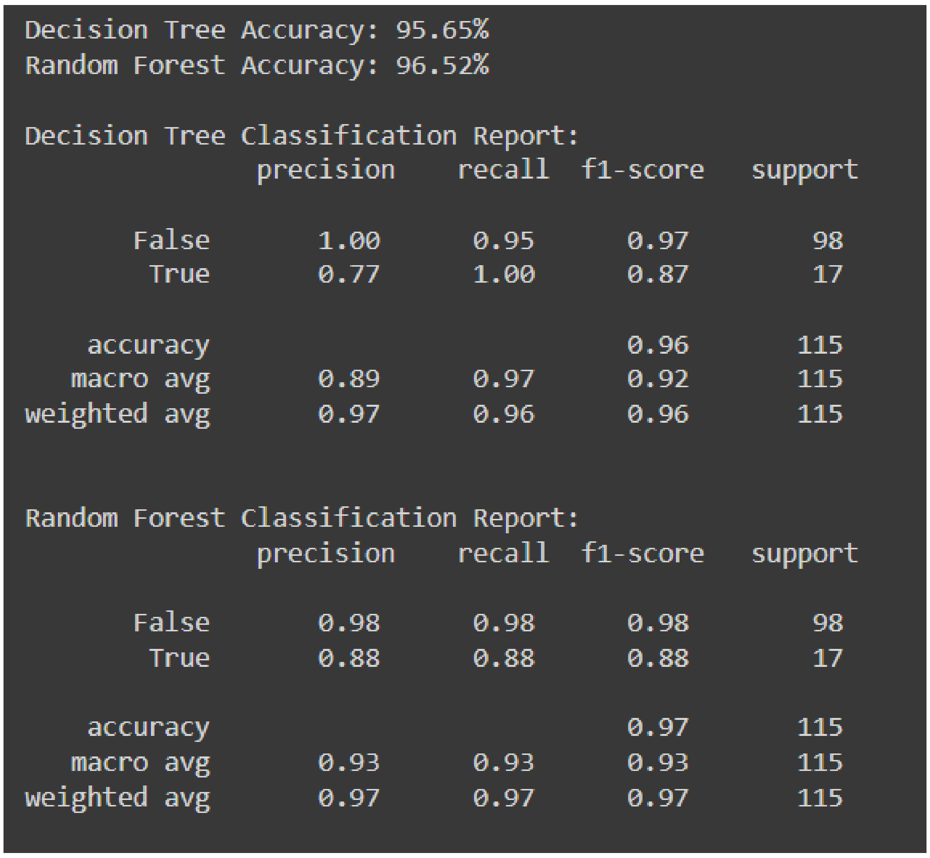
C-4) Decision Tree vs. Random Forest

To compare the best results of our Decision Tree and Random Forest model, we wrote the code below to show their accuracy and classification report.



```python
# c-4)
dt_pred = best_dt.predict(X_test)
rf_pred = best_rf.predict(X_test)

print("Decision Tree Accuracy:", accuracy_score(Y_test, dt_pred))
print("Random Forest Accuracy:", accuracy_score(Y_test, rf_pred))
print("\nDecision Tree Classification Report:\n",
      classification_report(Y_test, dt_pred))
print("\nRandom Forest Classification Report:\n",
      classification_report(Y_test, rf_pred))
```

Output:



```
Decision Tree Accuracy: 95.65%
Random Forest Accuracy: 96.52%

Decision Tree Classification Report:
              precision    recall  f1-score   support

       False       1.00      0.95      0.97        98
        True       0.77      1.00      0.87        17

    accuracy                           0.96       115
   macro avg       0.89      0.97      0.92       115
weighted avg       0.97      0.96      0.96       115


Random Forest Classification Report:
              precision    recall  f1-score   support

       False       0.98      0.98      0.98        98
        True       0.88      0.88      0.88        17

    accuracy                           0.97       115
   macro avg       0.93      0.93      0.93       115
weighted avg       0.97      0.97      0.97       115
```

Bias and variance are two fundamental concepts in statistics and machine learning that describe different types of error in predictive models. Understanding the trade-off between bias and variance is crucial to developing models that generalize well from training data to unseen data.

Bias refers to the error introduced by approximating a real-world problem, which may be quite complicated, by a much simpler model. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting). The model is not complex enough to capture the underlying pattern of the data and hence has a high error on training and test data. On the other hand, Low bias means the model is flexible enough to capture the main trends in the data and hence has a low error on training data.

Variance refers to the amount by which the model's predictions would change if it were trained on a different dataset from the same overall population. Variance captures how much the model is influenced by the randomness in the training data. High variance means the model's predictions vary widely based on the specific observations in the training data, and it captures a lot of noise from the data (overfitting). Such a model performs well on training data but poorly on unseen test data. However, Low variance implies the model makes consistent predictions across different datasets, but it might not capture all the nuances of the data if the model is too simple.

The relationship between bias and variance is commonly described by the bias-variance trade-off. This trade-off is a fundamental problem that faces any supervised learning algorithm. The goal is to find a balance where both bias and variance are minimized to achieve the most generalizable model. Ideally, one wants a model with both low bias (accurately models the real data) and low variance (performs well on any input data from the population). However, these two properties are often at odds. Improving bias usually increases variance and vice versa.

In the context of bias and variance, Decision Trees typically exhibit high variance and low bias, while Random Forests, which are ensembles of decision trees, generally show lower variance and slightly higher bias. Here’s how this plays out:

Decision Tree:

High Variance: A single decision tree is highly sensitive to the data on which it is trained, and small changes to the data can lead to different tree structures. This is why decision trees are prone to overfitting, especially if they are allowed to grow deeply or complexly.

Low Bias: Decision trees make very few assumptions about the structure of the data; hence they are extremely flexible. They can capture complex patterns which can lead to a very accurate performance on the training data, but possibly poor generalization to unseen data.

Random Forest:

Lower Variance: Random forests create several decision trees on randomly selected data samples, then aggregate their predictions to make a final decision.

Slightly Higher Bias: In introducing randomness, random forests sacrifice a bit of the low-bias nature of individual decision trees. The combined effect of multiple trees in the forest leads to the averaging of results, which can smooth out the predictions, leading to slightly increased bias.

For the Decision Tree model, high recall and precision for the 'True' class suggest the tree model captured most of the positive cases perfectly, possibly due to overfitting to the training data.

For the Random Forest model, more balanced precision and recall across both classes and a slightly higher overall accuracy indicate better generalization to the data. The random forest model is less likely to overfit compared to a single decision tree.

The reported results align with this understanding: Random Forest shows slightly better performance in terms of overall accuracy and maintains more consistency across different metrics like precision and recall for both classes.

C-5) Overfitting

This section is included in the attached code. However, since this section is omitted, we will not explain it in the report.