Sollution

February 10, 2023

1 Assignment 2

```
[ ]: %load_ext autoreload
     %autoreload 2
```

```
[ ]: import seaborn as sns
     import matplotlib.pyplot as plt
```

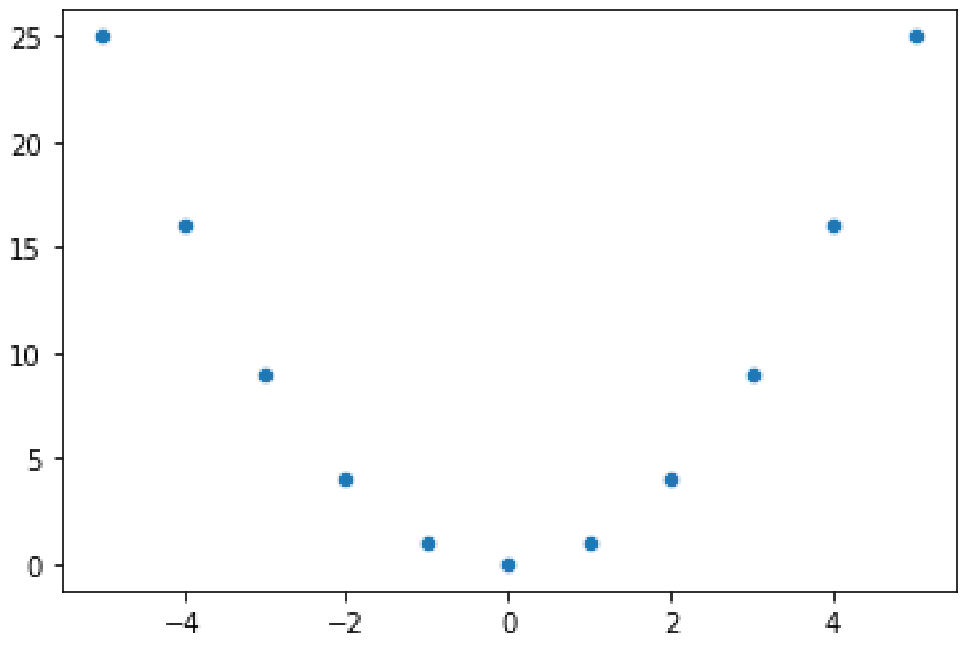
1.1 Question 2

```
[ ]: # Dataset
     x = [-4, -2, 1, 3, -1, -5, 4, 2, 0, -3, 5]
     y = [16, 4, 1, 9, 1, 25, 16, 4, 0, 9, 25]
```

1.1.1 (2 - A) Examine the scatter plot of Y versus X. Is there a relationship between Y and X ?

```
[ ]: sns.scatterplot(x=x, y=y)
```

```
[ ]: <AxesSubplot:>
```

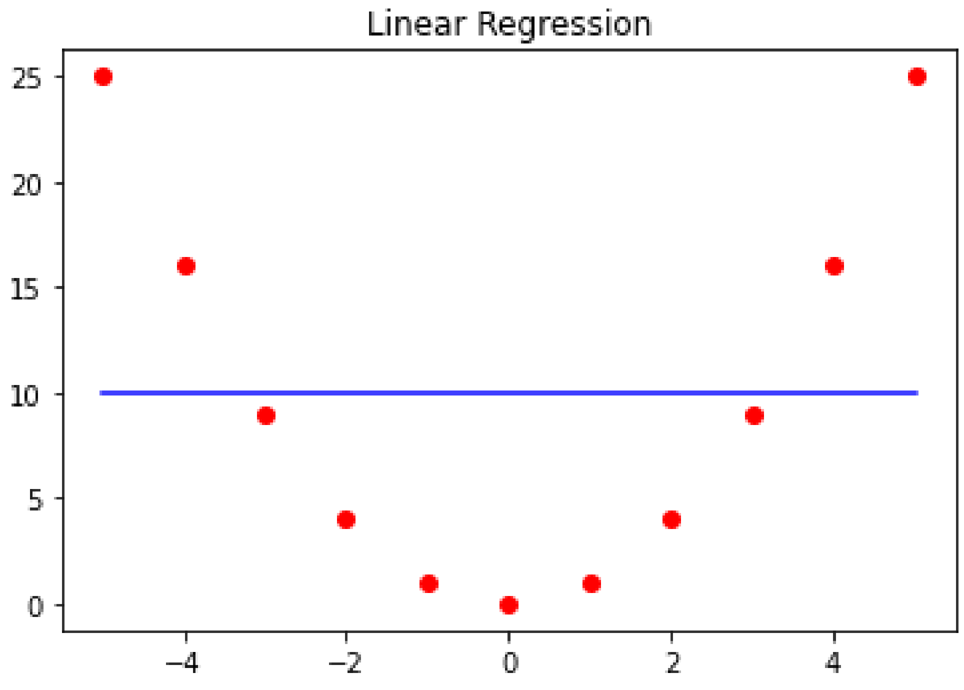


The relation between X and Y is quadratic relation. The quadratic relation between two variables is given as $y = ax^2 + bx + c$ while in this case our data is of the form $y = x^2$ which makes it the simplest form of quadratic relation where $a = 1$, $b = 0$ and $c = 0$

1.1.2 (2 - B) What is the estimated linear regression equation relating Y to X ? What type of regression model it is?

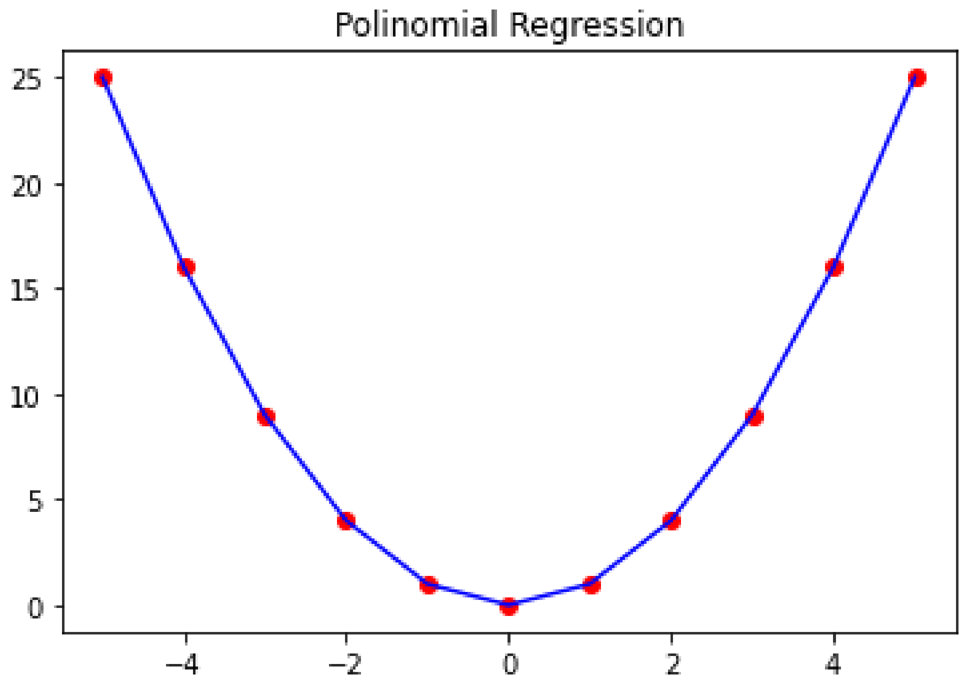
```
[ ]: # (a-2) What is the estimated linear regression equation relating Y to X ? What␣
     ↪type of regression model it is?

     # Estimating Vanila Linear Regression Model
     import numpy as np
     from sklearn.linear_model import LinearRegression

     # Function to Preprocess List Data into Array
     def preprocess_list(x, y):
         if isinstance(x, list) or isinstance(y, list):
             x = np.asarray(x)
             y = np.asarray(y)
         if len(x.shape) == 1:
             x = x.reshape(-1, 1)
         if len(y.shape) == 1:
             y = y.reshape(-1, 1)
         return x, y

     # Function to Estimate Simple Linear Regression Model
```

```python
def estimate_linear_regression(x, y):
    x, y = preprocess_list(x, y)
    linear_reg = LinearRegression(fit_intercept=True)
    linear_reg.fit(x, y)
    y_hat = linear_reg.predict(x)
    return linear_reg.coef_[0][0], linear_reg.intercept_[0], y_hat

# Visualizing the Regression results
def visualize_regression(x, y, y_hat, title="Regression"):
    order = np.argsort(x)
    x = np.array(x)[order]
    y = np.array(y)[order]
    y_hat = np.array(y_hat)[order]

    plt.scatter(x, y, color='red')
    plt.plot(x, y_hat, color='blue')
    plt.title(title)
    plt.show()

coefficients, intercept, y_hat = estimate_linear_regression(x, y)
visualize_regression(x, y, y_hat, title="Linear Regression")
print(f"Estimated Coefficients is {coefficients} and Intercept is {intercept}")
```



```
Estimated Coefficients is 9.539787621017582e-17 and Intercept is 10.0
```

```
[ ]: # Estimate Polynomial Regression
from sklearn.preprocessing import PolynomialFeatures

def estimate_polinomial_regression(x, y, order=2):
    x, y = preprocess_list(x, y)
    poly = PolynomialFeatures(degree=order, include_bias=False)
    poly_features = poly.fit_transform(x)
    polinomial_reg = LinearRegression(fit_intercept=True)
    polinomial_reg.fit(poly_features, y)
    y_hat = polinomial_reg.predict(poly_features)
    return polinomial_reg.coef_[0], polinomial_reg.intercept_[0], y_hat

coefficients, intercept, y_hat = estimate_polinomial_regression(x, y, order=2)
visualize_regression(x, y, y_hat, title="Polinomial Regression")

print(f"Estimated Coefficients is {coefficients[0]} and {coefficients[1]} where␣
 ↪as  Intercept is {intercept}")
```



```
Estimated Coefficients is 9.539787621017581e-17 and 1.0 where as  Intercept is
0.0
```

As we can see that the Simple Linear Regression is not able to properly fit the data while the Polinomial Linear Regression is able to fit the data.

From this run we get the Polinomial Regression model as

$$\hat{y} = 1.0 \times x^2 + 9.539787621017581e - 17 \times x + 0.0$$

Here the coefficient for x is very small (i.e. 9.539787621017581e-17) and intercet is 0.0. We can ignore them and rewrite the model as

$$\hat{y} \sim 1.0 \times x^2$$

This proves our initial answer for the first question.

1.1.3 (2 - C) Test the hypothesis that the slope equals 0. Can we say there is no relationship between two data?

This part of analysis is for Simple Linear Regression Model $y = mX + c$

1. Define the hypothesis
   - Null Hypothesis: $m = 0 \implies$ There is no significant linear relationship between the independent variable X and the dependent variable Y.
   - Alternative Hypothesis: $m \neq 0 \implies$ There is a significant linear relationship between the independent variable X and the dependent variable Y.
2. Decide Significance Level for the test
   - For our case we are choosing Significance Level $\alpha = 0.05$
3. Select Statistical Test
   - We are using **linear regression t-test** to determine whether the slope of the regression line differs significantly from zero.
   - We found the detail of the test on Wikipedia under Slope of a Regression Line section.
4. Compute Test Statistics
   - To Perform **linear regression t-test** we need to compute following set of parameters using our Sample Dataset.
     - Standard error of the slope
     - The Slope of the Regression Line
     - The degrees of freedom –> For Simple Linear Regression Degree of Freedom is $n - 2$ where n is number of data points.
     - The test statistic
     - The p-value associated with the test statistic.
   - To calculate these parameters we will use SciPy Library

[ ]:
```python
from scipy import stats

slope, intercept, r, p, std_err = stats.linregress(x, y)

print(f"The Standard Error of the Slope is {std_err}")
print(f"The Slope of th Regression Line is {slope}")
print(f"The p-value associated with the test statistic is {p}")
```

```
The Standard Error of the Slope is 0.9309493362512627
The Slope of th Regression Line is 0.0
The p-value associated with the test statistic is 1.0
```

[ ]:
```
alpha = 0.05
if p < alpha:
    print(f"As p-value {p} is less than significance level alpha {alpha}, We␣
↪can reject the null hypothesis and accept the alternative hypothesis.")
else:
    print(f"As p-value {p} is greter than or equal to significance level alpha␣
↪{alpha}, We fail to reject the null hypothesis.")
```

```
As p-value 1.0 is greter than or equal to significance level alpha 0.05, We fail
to reject the null hypothesis.
```

As during our test we found p-value to be higher that significance level, we fail to reject null hypothesis.

This leds us to belive that there is **not enough statistical evidence** to conclude that there is some significant linear relationship between the independent variable X and the dependent variable Y.

1.2 Question 3

[ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from utils import styled_print, download_data, read_and_clean_data, \
     plot_box_plot_hist_plot, plot_count_plot, discrete_to_target_plot, \
     continuous_to_target_plot, correlation_analysis,␣
↪traditional_feature_importance
```

[ ]:
```
cleveland_url = "http://archive.ics.uci.edu/ml/machine-learning-databases/
↪heart-disease/processed.cleveland.data"
```

[ ]:
```
headers = {
    0: "age",
    1: "sex",
    2: "cp",
    3: "trestbps",
    4: "chol",
    5: "fbs",
    6: "restecg",
    7: "thalach",
    8: "exang",
    9: "oldpeak",
    10: "slope",
    11: "ca",
    12: "thal",
    13: "target"
}
```

```
[ ]: styled_print(f"Heart Disease Data Analysis", header=True)
     styled_print(f"Extracting Data From {cleveland_url}")
     cleveland_file = download_data(cleveland_url, path_to_download="./data")
     cleveland_df = read_and_clean_data(cleveland_file, header=headers.values())
```

```
> Heart Disease Data Analysis
    Extracting Data From http://archive.ics.uci.edu/ml/machine-learning-
databases/heart-disease/processed.cleveland.data
```

```
[ ]: styled_print(f"Cleveland Dataframe Info", header=True)
     cleveland_df.info()
```

```
> Cleveland Dataframe Info
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       303 non-null    float64
 1   sex       303 non-null    float64
 2   cp        303 non-null    float64
 3   trestbps  303 non-null    float64
 4   chol      303 non-null    float64
 5   fbs       303 non-null    float64
 6   restecg   303 non-null    float64
 7   thalach   303 non-null    float64
 8   exang     303 non-null    float64
 9   oldpeak   303 non-null    float64
 10  slope     303 non-null    float64
 11  ca        299 non-null    float64
 12  thal      301 non-null    float64
 13  target    303 non-null    int64
dtypes: float64(13), int64(1)
memory usage: 33.3 KB
```

2 Dataset Understanding and Observations

Here are some observations from the `heart-disease.names` file regarding the features.

1. `age` is a `continuous` feature which indicates the age of the person in years.
2. `sex` is a `binary categorical` feature indicating sex information.
   - 1 : male
   - 0 : female
3. `cp` is a `categorical` feature which indicates the type of chest pain.
   - Value 1: typical angina
   - Value 2: atypical angina
   - Value 3: non-anginal pain
   - Value 4: asymptomatic

4. `trestbps` is a `continuous` feature indicating resting blood pressure (in mm Hg on admission to the hospital).
5. `chol` is a `continuous` feature indicating serum cholestoral in mg/dl.
6. `fbs` is a `binary categorical` feature indicating fasting blood sugar > 120 mg/dl.
    - 1 : true
    - 0 : false
7. `restecg` is a `categorical` feature indicating resting electrocardiographic results.
    - Value 0: normal
    - Value 1: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV)
    - Value 2: showing probable or definite left ventricular hypertrophy by Estes' criteria
8. `thalach` is a `continuous` feature indicating maximum heart rate achieved.
9. `exang` is a `binary categorical` feature indicating exercise induced angina.
    - 1 : yes
    - 0 : no
10. `oldpeak` is a `continuos` feature indicating ST depression induced by exercise relative to rest.
11. `slope` is a `categorical`feature indicating the slope of the peak exercise ST segment.
    - Value 1: upsloping
    - Value 2: flat
    - Value 3: downsloping
12. `ca` is a `categorical` feature indicating number of major vessels (0-3) colored by flourosopy.
13. `thal` is a `categorical` feature.
    - 3 : normal
    - 6 : fixed defect
    - 7 : reversable defect
14. `target` is a `categorical` feature (target) indicating the diagnosis of heart disease (angiographic disease status)

Two main observations: 1. As all of over categorical features are already numerically encoded we will treat them as discrete feature and not traditional categorical features. 2. As provided in `heart-disease.names` file:

~~~
```The "goal" field refers to the presence of heart disease in the patient.  It is integer valu

So Initially We can convert the `target` into two categories
- 0: Absence of Heart disease
- 1: Presence of Heart disease (Combine current categories 1, 2, 3, and 4)
~~~

[ ]:
```python
categorical_columns = ["cp", "restecg", "slope", "thal", "ca"]
binary_columns = ["sex", "fbs", "exang"]

continuous_columns = ["age", "trestbps", "chol", "thalach", "oldpeak"]
discrete_columns = categorical_columns + binary_columns
target_column = ["target"]
```

[ ]:
```python
# Creating Copy of Dataframe for Data Processing
data_df = cleveland_df.copy()
```

2.1 Data Preprocessing and Exploratory Data Analysis

2.1.1 Preprocessing Target

[ ]:
```
# Check unique values for target and its percentage
data_df["target"].value_counts(dropna=False)
```

[ ]:
```
0    164
1     55
2     36
3     35
4     13
Name: target, dtype: int64
```

[ ]:
```
# Mapping target 2, 3, and 4 to 1.
target_mapping = {2: 1, 3: 1, 4: 1}
data_df["target"] = data_df["target"].apply(lambda x: 1 if x == 2 or x == 3 or␣
 ↪x == 4 else x)
```

[ ]:
```
# Check unique values for target and its percentage
data_df["target"].value_counts(dropna=False)
```

[ ]:
```
0    164
1    139
Name: target, dtype: int64
```

2.1.2 Splitting The Data

To split the data we are using `train_test_split()` method from `sklearn's model_selection` module. The splitting is based on the following parameters: 1. `test_size` is set to `0.2`. It will makes sure that we have 20% of our data for testing and rest 80% of data we can use for training and/or cross-validation. 2. `random_state` is set to `10`. We can set it to any fix number as it will help us in reproducibility of our experiment. 3. `stratify` is set to `target` feature. This will ensure the stratified sampling process. In simple words it will make sure that the distribution of Heart Disease and Non-Heart Disease patient remains as it is even after the split. Refer this for further details. 4. `shuffle` is set to `True`.

[ ]:
```
train_df, test_df = train_test_split(data_df, test_size=.2, random_state=10,␣
 ↪stratify=data_df["target"], shuffle=True)
```

Let's check how stratify sampling make sure that the distribution of data is balance after the split too.

[ ]:
```
# Check unique values for target and its percentage
data_df["target"].value_counts(normalize=True)*100
```

[ ]:
```
0    54.125413
1    45.874587
Name: target, dtype: float64
```

```
[ ]: # Check unique values for target and its percentage
     train_df["target"].value_counts(normalize=True)*100
```

```
[ ]: 0    54.132231
     1    45.867769
     Name: target, dtype: float64
```

```
[ ]: # Check unique values for target and its percentage
     test_df["target"].value_counts(normalize=True)*100
```

```
[ ]: 0    54.098361
     1    45.901639
     Name: target, dtype: float64
```

As we can see that in both training and testing dataset, `54%` of data comes from the `label 0` i.e. Absence of Heart Disease while `45%` of data comes from the `label 1` i.e. Presence of Heart Disease. **These percentages matches the percentage distribution in original dataset.**

```
[ ]: styled_print(f"There are {train_df.shape[0]} data points for training and␣
      ↪{test_df.shape[0]} data points for testing.", header=True)
```

> `There are 242 data points for training and 61 data points fortesting.`

Why are we splitting data first before any exploratory data analysis or even treating missing values??

Our reasoning to split the data at the very beginning of workflow is to make sure that we can ensure that there is no data leak issues. For example, we usually use median value to replace the missing values in a continuous feature. We want to make sure that the median value which we calculate comes only from the training set and we apply it to test set. This way we can gurantee that even in data preprocessing we are not introducing any direct or indirect data leak issues.

This fact is usually ignored in many books and material but in practice it is heavily been used.

2.1.3 Descriptive Statistics

```
[ ]: train_df[continuous_columns].describe().T
```

```
[ ]:           count        mean        std    min    25%    50%     75%    max
     age       242.0   54.669421   9.102814   29.0   48.0   56.0   61.00   77.0
     trestbps  242.0  131.727273  17.601160   94.0  120.0  130.0  140.00  200.0
     chol      242.0  247.909091  53.201878  126.0  212.0  240.0  276.75  564.0
     thalach   242.0  148.896694  23.489242   71.0  132.0  153.0  165.75  202.0
     oldpeak   242.0    1.008678   1.109880    0.0    0.0    0.7    1.60    5.6
```

Observations - The `average age is 54 years` in our dataset while the `median age is 56`. These two numbers are relatively close and because of that we hope to see almost `normal distribution of age feature`. - The `average trestbps` i.e. resting blood pressure at the time of admission to hospital is `~ 131 mm Hg` while `median is ~ 130 mm Hg`. These two numbers are relatively close and because of that we hope to see almost `normal distribution of trestbps feature`. - The `min chol is 126` while the `max is 564`. The range of values between min and median is 114 (240 - 126) while the range of values between max and median is 324 (564 - 240). The standard deviation for `chol` is also 53 which is relatively higher than other features. This makes us believe that there is some skewness in `chol` feature and it would be really interesting to check the distribution of it.

[ ]:
```python
for col in discrete_columns:
    styled_print("~"*5 + f"Unique Value Counts for {col}" + "~"*5, header=True)
    print(train_df[col].value_counts(normalize=True, dropna=False)*100)
```

```
> ~~~~~Unique Value Counts for cp~~~~~
4.0    46.694215
3.0    28.099174
2.0    17.768595
1.0     7.438017
Name: cp, dtype: float64
> ~~~~~Unique Value Counts for restecg~~~~~
0.0    51.652893
2.0    47.520661
1.0     0.826446
Name: restecg, dtype: float64
> ~~~~~Unique Value Counts for slope~~~~~
2.0    50.000000
1.0    44.628099
3.0     5.371901
Name: slope, dtype: float64
> ~~~~~Unique Value Counts for thal~~~~~
3.0    54.132231
7.0    39.256198
6.0     5.785124
NaN     0.826446
Name: thal, dtype: float64
> ~~~~~Unique Value Counts for ca~~~~~
0.0    56.611570
1.0    22.727273
2.0    13.223140
3.0     6.198347
NaN     1.239669
Name: ca, dtype: float64
> ~~~~~Unique Value Counts for sex~~~~~
1.0    67.768595
0.0    32.231405
Name: sex, dtype: float64
> ~~~~~Unique Value Counts for fbs~~~~~
0.0    83.884298
1.0    16.115702
```

```
Name: fbs, dtype: float64
> ~~~~~Unique Value Counts for exang~~~~~
0.0    68.595041
1.0    31.404959
Name: exang, dtype: float64
```

Observations - ~ 67% of our training data represents `male` while ~ 33% of our training data represents `female`. It would be really interesting to see whether there is any relation between `sex` of an individual and presence of heart disease. - ~ 16% of our patients in our data has fasting blood sugar > 120 mg/dl. It would be interesting to see how this relates to the presence of heart disease. - ~ 1.2% of data in `ca` feature is missing. On the other hand ~ 0.82% of data is missing from `thal` feature. We might need to decide a strategy to replace missing values.

2.1.4 Univariate Analysis - Continuous Features

[ ]:
```python
import os
result_path = "./images"
for col in continuous_columns:
    plot_box_plot_hist_plot(
        df=train_df,
        column=col,
        title=f"Distribution Plot for {col}",
        figsize=(4, 4),
        save_flag=True,
        dpi=100,
        file_path=os.path.join(result_path, f"box-hist-plot-{col}.png")

)
```

2.1.5 Univariate Analysis - Discrete Features

[ ]:
```python
for col in discrete_columns:
    plot_count_plot(
        df=train_df,
        column=col,
        title=f"Distribution Plot for {col}",
        figsize=(4, 4),
        save_flag=True,
        dpi=100,
        file_path=os.path.join(result_path, f"count-plot-{col}.png")
    )
```

2.1.6 Bivariate Analysis - Categorical Features - Target Feature

```
[ ]: for discrete_column in discrete_columns:
         discrete_to_target_plot(
             train_df,
             discrete_column,
             target_column[0],
             title=f"Relative Plot for {discrete_column} and {target_column[0]}",
             figsize=(4, 4),
             save_flag=True,
             dpi=100,
             file_path=os.path.join(result_path,␣
      ↪f"bi-disc-target-plot-{discrete_column}.png")
         )
```

Observations

- **`sex`** : `male` has more than `55%` of chances of having presence of heart disease. This is relatively higher number as compared to `25%` for `female`.
- **`cp`** : Person reporting asymptomatic type of chest pain has almost 75% chances of having presence of heart disease. This is relatively higher number as compared to `40%` chances for person reporting `typical angina`, `15%` chances for person reporting `atypical angina` and `25%` chances for person reporting `non-anginal pain`.
- **`fbs`** : People with `higher` fasting blood sugar and people with `lower` fasting blood sugar has almost similar chances of having presence of heart disease. This is really interesting find as we traditionally believe that blood sugar level has higher impact on heart dieses.
- **`restecg`** : People showing `normal` resting electrocardiographic results has lower chances of having presence of heart disease.
- **`thal`** : People with `fixed defect` or `reversible defect` has higher chances of having presence of heart disease as compared to people with `normal` value for `thal`.

2.1.7 Bivariate Analysis - Numerical Features - Target Feature

```
[ ]: for continuous_column in continuous_columns:
         continuous_to_target_plot(
             train_df,
             continuous_column,
             target_column[0],
             title=f"Relative Plot for {continuous_column} and {target_column[0]}",
             figsize=(4, 4),
             save_flag=True,
             dpi=100,
             file_path=os.path.join(result_path,␣
      ↪f"bi-conti-target-plot-{continuous_column}.png")
         )
```

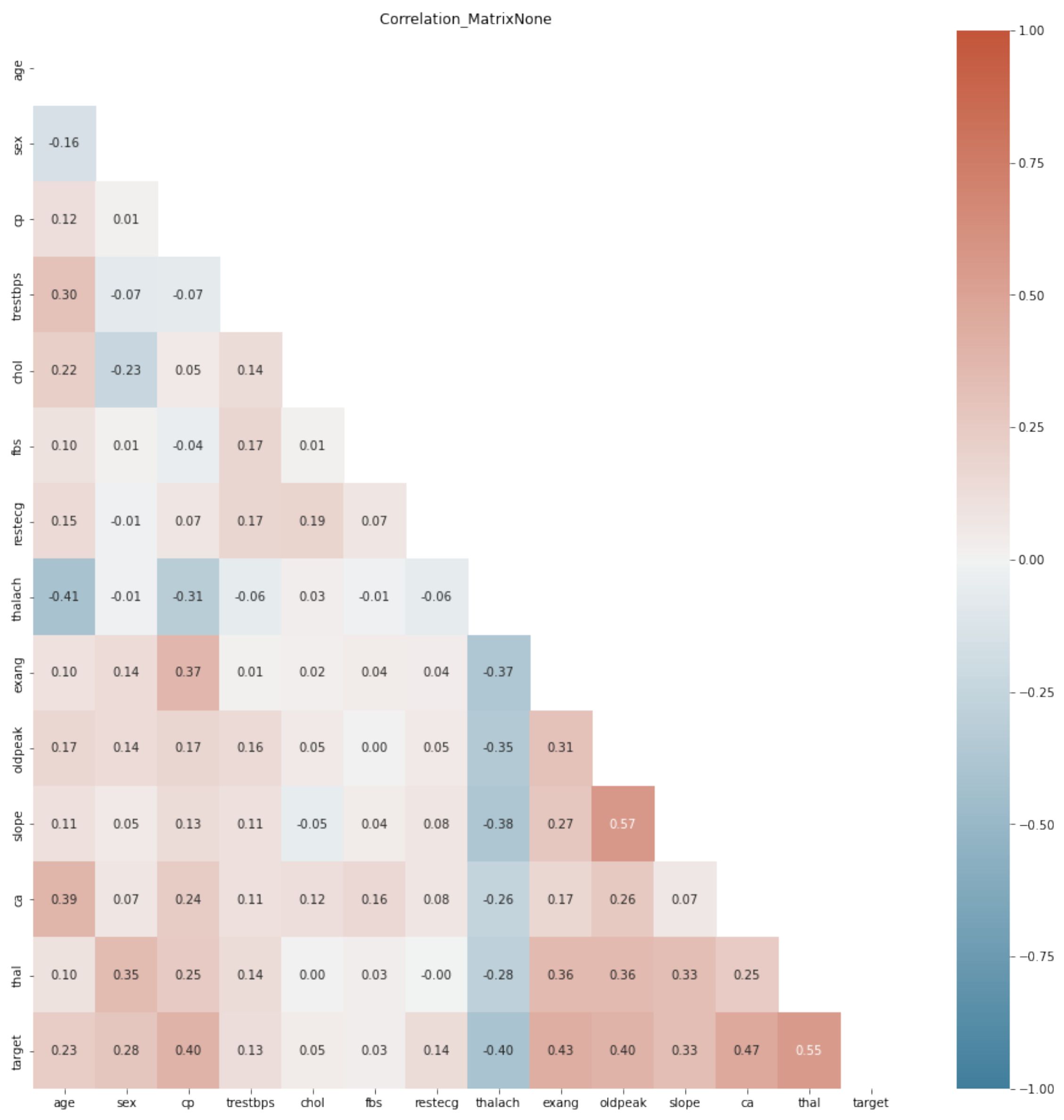
Observations

- **`age`** : People reporting `presence of heart dieses` has `higher median age` as compared to people reporting absence of heart dieses (~ 60 vs 52 years).

- **age** : There some individuals who are relatively young (age < 40) but still reports the presence of heart dieses. Even though it is a small number of sample it would be interesting to see those outliers and analyze our model around it.
- **oldpeak** : People reporting `presence of heart dieses` has `higher median ST depression induced by exercise relative to rest` as compared to people who are reporting absence of heart dieses.

2.1.8 Correlation Analysis

[ ]:
```
corr = correlation_analysis(
    train_df,
    method='pearson',
    save_flag=False,
    plot_dir="plots",
    title=None,
    prefix="Correlation_Matrix",
    postfix=" ",
    figsize=(16, 16)
)
```



Observations

- There are no features which are heavily correlated with one another apart from `oldpeak` and `slope`.
- `oldpeak` and `slope` has positive correlation of 0.57 which is not that significant that we end up removing one of them. For now we assume that both the features might help the model to learn a good segmentation of our sample dataset. Later we can perform model coefficient analysis and hypothesis testing to remove features if they are not significant.
- `thal` has a higher correlation with our `target` feature. This makes `thal` one of the important features to keep in our model.

2.1.9 Missing Value Treatment

```
[ ]: train_df.isnull().sum()
```

```
[ ]: age         0
     sex         0
     cp          0
     trestbps    0
     chol        0
     fbs         0
     restecg     0
     thalach     0
     exang       0
     oldpeak     0
     slope       0
     ca          3
     thal        2
     target      0
     dtype: int64
```

```
[ ]: test_df.isnull().sum()
```

```
[ ]: age         0
     sex         0
     cp          0
     trestbps    0
     chol        0
     fbs         0
     restecg     0
     thalach     0
     exang       0
     oldpeak     0
     slope       0
     ca          1
     thal        0
     target      0
     dtype: int64
```

As we have a very small number of missing values in the training and test dataset, it would be better to drop those rows instead of trying to figure out strategy to replace them.

```
[ ]: train_df = train_df.dropna()
     test_df = test_df.dropna()
```

Let's verify that all the rows with missing values are dropped.

```
[ ]: train_df.isnull().sum()
[ ]: age         0
     sex         0
     cp          0
     trestbps    0
     chol        0
     fbs         0
     restecg     0
     thalach     0
     exang       0
     oldpeak     0
     slope       0
     ca          0
     thal        0
     target      0
     dtype: int64
```

```
[ ]: test_df.isnull().sum()
```

```
[ ]: age         0
     sex         0
     cp          0
     trestbps    0
     chol        0
     fbs         0
     restecg     0
     thalach     0
     exang       0
     oldpeak     0
     slope       0
     ca          0
     thal        0
     target      0
     dtype: int64
```

```
[ ]: styled_print(f"There are {train_df.shape[0]} data points for training and␣
      ↪{test_df.shape[0]} data points for testing.")
```

```
There are 237 data points for training and 60 data points for testing.
```

2.2 Model Creation

We will follow these steps to create the `BASELINE` model: - Prepare the data for modeling. - Create X and Y - (x_train, y_train) and (x_test, y_test) - Scale Continuous Features using Min-Max Scaler. - Scale Discrete Features using Min-Max Scaler. - Build the `BASELINE` model on the train data. - Create Linear Regression Model - Test the model on the test set. - Calculate R^2 score to measure the performance of the model.

Here `BASELINE` model means - We will use all features to create the model.

Later we will improve the model based on the learnings from the BASELINE model.

2.2.1 Prepare the data for modeling

```
[ ]: from sklearn.preprocessing import MinMaxScaler
```

```
[ ]: y_train = train_df[target_column[0]].copy()
     x_train = train_df.drop(target_column[0], axis=1)
```

```
[ ]: y_test = test_df[target_column[0]].copy()
     x_test = test_df.drop(target_column[0], axis=1)
```

```
[ ]: scaler = MinMaxScaler()
     scaler.fit(x_train)
     x_train = pd.DataFrame(scaler.transform(x_train),columns = x_train.columns)
     x_test = pd.DataFrame(scaler.transform(x_test),columns = x_test.columns)
```

```
[ ]: x_train.head(10)
```

```
[ ]:         age  sex        cp  trestbps      chol  fbs  restecg   thalach  exang  \
     0  0.708333  0.0  1.000000  0.528302  0.641553  0.0      1.0  0.633588    0.0
     1  0.395833  1.0  1.000000  0.339623  0.296804  1.0      1.0  0.603053    1.0
     2  0.437500  1.0  1.000000  0.528302  0.267123  0.0      1.0  0.435115    0.0
     3  0.708333  0.0  0.333333  0.433962  0.157534  0.0      0.0  0.824427    0.0
     4  0.416667  0.0  0.333333  0.377358  0.331050  0.0      0.0  0.694656    0.0
     5  0.583333  1.0  1.000000  0.547170  0.337900  0.0      0.0  0.129771    1.0
     6  0.645833  0.0  1.000000  0.603774  0.408676  0.0      1.0  0.687023    0.0
     7  0.750000  0.0  0.666667  0.433962  0.664384  1.0      1.0  0.656489    0.0
     8  0.291667  0.0  0.666667  0.264151  0.198630  0.0      0.0  0.717557    0.0
     9  0.666667  1.0  1.000000  0.415094  0.091324  0.0      1.0  0.412214    1.0

         oldpeak  slope        ca  thal
     0  0.714286    0.5  1.000000   1.0
     1  0.000000    0.0  0.666667   1.0
     2  0.464286    0.5  0.000000   1.0
     3  0.000000    0.0  0.666667   0.0
     4  0.000000    0.5  0.000000   0.0
     5  0.214286    0.5  0.333333   1.0
     6  0.000000    0.0  0.000000   0.0
     7  0.142857    0.0  0.333333   0.0
     8  0.035714    0.5  0.000000   0.0
     9  0.642857    0.5  0.333333   0.0
```

```
[ ]: x_test.head(10)
```

```
[ ]:         age  sex        cp  trestbps      chol  fbs  restecg   thalach  exang  \
     0  0.250000  0.0  0.666667  0.169811  0.324201  0.0      1.0  0.770992    1.0
```

```
1  0.729167  1.0  1.000000  0.245283  0.273973  0.0      1.0  0.190840    1.0
2  0.291667  1.0  1.000000  0.245283  0.116438  0.0      1.0  0.374046    1.0
3  0.125000  1.0  1.000000  0.301887  0.356164  0.0      1.0  0.648855    1.0
4  0.562500  1.0  0.333333  0.339623  0.216895  0.0      1.0  0.702290    0.0
5  0.520833  1.0  0.666667  0.292453  0.335616  0.0      1.0  0.618321    0.0
6  0.604167  1.0  1.000000  0.188679  0.438356  0.0      0.5  0.526718    0.0
7  0.708333  0.0  0.666667  0.386792  0.287671  0.0      1.0  0.770992    0.0
8  0.250000  0.0  0.333333  0.103774  0.164384  0.0      0.0  0.740458    0.0
9  0.354167  1.0  1.000000  0.245283  0.280822  0.0      1.0  0.557252    0.0

    oldpeak  slope        ca  thal
0  0.000000    0.0  0.000000  0.00
1  0.392857    1.0  0.333333  0.00
2  0.446429    0.5  0.000000  1.00
3  0.000000    0.0  0.000000  1.00
4  0.000000    0.0  0.000000  1.00
5  0.089286    1.0  0.333333  0.00
6  0.785714    1.0  1.000000  0.75
7  0.000000    0.0  0.000000  0.00
8  0.000000    0.0  0.333333  0.00
9  0.142857    0.0  0.000000  1.00
```

2.2.2 Build the `BASELINE` model on the train data.

[ ]:
```python
from sklearn.linear_model import LinearRegression
```

[ ]:
```python
linear_regression = LinearRegression(fit_intercept=True, n_jobs=-1)
linear_regression.fit(x_train, y_train)
```

[ ]:
```
LinearRegression(n_jobs=-1)
```

[ ]:
```python
train_r2_score = linear_regression.score(x_train, y_train, sample_weight=None)
test_r2_score = linear_regression.score(x_test, y_test, sample_weight=None)
```

[ ]:
```python
styled_print("Performance of Baseline Linear Regression Model", header=True)
styled_print(f"The train R2 Score for Linear Regression is {train_r2_score}")
styled_print(f"The test R2 Score for Linear Regression is {test_r2_score}")
```

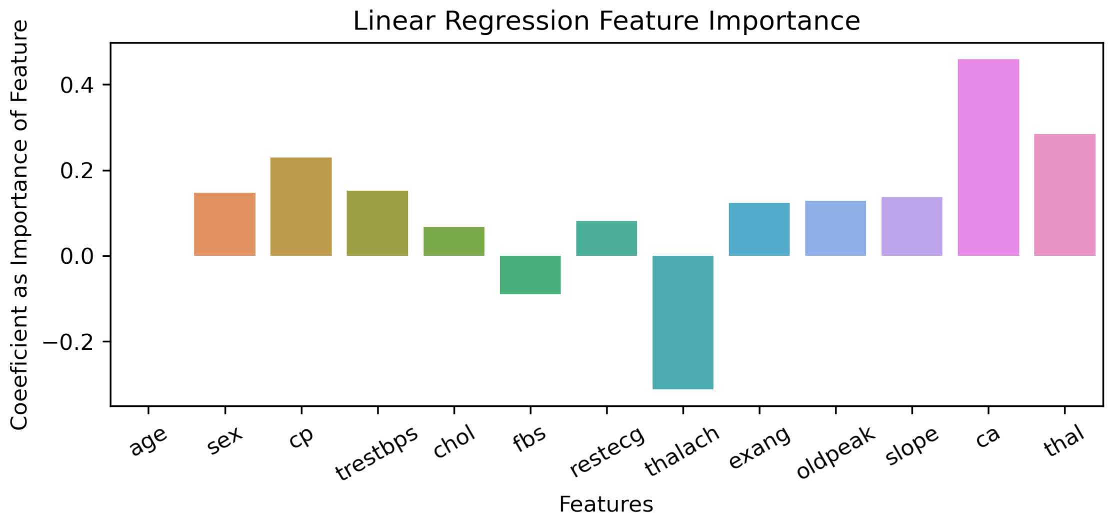
```
› Performance of Baseline Linear Regression Model
    The train R2 Score for Linear Regression is 0.5453825504687724
    The test R2 Score for Linear Regression is 0.4735400197312123
```

Observations

As indicated here, our training and test R^2 scores are very low. The main reason behind this is vanila linear regression models usually have poor performance for `Discreate` target variable.

For `Discrete` target variable, it is recommended to use Logistic Regression.

```
[ ]: feature_importance = traditional_feature_importance(linear_regression, x_train,␣
     ↪figsize=(8, 3), title="Linear Regression Feature Importance")
```



2.2.3 Improve BASELINE model with Ridge Regression

```
[ ]: from sklearn.linear_model import Ridge
     from sklearn.model_selection import GridSearchCV
```

```
[ ]: alpha_space = list(np.logspace(-4, 0, 30))    # Checking for alpha from .0001 to␣
     ↪1 and finding the best value for alpha
     parameters = {'alpha': alpha_space + [5, 10, 15]}
```

```
[ ]: # define the model/ estimator
     ridge_regressor = Ridge()
     # define the grid search
     Ridge_reg= GridSearchCV(ridge_regressor, parameters, scoring='r2', cv=5,␣
     ↪n_jobs=-1)
     #fit the grid search
     Ridge_reg.fit(x_train, y_train)
```

```
[ ]: GridSearchCV(cv=5, estimator=Ridge(), n_jobs=-1,
                  param_grid={'alpha': [0.0001, 0.00013738237958832623,
                                        0.00018873918221350977,
                                        0.0002592943797404667, 0.0003562247890262444,
                                        0.0004893900918477494, 0.0006723357536499335,
                                        0.0009236708571873865, 0.0012689610031679222,
                                        0.0017433288221999873, 0.0023950266199874 86,
                                        0.0032903445623126675, 0.004520353656360241,
                                        0.006210169418915616, 0.008531678524172805,
                                        0.011721022975334805, 0.01610262027560939,
                                        0.02212216291070448, 0.03039195382313198,
                                        0.041753189365604, 0.05736152510448681,
                                        0.07880462815669913, 0.1082636733874054,
                                        0.14873521072935117, 0.20433597178569418,
                                        0.2807216203941176, 0.38566204211634725,
                                        0.5298316906283708, 0.7278953843983146, 1.0,
     …]},
                  scoring='r2')
```

```
[ ]: # best estimator
     print(Ridge_reg.best_estimator_)
```

```
Ridge(alpha=5)
```

```
[ ]: # best model
     ridge_regression_model = Ridge_reg.best_estimator_
     ridge_regression_model.fit(x_train, y_train)
```

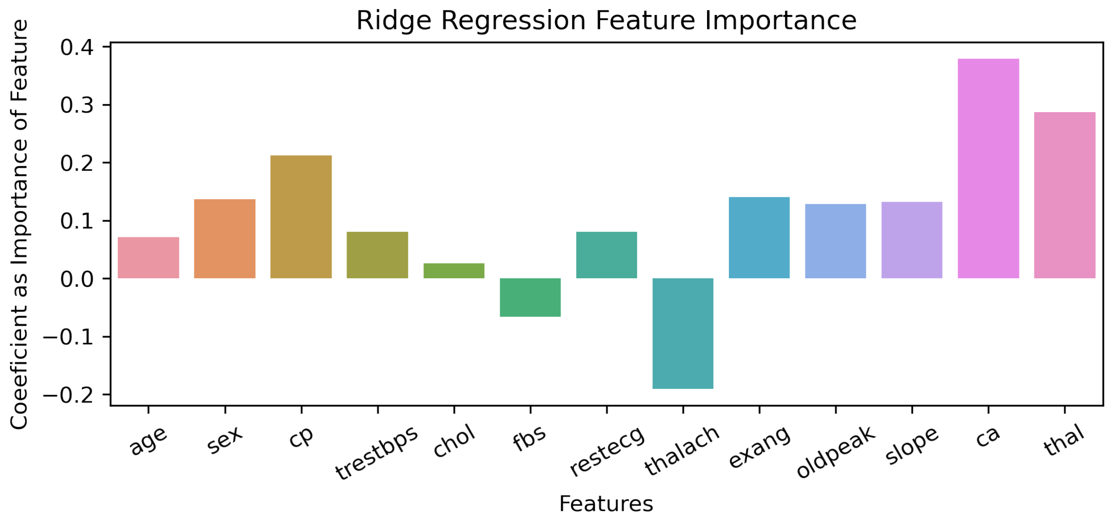
```
[ ]: Ridge(alpha=5)
```

```
[ ]: train_r2_score = ridge_regression_model.score(x_train, y_train,␣
      ↪sample_weight=None)
     test_r2_score = ridge_regression_model.score(x_test, y_test, sample_weight=None)
```

```
[ ]: styled_print("Performance of Baseline Ridge Regression Model", header=True)
     styled_print(f"The train R2 Score for Ridge Regression is {train_r2_score}")
     styled_print(f"The test R2 Score for Ridge Regression is {test_r2_score}")
```

```
> Performance of Baseline Ridge Regression Model
    The train R2 Score for Ridge Regression is 0.5403262607230832
    The test R2 Score for Ridge Regression is 0.4697211130938044
```

```
[ ]: feature_importance = traditional_feature_importance(ridge_regression_model,␣
      ↪x_train, figsize=(8, 3), title="Ridge Regression Feature Importance")
```



2.2.4 Improve BASELINE model with Lasso Regression

```
[ ]: from sklearn.linear_model import Lasso
```

```
[ ]: alpha_space = list(np.logspace(-4, 0, 30))    # Checking for alpha from .0001 to␣
      ↪1 and finding the best value for alpha
     parameters = {'alpha': alpha_space + [5, 10, 15]}
```

```
[ ]: # define the model/ estimator
     lasso_regressor = Lasso()
     # define the grid search
     Lasso_reg= GridSearchCV(lasso_regressor, parameters, scoring='r2', cv=5,␣
      ↪n_jobs=-1)
     #fit the grid search
     Lasso_reg.fit(x_train, y_train)
```

```
[ ]: GridSearchCV(cv=5, estimator=Lasso(), n_jobs=-1,
                  param_grid={'alpha': [0.0001, 0.00013738237958832623,
                                        0.00018873918221350977,
                                        0.0002592943797404667, 0.0003562247890262444,
                                        0.0004893900918477494, 0.0006723357536499335,
                                        0.0009236708571873865, 0.0012689610031679222,
                                        0.0017433288221999873, 0.002395026619987486,
                                        0.0032903445623126675, 0.004520353656360241,
                                        0.006210169418915616, 0.008531678524172805,
                                        0.011721022975334805, 0.01610262027560939,
                                        0.02212216291070448, 0.03039195382313198,
                                        0.041753189365604, 0.05736152510448681,
                                        0.07880462815669913, 0.1082636733874054,
                                        0.14873521072935117, 0.20433597178569418,
                                        0.2807216203941176, 0.38566204211634725,
                                        0.5298316906283708, 0.7278953843983146, 1.0,
…]},
             scoring='r2')
```

[ ]:
```
# best estimator
print(Lasso_reg.best_estimator_)
```

```
Lasso(alpha=0.006210169418915616)
```

[ ]:
```
# best model
lasso_regression_model = Lasso_reg.best_estimator_
lasso_regression_model.fit(x_train, y_train)
```

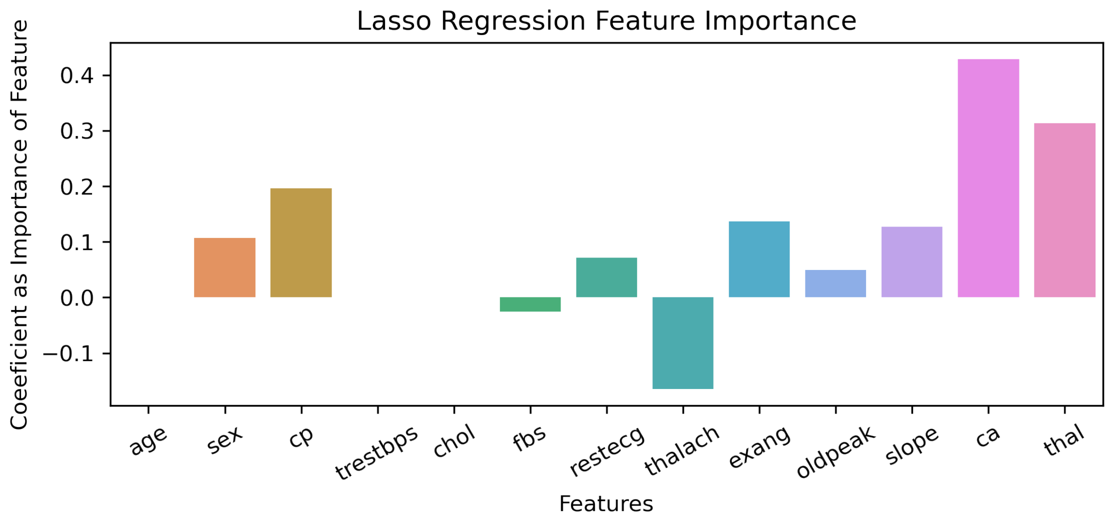
[ ]:
```
Lasso(alpha=0.006210169418915616)
```

[ ]:
```
train_r2_score = lasso_regression_model.score(x_train, y_train,␣
↪sample_weight=None)
test_r2_score = lasso_regression_model.score(x_test, y_test, sample_weight=None)
```

[ ]:
```
styled_print("Performance of Baseline Lasso Regression Model", header=True)
styled_print(f"The train R2 Score for Lasso Regression is {train_r2_score}")
styled_print(f"The test R2 Score for Lasso Regression is {test_r2_score}")
```

```
› Performance of Baseline Lasso Regression Model
    The train R2 Score for Lasso Regression is 0.5333242531889925
    The test R2 Score for Lasso Regression is 0.4470181314375199
```

[ ]:
```
feature_importance = traditional_feature_importance(lasso_regression_model,␣
↪x_train, figsize=(8, 3), title="Lasso Regression Feature Importance")
```



2.2.5 Improve BASELINE model with Removing Less Important Features

- **As Linear and Lasso Regression Showing age as least important feature**
- **Second least important feature is chol.**

We start the analysis by dropping age and chol features.

```
[ ]: x_train_updated = x_train.drop(columns=['age', 'chol'], axis=1)
     x_test_updated = x_test.drop(columns=['age', 'chol'], axis=1)
```

```
[ ]: linear_regression = LinearRegression(fit_intercept=True, n_jobs=-1)
     linear_regression.fit(x_train_updated, y_train)
```

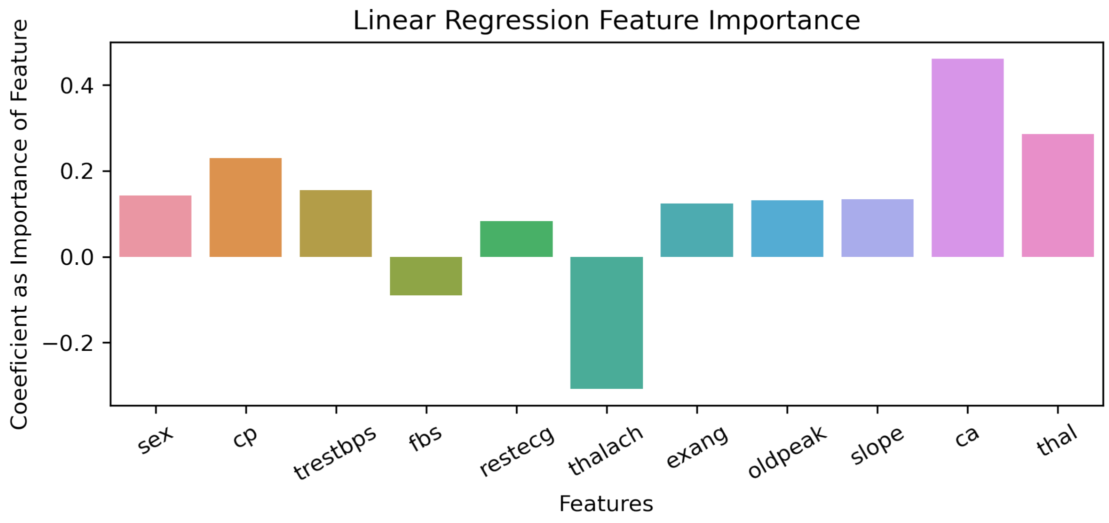
```
[ ]: LinearRegression(n_jobs=-1)
```

```
[ ]: train_r2_score = linear_regression.score(x_train_updated, y_train,␣
      ↪sample_weight=None)
     test_r2_score = linear_regression.score(x_test_updated, y_test,␣
      ↪sample_weight=None)
```

```
[ ]: styled_print("Performance of Baseline Linear Regression Model", header=True)
     styled_print(f"The train R2 Score for Linear Regression is {train_r2_score}")
     styled_print(f"The test R2 Score for Linear Regression is {test_r2_score}")
```

```
> Performance of Baseline Linear Regression Model
    The train R2 Score for Linear Regression is 0.5451528179036751
    The test R2 Score for Linear Regression is 0.4708226296387049
```

```
[ ]: feature_importance = traditional_feature_importance(linear_regression,␣
      ↪x_train_updated, figsize=(8, 3), title="Linear Regression Feature␣
      ↪Importance")
```



As you can see that even after dropping two features there is no improvement in the performance of the model. Our best assumption is that as over `target` variable is `discreate`, Linear, Ridge and Lasso regression wouldn't be a proper choice as algorithm. However Logistic Regression is a good choice for `discrete` target variables. In next step we briefly try Logistic Regression to prove our hypothesis. If permitted in next set of assignments, we would like to further investigate into it.

2.2.6 Improve `BASELINE` model with Logistic Regression

[ ]:
```
from sklearn.linear_model import LogisticRegression
```

[ ]:
```
logistic_regression = LogisticRegression(penalty='l2', multi_class='ovr',␣
 ↪fit_intercept=True, n_jobs=-1)
logistic_regression.fit(x_train, y_train)
```

[ ]:
```
LogisticRegression(multi_class='ovr', n_jobs=-1)
```

[ ]:
```
train_mean_acc = logistic_regression.score(x_train, y_train, sample_weight=None)
test_mean_acc = logistic_regression.score(x_test, y_test, sample_weight=None)
```

[ ]:
```
styled_print("Performance of Baseline Logistic Regression Model", header=True)
styled_print(f"The train Mean Accuracy for Logistic Regression is␣
 ↪{train_mean_acc}")
styled_print(f"The test Mean Accuracy for Logistic Regression is␣
 ↪{test_mean_acc}")
```

```
> Performance of Baseline Logistic Regression Model
    The train Mean Accuracy for Logistic Regression is 0.8438818565400844
    The test Mean Accuracy for Logistic Regression is 0.8333333333333334
```

```
[ ]: from sklearn.metrics import classification_report
     target_names = ['No Heart Disease', 'Heart Disease']
```

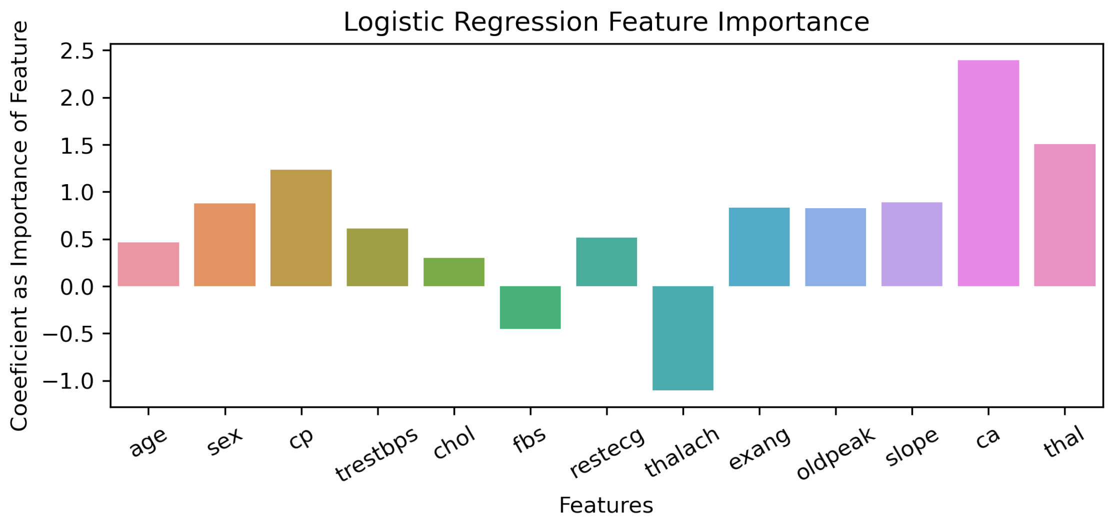
```
[ ]: y_train_pred = logistic_regression.predict(x_train)
     print(classification_report(y_train, y_train_pred, target_names=target_names))
```

```
                  precision    recall  f1-score   support

No Heart Disease       0.84      0.88      0.86       128
   Heart Disease       0.85      0.81      0.83       109

        accuracy                           0.84       237
       macro avg       0.84      0.84      0.84       237
    weighted avg       0.84      0.84      0.84       237
```

```
[ ]: y_test_pred = logistic_regression.predict(x_test)
     print(classification_report(y_test, y_test_pred, target_names=target_names))
```

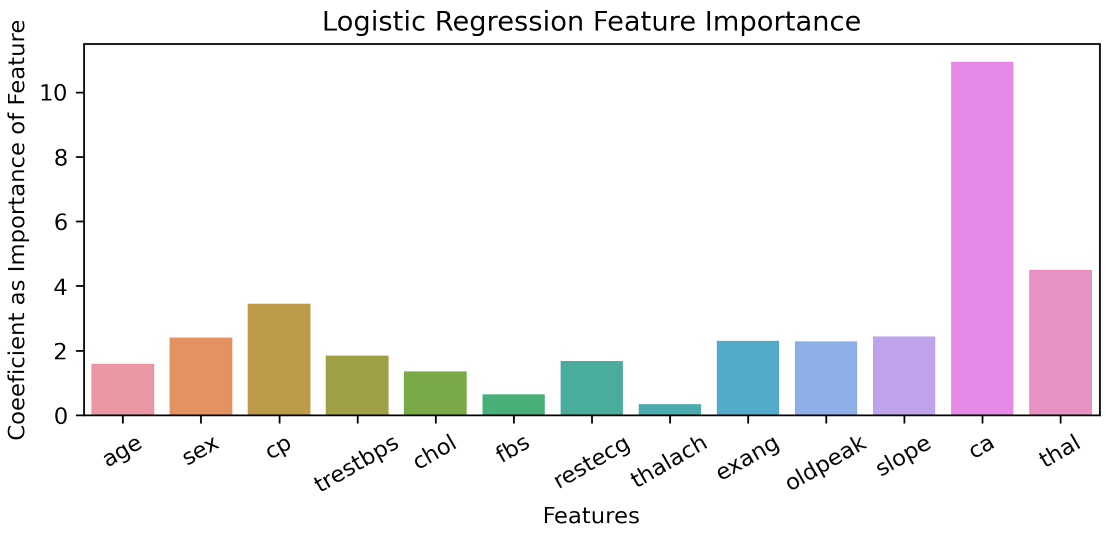
```
                  precision    recall  f1-score   support

No Heart Disease       0.82      0.88      0.85        32
   Heart Disease       0.85      0.79      0.81        28

        accuracy                           0.83        60
       macro avg       0.83      0.83      0.83        60
    weighted avg       0.83      0.83      0.83        60
```

```
[ ]: feature_importance = traditional_feature_importance(logistic_regression,␣
      ↪x_train, figsize=(8, 3), title="Logistic Regression Feature Importance")
```



Logistic Regression outputs the log odds of $Y = 1$. This means that to extract the actual coeffients we need to apply exponetial to the coefficient we got.

```
[ ]: feature_importance = traditional_feature_importance(logistic_regression,␣
      ↪x_train, apply_ln=True, figsize=(8, 3), title="Logistic Regression Feature␣
      ↪Importance")
```



2.3 Conclusion

Vanila Linear Regression are only better when the target variable is continuous in nature. Whenever the target variable is discrete, we should use Logistic Regression model. That is also the fundamental difference between classification and regression type of problems.