Dataset:

Heart Disease Dataset

Abstract:

In this project I am working on “The Heart Disease dataset” that I have collected from Kaggle. The purpose of the project is to implement and compare between several classification algorithms for example Naïve Bayesian and Single Layer Perceptron. The dataset contains both categorical and continuous variables. These algorithms predict whether a patient has heart disease based on various medical and demographic features. I have implemented Naive Bayesian classifier and Single Layer Perceptron algorithms from scratch. In Naïve Bayesian classifier for discreet features the probability formula is applied and for continuous feature probability distribution function is used to calculate likelihood. On the other hand, in Single Layer Perceptron to make a predict the classifier is made by using activation function. After implementing both classifier the accuracy for both classifier is calculated and compared them.

Overall in this project there is detailed data description, Naive Bayesian classifier and Single Layer Perceptron classifier implementation to predict whether patient has heart disease and lastly a comparison of both classifiers by calculating accuracy.

Data Description:

The Heart Disease dataset contains 1025 rows and 14 columns. Each row represents a patient who underwent a medical examination for heart disease. The columns represent various medical and demographic features and class, such as

1. age: age in years
2. sex: sex (1 = male; 0 = female)
3. cp: chest pain type
4. trestbps: resting blood pressure
5. chol: cholesterol
6. fbs: fasting blood sugar
7. restecg: resting electrocardiographic results
8. thalach: maximum heart rate achieved
9. exang: exercise induced angina
10. oldpeak: ST depression induced by exercise relative to rest
11. slope: the slope of the peak exercise ST segment
12. ca: number of major vessels (0-3) colored by fluoroscopy
13. thal: normal; fixed defect; reversible defect.
14. target: diagnosis of heart diseases (0 or 1)

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	52	1	0	125	212	0	1	168	0	1.0	2	2	3	0
1	53	1	0	140	203	1	0	155	1	3.1	0	0	3	0
2	70	1	0	145	174	0	1	125	1	2.6	0	0	3	0
3	61	1	0	148	203	0	1	161	0	0.0	2	1	3	0
4	62	0	0	138	294	1	1	106	0	1.9	1	3	2	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1020	59	1	1	140	221	0	1	164	1	0.0	2	0	2	1
1021	60	1	0	125	258	0	0	141	1	2.8	1	1	3	0
1022	47	1	0	110	275	0	0	118	1	1.0	1	1	2	0
1023	50	0	0	110	254	0	0	159	0	0.0	2	0	2	1
1024	54	1	0	120	188	0	1	113	0	1.4	1	1	3	0

1025 rows × 14 columns

Figure 1: The Heart Disease Dataset

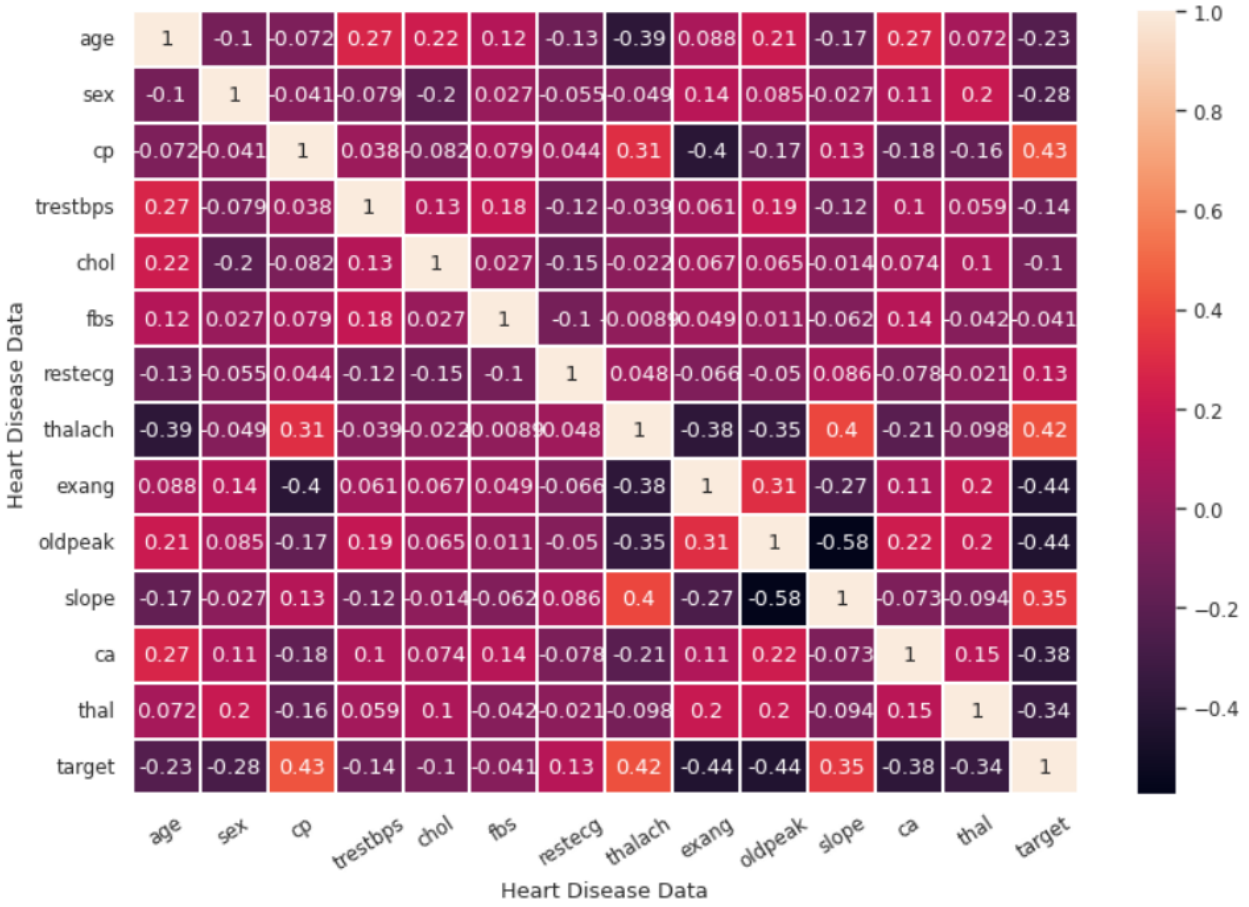

Figure 2: Heat map of Heart Disease Dataset

Methodology:

To read and load the Heart Disease dataset from Kaggle, we will first need to download the CSV file from the Kaggle website. Then, we can use the pandas' library to read the CSV file into a pandas DataFrame.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
import pandas as pd
path = "/content/drive/MyDrive/CSE366_Lab/heart.csv"

df= pd.read_csv(path)
```

Figure 3: Heart Disease Dataset Loading and reading

This code loads the Heart Disease dataset from the CSV file. Once we have loaded the dataset, we can perform various analyses, such as exploring the distribution of each feature, visualizing the correlation between features, and building predictive models.

At first using train_test_split from scikit learn, I split the dataset with test_size 0.15 which splits ta dataset into 85:15 ratios.

```
from sklearn.model_selection import train_test_split
newDF = df.copy()
x = newDF[['age', 'sex', 'cp', 'chol', 'fbs','thalach', 'target']]
y = newDF[['target']]

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.15, random_state = 0)
print(x_train)
print(y_train)
print(x_test)
print(y_test)
```

Figure 4: Splitting the dataset into 85:15 ration

Naive Bayesian classifier:

Naive Bayesian Classifier is Supervised machine learning Algorithm which is based on Bayes theorem with an assumption of independence among the features when class is given.

Algorithm: P(C|X) = P(X|C) * P(C)
Where:

- P(C|X) is the posterior probability of class C given the input features X.
- P(X|C) is the likelihood probability of observing the input features X given class C.
- P(C) is the prior probability of class C.
- P(X) is the probability of observing the input features X (also known as evidence).

In the context of the Naive Bayes classifier, the assumption of feature independence allows us to simplify the equation further. **Assuming X = (x1, x2, ..., xn), where x1, x2, ..., xn** are the individual features, the equation becomes:
P(C|X) = P(x1|C) * P(x2|C) * ... * P(xn|C) * P(C)

In my dataset the column "target" is the target class which is discreet feature.
I have chosen three discreet features and three continuous features.

Discreet features	**Continuous features**
Sex	Age
Cp	Chol
Fbs	Thalach

The prior probability calculation equation in Naive Bayes for both discreet and continuous feature is as follows:
Prior: P (Class = C) = Count (Class = C) / TotalCount

Likelihood Calculation:

1. Discrete feature likelihood

In Naive Bayes, the likelihood calculation for discrete feature values involves estimating the conditional probability of observing a specific value of a feature given a particular class. The likelihood is calculated as the ratio of the count of instances where the feature has the desired value and belongs to the given class, to the count of instances belonging to the given class.

Equation:

P (Feature = x | Class = C) = Count (Feature = x and Class = C) / Count (Class = C)

1. Count the number of instances in the training dataset that belong to the given class, denoted as Count (Class = C).
2. Count the number of instances in the training dataset where the feature has the desired value and belongs to the given class, denoted as Count (Feature = x and Class = C).
3. Calculate the likelihood probability by dividing the count from step 2 by the count from step 1 using above equation

This provides the estimated likelihood probability of observing the specific value x for the feature, given the class C.

2. Continuous feature likelihood

In Naive Bayes, the likelihood calculation for continuous feature values involves estimating the probability density function (PDF) or probability distribution of each feature given a particular class. This is typically done by assuming a specific distribution, such as the normal distribution, and estimating the parameters of that distribution (mean and variance) based on the training data.

The formula:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Where:

- f(x) is the Probability Density Function
- μ is the mean (or expected value) of the distribution
- σ is the standard deviation of the distribution
- π is the mathematical constant pi (3.14159...)

1. Separate the training instances into classes, creating subsets of instances for each class.
2. For each class C: a. Calculate the mean (μ) and variance (σ^2) of the feature within that class. b. Using the mean and variance, construct the probability density function (PDF) of the normal distribution for that feature within the class.

3. Given a new instance with a specific feature value x and class C, calculate the likelihood by evaluating the PDF of the normal distribution for that feature within class C at the given value x.

Posterior probability:

After calculating prior and likelihood we calculate posterior probability. The posterior probability in Naive Bayes refers to the updated probability distribution of the class labels given the observed features. It is calculated multiplying the prior probabilities and the likelihood probabilities.

The equation for calculating the posterior probability in Naive Bayes is as follows:

Equation:

P (Class = C | Features) = P (Class = C) * P (Features | Class = C)

1. Calculate P (Class = C) that is prior probability of class C.
2. Multiply P (Features | Class = C) which likelihood probability of the observed features given class C with prior probability

Implementation:

At first, I have defined a function ‘naive_bayeisian’, which takes the ‘x_train’ as a parameter to train the model, and ' age, sex, cp, chol, fbs, thalach, target as parameters

```
def naive_bayeisian(x_train,age,sex,cp,chol,fbs,thalach,target):
```

Figure 5: Defining Naive Bayesian function

Next, For Disease =1(patient has disease), I calculated prior probability for both Discrete and continuous features.

```
#For Disease : 1
#prior probability
prior =x_train[x_train['target']==1].count()[0] / len(x_train)
```

Figure 6: Calculating prior probability

Then calculated likelihood of all Discrete features.

```
#discreet feature
#likelihood
sex1 = x_train[(x_train['sex']==sex) & (x_train['target']==1)].count()[0] / x_train[x_train['target']==1].count()[0]
cp1= x_train[(x_train['cp']==cp) & (x_train['target']==1)].count()[0] / x_train[x_train['target']==1].count()[0]
fbs1= x_train[(x_train['fbs']==fbs) & (x_train['target']==1)].count()[0] / x_train[x_train['target']==1].count()[0]
```

Figure 7: Calculating likelihood

Next, using Probability Density Function, I calculated likelihood for all continuous Features

```
#continuous feaure
#likelihood

 age_mean1 = x_train[x_train['target'] == 1]['age'].mean()
 age_std1 = x_train[x_train['target'] == 1]['age'].std()
 age1 = (1 / (np.sqrt(2 * np.pi) * age_std1)) * np.exp(-((age - age_mean1) ** 2 / (2 * age_std1 ** 2)))
```

Figure 8: Calculating likelihood for age

Similarly, I calculated for other continuous features Chol and Thalach. Then calculated posterior probability and stored it in target_1.

```
#Posterior probability for 1
target_1 = sex1 * cp1 *fbs1 * age1 *  chol1 * thalach1 * prior
```

Figure 9: Calculating posterior probability

Next, For Disease =0(patient has no disease), I calculated prior probability for both Discrete and continuous features.

```
#For Disease: 0
#calculate prior probability
 prior2 = x_train[x_train['target']==0].count()[0] / len(x_train)
```

Figure 10: Calculating prior probability

Then calculated likelihood of all Discrete features.

```
#discreet feature
#Likelihood
 sex = x_train[(x_train['sex']==sex) & (x_train['target']==0)].count()[0] / x_train[x_train['target']==0].count()[0]
 cp= x_train[(x_train['cp']==cp) & (x_train['target']==0)].count()[0] / x_train[x_train['target']==0].count()[0]
 fbs= x_train[(x_train['fbs']==fbs) & (x_train['target']==0)].count()[0] /x_train[x_train['target']==0].count()[0]
```

Figure 11: Calculating likelihood

Next, using Probability Density Function, I calculated likelihood for all continuous Features

```
#continuous feature
# likelihood

 age_mean = x_train[x_train['target'] == 0]['age'].mean()
 age_std = x_train[x_train['target'] == 0]['age'].std()
 age = (1 / (np.sqrt(2 * np.pi) * age_std)) * np.exp(-((age - age_mean) ** 2 / (2 * age_std** 2)))
```

Figure 12: Calculating likelihood for age

Similarly, I calculated for other continuous features Chol and Thalach. Then calculated posterior probability and stored in in target_0.

```
#Posterior probability for 0
target_0 = sex * cp * fbs * age *  chol * thalach * prior2
```

Figure 13: Calculating posterior probability

Next by comparing the posterior probability for both Disease 1 (**patient has disease**) and 0 (**patient has no disease**), made decision in favor of the maximum probability.

```
if target_1 < target_0:
   return 0
else:
   return 1
```

Figure 14: Comparing posterior probabilities of both classes

Next, by iterating through all the rows of x_test, using naive_bayeisian function, made prediction using the x_train and features from x_test and stored values in `target_pred` list

```
target_pred = []

for i, data in x_test.iterrows():
  age = data['age']
  sex = data['sex']
  cp = data['cp']
  chol = data['chol']
  fbs = data['fbs']
  thalach = data['thalach']
  target = data['target']

  pred = naive_bayeisian(x_train,age,sex,cp,chol,fbs,thalach,target)
  target_pred.append(pred)
```

Figure 15: Making Prediction

Next, using accuracy_score from sklearn.metrics, calculated the accuracy

```
from sklearn.metrics import confusion_matrix
y_test_arr = y_test['target'].values
y_pred = np.array(target_pred)
con_mat = confusion_matrix(y_test_arr, y_pred)
print(con_mat)
```

```
[[64 14]
 [15 61]]
```

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test_arr, y_pred)
print(f"Accuracy: {accuracy*100}%")
```

```
Accuracy: 81.16883116883116%
```

Figure 16: Confusion Matrix and Accuracy for Naive Bayesian classifier

Single Layer Perceptron:

A single layer perceptron (SLP) is a type of artificial neural network (ANN) architecture consisting of a single layer of artificial neurons and can only classify linearly separable cases with a binary target (1, 0).
In SLP, we have a function f(x) which calculates the weighted sum of the input features X,

with weight vector W. Also, we have an activation function A(X), which returns 1 if X>0 and 0 otherwise. And, we have a classifier A(f(x)), which gives the value of the Predicted class

```
#equation
def f(X, W):
  return W[0] + np.dot(X, W[1:])

#activation function
def A(x):
  if x>0:
    return 1
  else:
    return 0

#classifier
def Y(X, W):
  return A(f(X,W))
```

Figure 17: f (X, W), A(x), and Y (X, W)

Next, I have defined a function train, which takes x_train and y_train as arguments. Next, I have initialized W with zeros and set learning rate as 0.05 and set epoch as 0. Also, I initialized 2 additional variables, prev_mse and best_weights to keep track at which the weight stops decreasing. Next, I run a while loop, which runs until the MSE does not changes. The function calls the classifier and make prediction. After every prediction, delw, W [0], and W [1:] is updated and calculated MSE. When MSE stops changing and remains same, I have copied the weights into a variable best_weights and MSE into a variable best_mse and returned these variables.

```
def train(x_train, y_train):
  W = np.zeros(x_train.shape[1] + 1)
  eta = 0.05 #learning rate
  epoch = 0
  prev_mse = np.inf
  best_weights = None

  while True:
    target_pred_p = []
    for i in range(len(x_train)):
      X = x_train.iloc[i].values
      Ti = y_train[i][0]
      Pi = Y(X, W)
      target_pred_p.append(Pi)
      delw = eta * (Ti - Pi)
      W[0] += delw * 1
      W[1:] += delw * X

    #Mean squared error

    MSE = np.mean((np.array(target_pred_p) - y_train) ** 2)

    epoch += 1

    print(f"Epoch {epoch}: MSE = {MSE}")
    if epoch > 1 and MSE >= prev_mse:
      break
    prev_mse = MSE

    if best_weights is None or MSE<best_mse:
      best_weights = np.copy(W)
      best_mse = MSE

  return best_weights,best_mse
```

Figure 18: Train Perceptron

Next, I have defined a function named test_accuracy which takes x_test, y_test, and W as argument. Inside this, I initialized a counter named `accurate_preds` as 0. For each index on x_test, I checked the Target values and predicted values. If the prediction is correct, I have updated the counter variable. Finally, I have calculated the accuracy by dividing the counter variable by length of the x_test. And, finally I have returned the accuracy.

```
def test_accuracy(x_test, y_test, W):
  accurate_preds = 0
  for i in range(len(x_test)):
    X = x_test.iloc[i].values
    Ti = y_test[i][0]
    Pi = Y(X, W)
    if Pi == 1 and Ti == 1:
      accurate_preds += 1
    elif Pi == 0 and Ti == 0:
      accurate_preds += 1

  accuracy = accurate_preds/len(x_test)
  return accuracy
```

Figure 19: Test Perceptron and calculate accuracy

Next, by calling train and test_accuracy, I have trained the model and calculated the accuracy.

```
weight, train_mse = train(x_train, y_train)
accuracy_p = test_accuracy(x_test, y_test, weight)
acc_pre = accuracy_p * 100
print(f"Train MSE: {train_mse}")
print(f"Test Accuracy: {acc_pre}%")
print(f"Learned Weights: {weight}")
```

```
Epoch 1: MSE = 0.49894877814407607
Epoch 2: MSE = 0.4991399093906077
Train MSE: 0.49894877814407607
Test Accuracy: 69.48051948051948%
Learned Weights: [ -0.65 -51.45  -3.65   6.7   -2.55  -1.1   19.25   9.7 ]
```

Figure 20: Outputs of SLP

Results:

For **Naive Bayesian Classifier**, my model is giving an accuracy of **82.46753246753246%**

```
[92] from sklearn.metrics import confusion_matrix
     y_test_arr = y_test['target'].values
     y_pred = np.array(target_pred)
     con_mat = confusion_matrix(y_test_arr, y_pred)
     print(con_mat)
```

```
[[61 17]
 [10 66]]
```

```
[93] from sklearn.metrics import accuracy_score
     accuracy = accuracy_score(y_test_arr, y_pred)
     print(f"Accuracy: {accuracy*100}%")
```

```
Accuracy: 82.46753246753246%
```

Figure 21: Results of Naive Bayesian Classifier

For **Single Layer Perceptron (SLP)**, my model is giving an accuracy of **51.298701298701296%**

```
[98] weight, train_mse = train(x_train, y_train)
     accuracy_p = test_accuracy(x_test, y_test, weight)
     acc_pre = accuracy_p * 100
     print(f"Train MSE: {train_mse}")
     print(f"Test Accuracy: {acc_pre}%")
     print(f"Learned Weights: {weight}")
```

```
Epoch 1: MSE = 0.49894877814407607
Epoch 2: MSE = 0.4991399093906077
Train MSE: 0.4991399093906077
Test Accuracy: 51.298701298701296%
Learned Weights: [ -1.05 -70.4   -7.35  13.2  -10.8   -1.5   21.2   19.35]
```

Figure 22: Results of SLP

Conclusion:

In this experiment, the Naive Bayesian classifier achieved 82.46753246753246% accuracy, while the Single Layer perceptron achieved 51.298701298701296% accuracy. The Naive Bayesian classifier performed better than the perceptron in terms of accuracy. Naive Bayes and Single-Layer

Perceptron (SLP) are both machine learning algorithms commonly used for classification tasks. The Naïve Bayesian is Based on Bayes' theorem, it calculates the posterior probability of a class given the observed features whereas SLP is Based on the concept of a single-layer neural network, it learns to classify data by adjusting weights to minimize errors.

Naive Bayesian assumes feature independence, meaning that features are assumed to be conditionally independent given the class label which can be a limitation if there are strong relationships between the features. And SLP does not make any assumptions about feature independence.

Here in the Naive Bayesian achieved an accuracy of 82.46753246753246%, which is higher than the SLP. This suggests that the data might not be linearly separable or that the linear model was not able to capture the underlying patterns in the dataset properly. To improve the accuracy, several approaches can be taken such as properly evaluating the dataset. If the variance is high, I can obtain more data after analysis. I can test out various functionalities. Smaller sets of features can help to fix high variance, and adding features can help to fix high bias. Hyper parameter tuning, on the other hand, can be useful. When bias is high, regularization lowers, and when variance is high, regularization increases.

In conclusion, in this project I implemented naïve Bayesian and SLP classifier. Then compared their accuracy. The Naive Bayesian classifier outperformed the single-layer perceptron in terms of accuracy. Various methods, including feature engineering, model selection, and data processing techniques, might be taken to increase accuracy. Proper analysis of the dataset and assessment of the model’s assumptions and constraints are required to get accurate results.