When dealing with machine learning problems, there are generally two types of data (and machine learning models):

- Supervised data: always has one or multiple targets associated with it.
- Unsupervised data: does not have any target variable.

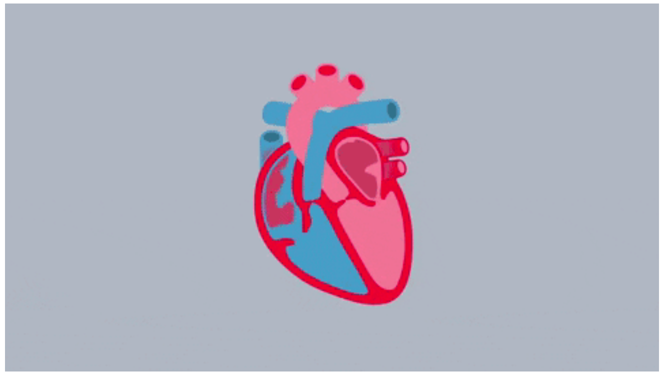
A supervised problem is considerably easier to tackle than an unsupervised one. A problem in which we are required to predict a value is known as a supervised problem. For example, if the problem is to predict house prices given historical house prices, with features like presence of a hospital, school or supermarket, distance to nearest public transport, etc. is a upervised problem. Similarly, when we are provided with images of cats and dogs, and we know beforehand which ones are cats and which ones are dogs, and if the task is to create a model which predicts whether a provided image is of a cat or a dog, the problem is considered to be supervised.

Here in this Dataset we have a Supervised Machine Learning Problem, For Heart Failure Prediction

Importing all the libraries needed

In [1]:
```
import os
import numpy as np
import pandas as pd
import warnings
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px
warnings.filterwarnings("ignore")
pd.set_option("display.max_rows",None)
from sklearn import preprocessing
import matplotlib
matplotlib.style.use('ggplot')
from sklearn.preprocessing import LabelEncoder
```

Context



Cardiovascular diseases (CVDs) are the number 1 cause of death globally, taking an estimated 17.9 million lives each year, which accounts for 31% of all deaths worldwide. Four out of 5CVD deaths are due to heart attacks and strokes, and one-third of these deaths occur prematurely in people under 70 years of age. Heart failure is a common event caused by CVDs and this dataset contains 11 features that can be used to predict a possible heart disease.

People with cardiovascular disease or who are at high cardiovascular risk (due to the presence of one or more risk factors such as hypertension, diabetes, hyperlipidaemia or already established disease) need early detection and management wherein a machine learning model can be of great help

In [2]:
```
df=pd.read_csv("D:\GradProjHeartAttackModel\heart1.csv")
df.head()
```

Out[2]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
0	62	F	TA	140	268	0	Normal	160	N	3.6	Down	0
1	62	F	TA	160	164	0	Normal	145	N	6.2	Down	0
2	56	F	TA	200	288	1	Normal	133	Y	4.0	Down	0
3	53	M	TA	140	203	1	Normal	155	Y	3.1	Down	0
4	59	M	TA	170	326	0	Normal	140	Y	3.4	Down	0

The describe() function in pandas is very handy in getting various summary statistics.This function returns the count, mean, standard deviation, minimum and maximum values and the quantiles of the data.

In [3]:
```
df.dtypes
```

Out[3]:
```
Age                 int64
Sex                object
ChestPainType      object
RestingBP           int64
Cholesterol         int64
FastingBS           int64
RestingECG         object
MaxHR               int64
ExerciseAngina     object
Oldpeak           float64
ST_Slope           object
HeartDisease        int64
dtype: object
```

As we can see the string data in the dataframe is in the form of object, we need to convert it back to string to work on it

In [4]:
```
string_col = df.select_dtypes(include="object").columns
df[string_col]=df[string_col].astype("string")
```

In [5]:
```
df.dtypes
```

Out[5]:
```
Age                 int64
Sex                string
ChestPainType      string
RestingBP           int64
Cholesterol         int64
FastingBS           int64
RestingECG         string
MaxHR               int64
ExerciseAngina     string
Oldpeak           float64
ST_Slope           string
HeartDisease        int64
dtype: object
```

So, as we can see here the object data has been converted to string

Getting the categorical columns

In [6]:
```
string_col=df.select_dtypes("string").columns.to_list()
```

In [7]:
```
num_col=df.columns.to_list()
#print(num_col)
for col in string_col:
    num_col.remove(col)
num_col.remove("HeartDisease")
```

In [8]:
```
df.describe().T
```

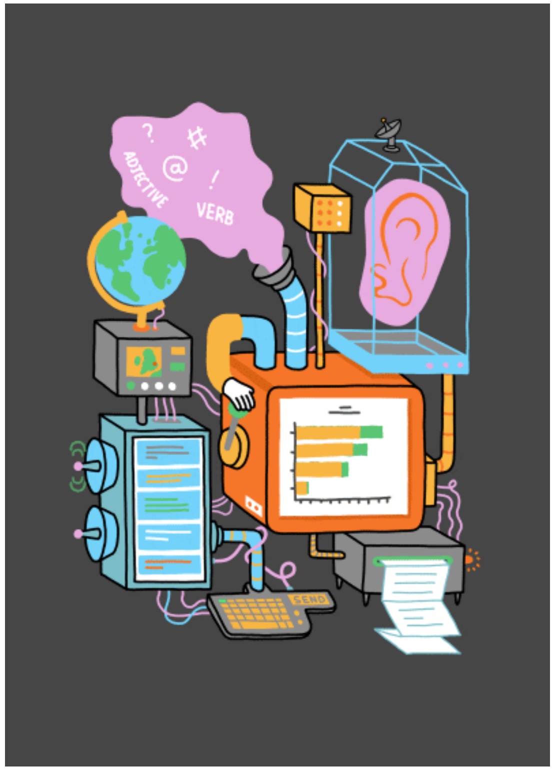
Out[8]:

	count	mean	std	min	25%	50%	75%	max
Age	303.0	54.366337	9.082101	29.0	47.5	55.0	61.0	77.0
RestingBP	303.0	131.623762	17.538143	94.0	120.0	130.0	140.0	200.0
Cholesterol	303.0	246.264026	51.830751	126.0	211.0	240.0	274.5	564.0
FastingBS	303.0	0.148515	0.356198	0.0	0.0	0.0	0.0	1.0
MaxHR	303.0	149.646865	22.905161	71.0	133.5	153.0	166.0	202.0
Oldpeak	303.0	1.039604	1.161075	0.0	0.0	0.8	1.6	6.2
HeartDisease	303.0	0.544554	0.498835	0.0	0.0	1.0	1.0	1.0

The Attributess include:

- Age: age of the patient [years]
- Sex: sex of the patient [M: Male, F: Female]
- ChestPainType: chest pain type [TA: Typical Angina, ATA: Atypical Angina, NAP: Non-Anginal Pain, ASY: Asymptomatic]
- RestingBP: resting blood pressure [mm Hg]
- Cholesterol: serum cholesterol [mm/dl]
- FastingBS: fasting blood sugar [1: if FastingBS > 120 mg/dl, 0: otherwise]
- RestingECG: resting electrocardiogram results [Normal: Normal, ST: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV), LVH: showing probable or definite left ventricular hypertrophy by Estes' criteria]
- MaxHR: maximum heart rate achieved [Numeric value between 60 and 202]
- ExerciseAngina: exercise-induced angina [Y: Yes, N: No]
- Oldpeak: oldpeak = ST [Numeric value measured in depression]
- ST_Slope: the slope of the peak exercise ST segment [Up: upsloping, Flat: flat, Down: downsloping]
- HeartDisease: output class [1: heart disease, 0: Normal]

Exploratory Data Analysis



First Question should be why do we need this ??

Out Come of this phase is as given below :

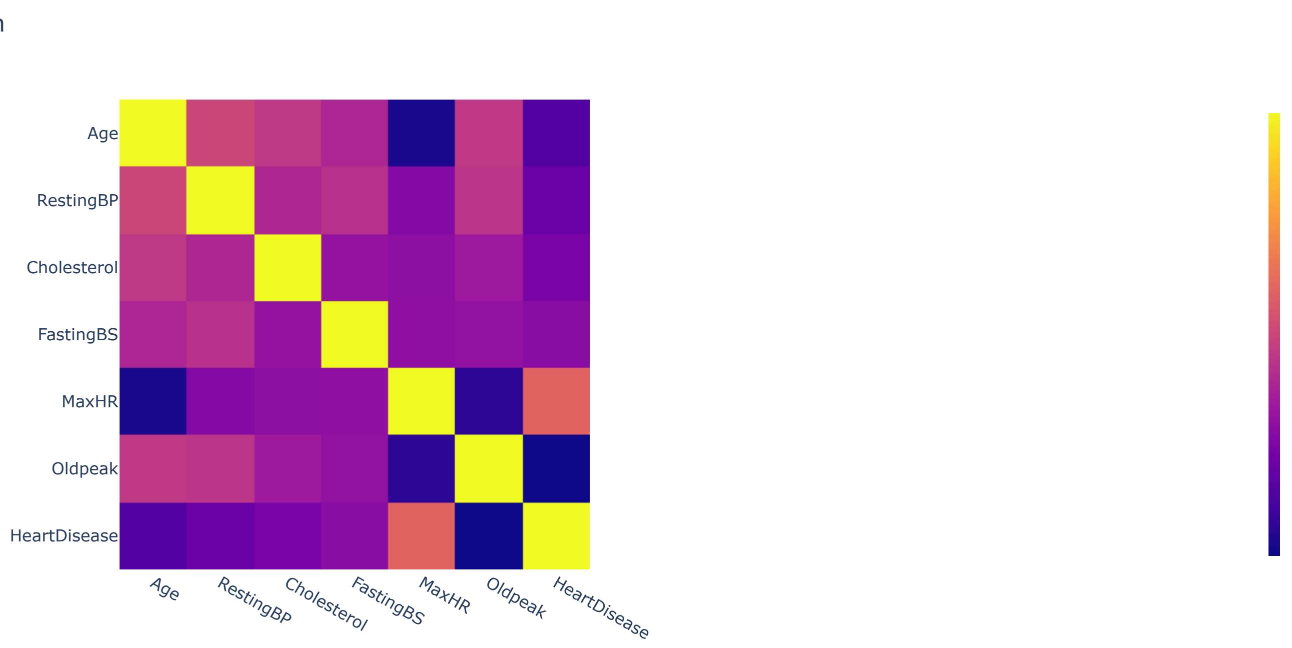
- Understanding the given dataset and helps clean up the given dataset.
- It gives you a clear picture of the features and the relationships between them.
- Providing guidelines for essential variables and leaving behind/removing non-essential variables.
- Handling Missing values or human error.
- Identifying outliers.
- EDA process would be maximizing insights of a dataset.
- This process is time-consuming but very effective,

Correlation Matrix

Its necessary to remove correlated variables to improve your model.One can find correlations using pandas ".corr()" function and can visualize the correlation matrix using plotly express.

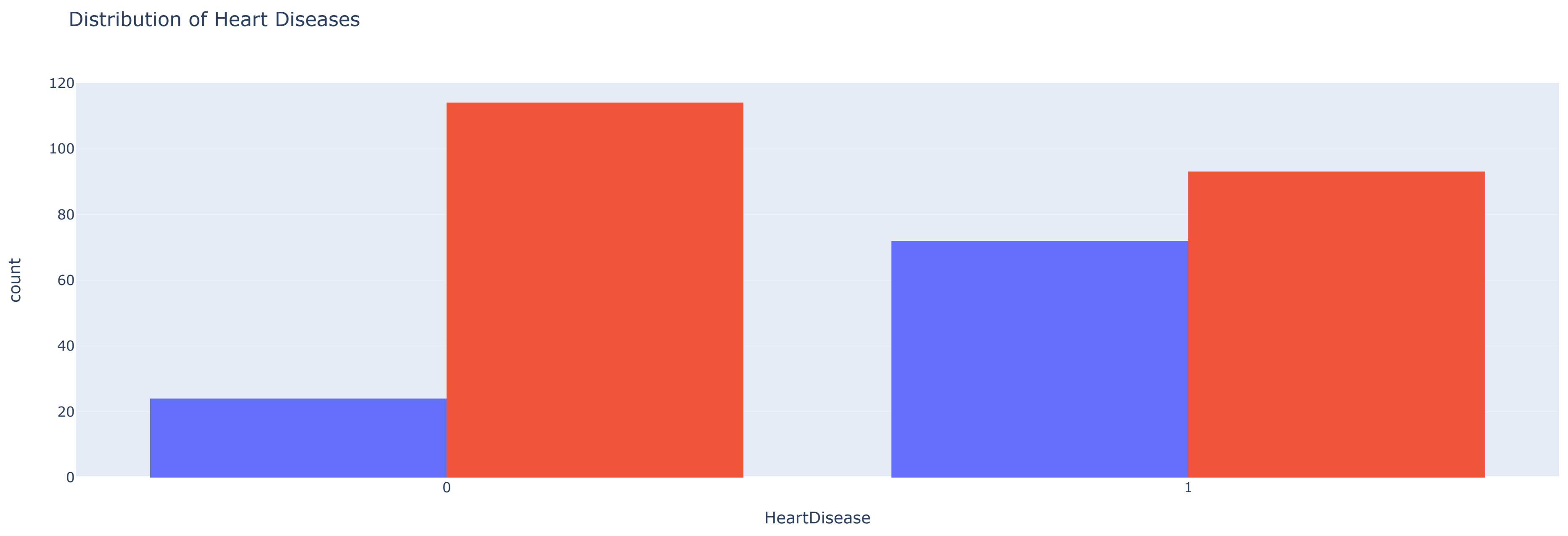
- Lighter shades represents positive correlation
- Darker shades represents negative correlation

In [9]:
```python
px.imshow(df.corr(),title="Correlation Plot of the Heat Failure Prediction")
```

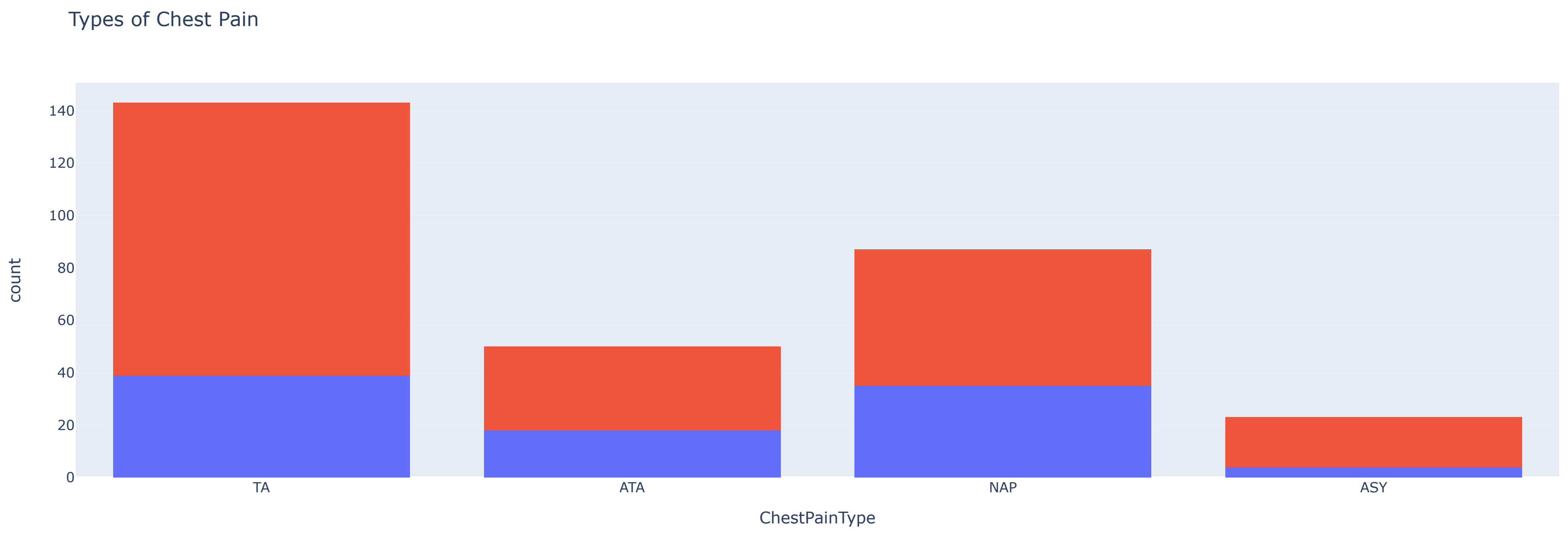


Here we can see Heart Disease has a high negative correlation with "MaxHR" and somewhat negative correlation wiht "Cholesterol", where as here positive correatlation with "Oldpeak","FastingBS" and "RestingBP"

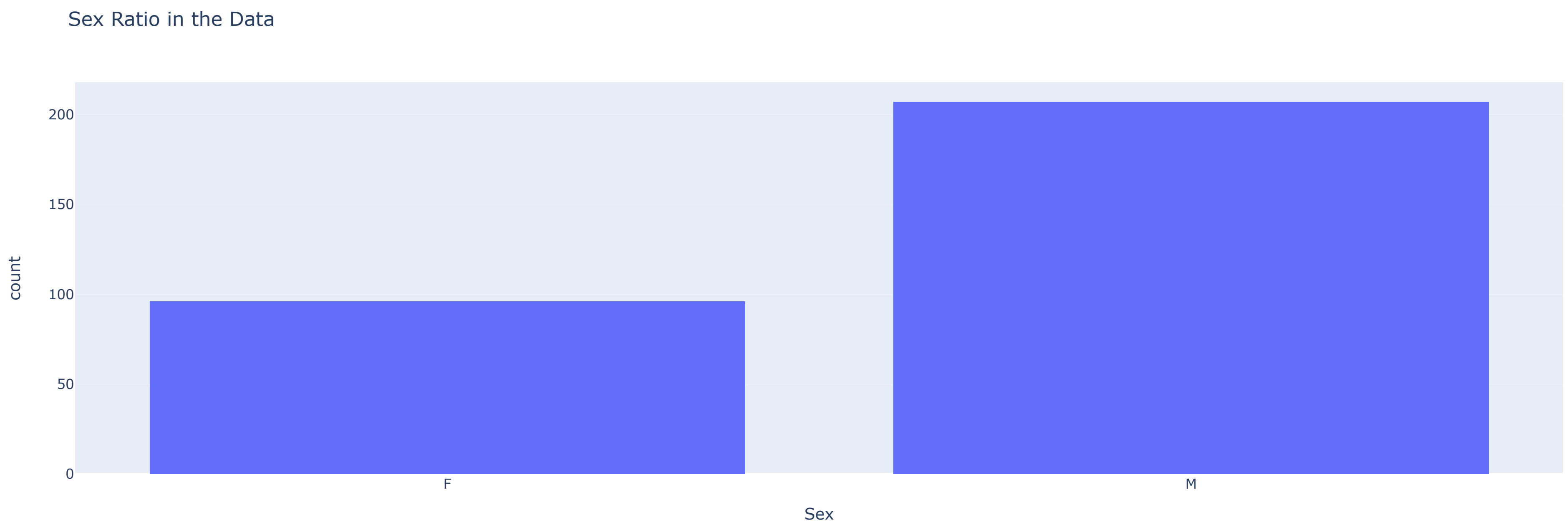
In [10]:
```python
# Shows the Distribution of Heat Diseases with respect to male and female
fig=px.histogram(df,
                 x="HeartDisease",
                 color="Sex",
                 hover_data=df.columns,
                 title="Distribution of Heart Diseases",
                 barmode="group")
fig.show()
```



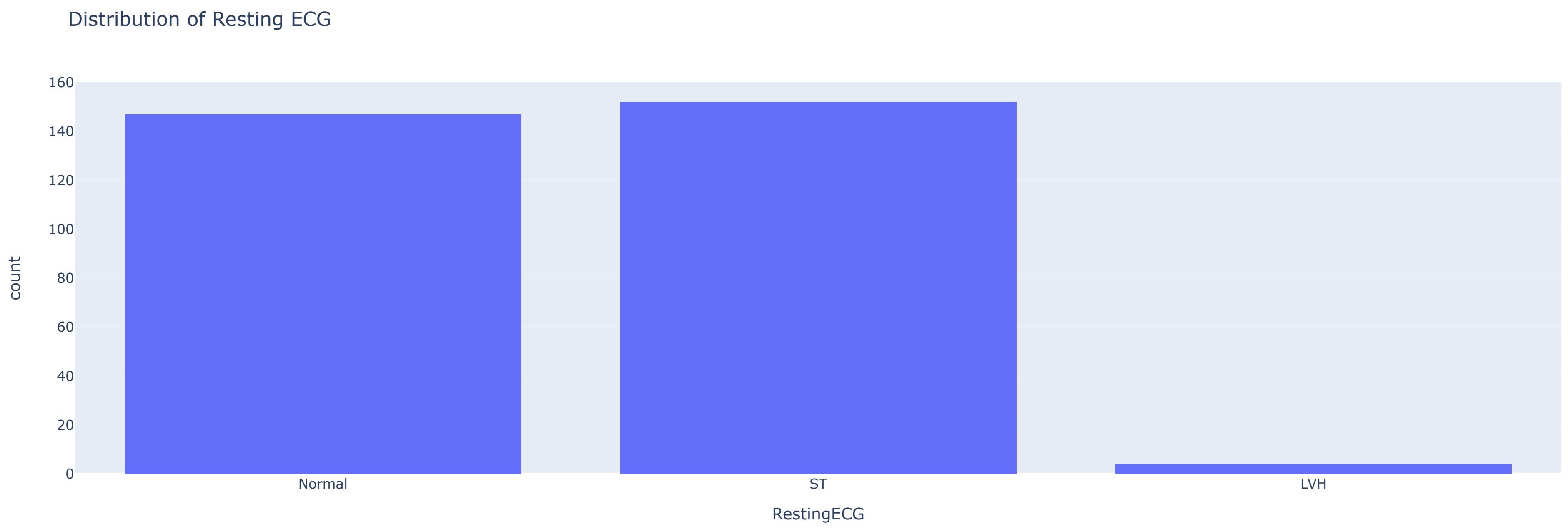
```
In [11]: fig=px.histogram(df,
                          x="ChestPainType",
                          color="Sex",
                          hover_data=df.columns,
                          title="Types of Chest Pain"
                         )
         fig.show()
```



```
In [12]: fig=px.histogram(df,
                          x="Sex",
                          hover_data=df.columns,
                          title="Sex Ratio in the Data")
         fig.show()
```



```
In [13]: fig=px.histogram(df,
                          x="RestingECG",
                          hover_data=df.columns,
                          title="Distribution of Resting ECG")
         fig.show()
```

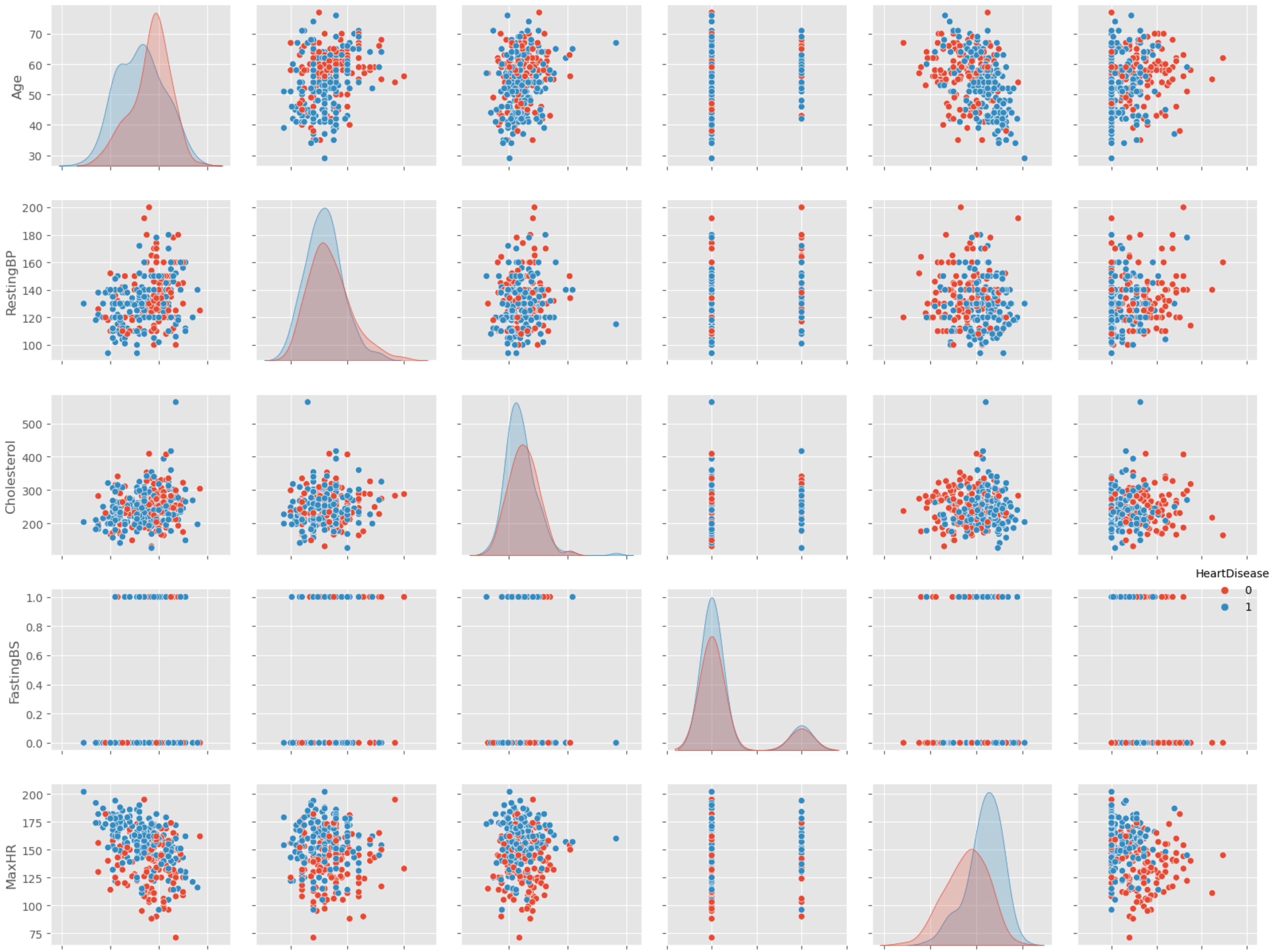


To plot multiple pairwise bivariate distributions in a dataset, you can use the pairplot() function. This shows the relationship for (n, 2) combination of variable in a DataFrame as a matrix of plots and the diagonal plots are the univariate plots.

In [14]:
```
plt.figure(figsize=(15,10))
sns.pairplot(df,hue="HeartDisease")
plt.title("Looking for Insites in Data")
plt.legend("HeartDisease")
plt.tight_layout()
plt.plot()
```

Out[14]:
```
[]
```

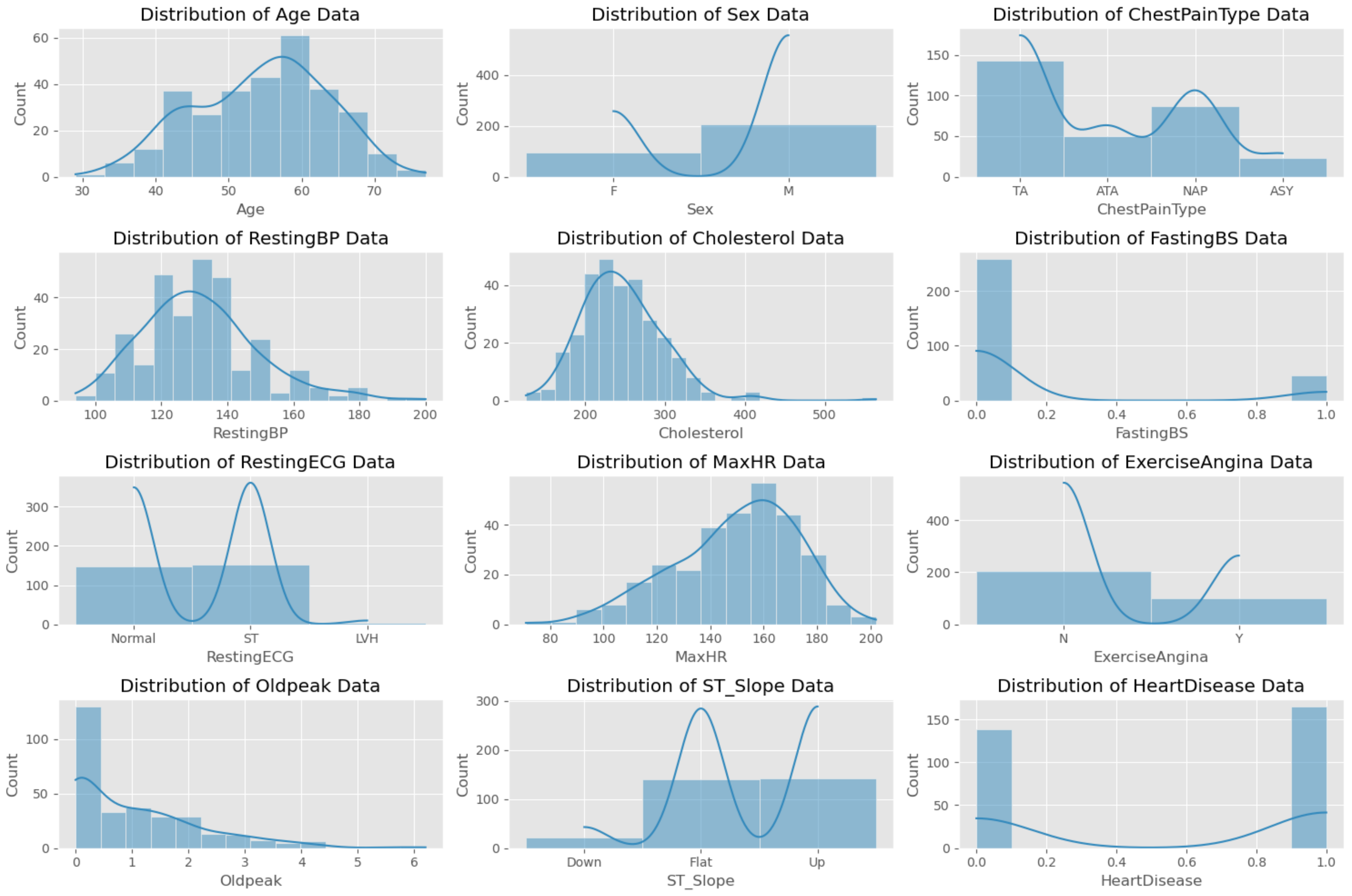
```
<Figure size 1500x1000 with 0 Axes>
```



Age
RestingBP
Cholesterol
FastingBS
MaxHR
HeartDisease
0
1

Now to check the linearity of the variables it is a good practice to plot distribution graph and look for skewness of features. Kernel density estimate (kde) is a quite useful tool for plotting the shape of a distribution.

In [15]:
```
plt.figure(figsize=(15,10))
for i,col in enumerate(df.columns,1):
    plt.subplot(4,3,i)
    plt.title(f"Distribution of {col} Data")
    sns.histplot(df[col],kde=True)
    plt.tight_layout()
    plt.plot()
```


Distribution of Age Data
Age
Distribution of Sex Data
Sex
Distribution of ChestPainType Data
ChestPainType
Distribution of RestingBP Data
RestingBP
Distribution of Cholesterol Data
Cholesterol
Distribution of FastingBS Data
FastingBS
Distribution of RestingECG Data
RestingECG
Distribution of MaxHR Data
MaxHR
Distribution of ExerciseAngina Data
ExerciseAngina
Distribution of Oldpeak Data
Oldpeak
Distribution of ST_Slope Data
ST_Slope
Distribution of HeartDisease Data
HeartDisease
Count

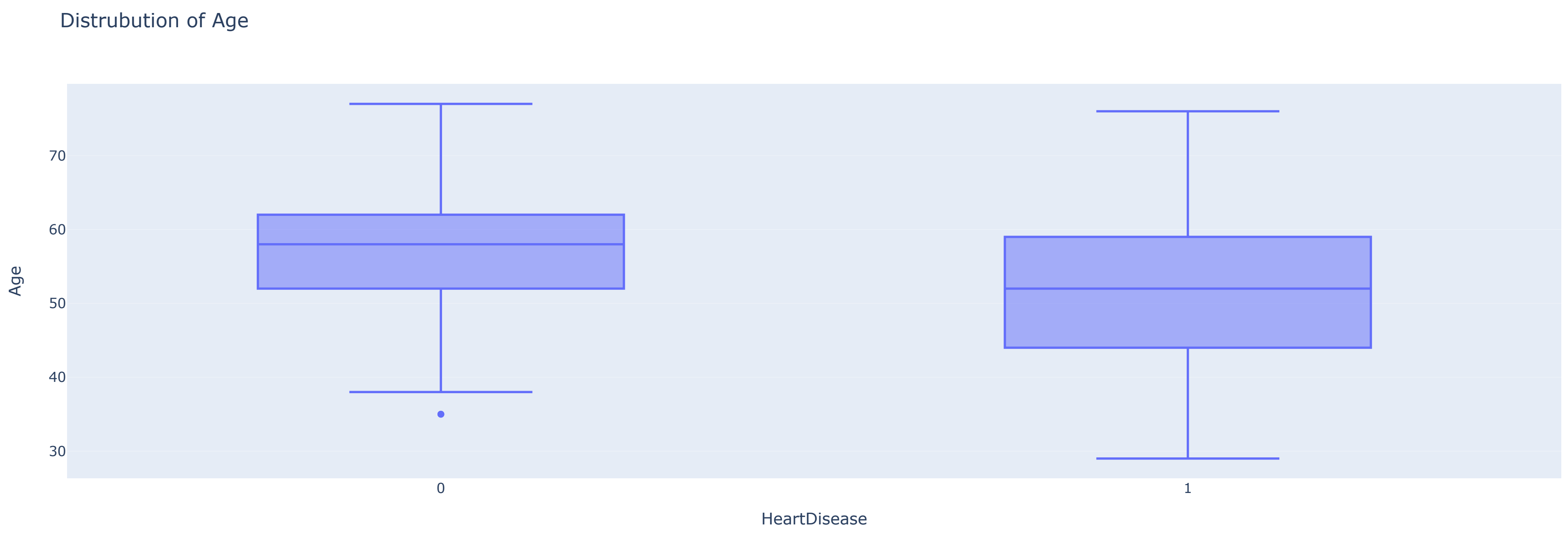
Outliers

A box plot (or box-and-whisker plot) shows the distribution of quantitative data in a way that facilitates comparisons between variables.The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution.The box plot (a.k.a. box and whisker diagram) is a standardized way of displaying the distribution of data based on the five number summary:

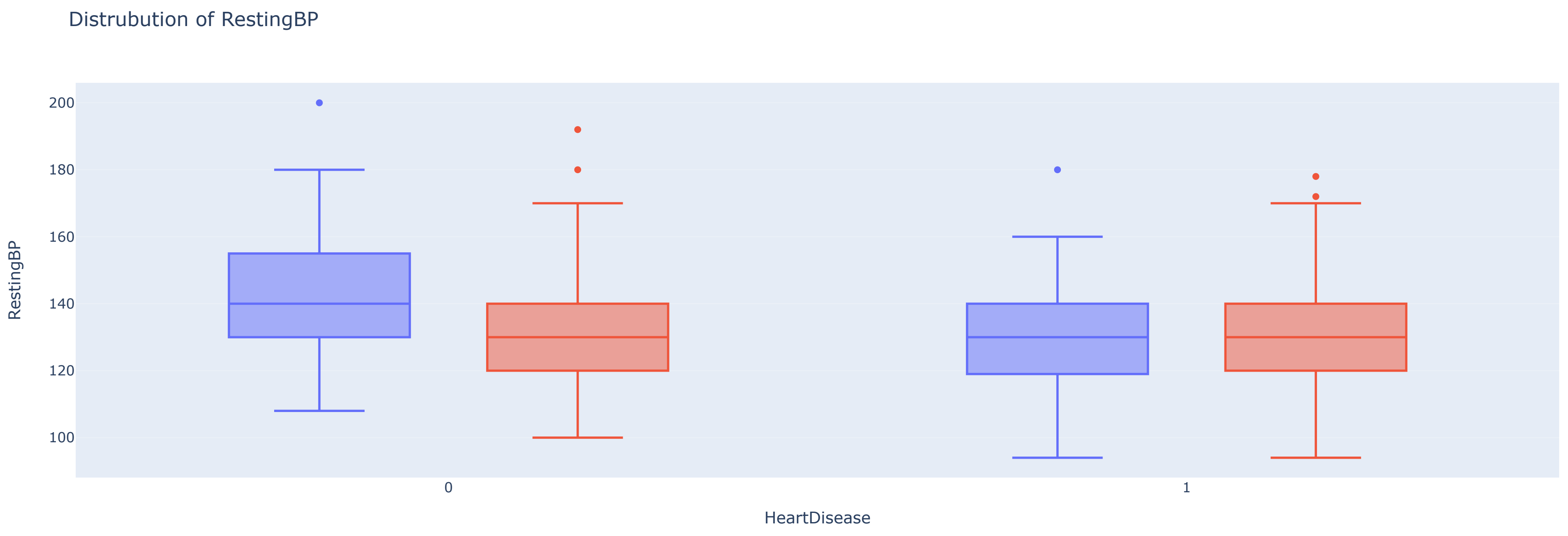
- Minimum
- First quartile
- Median
- Third quartile
- Maximum.

In the simplest box plot the central rectangle spans the first quartile to the third quartile (the interquartile range or IQR).A segment inside the rectangle shows the median and “whiskers” above and below the box show the locations of the minimum and maximum.

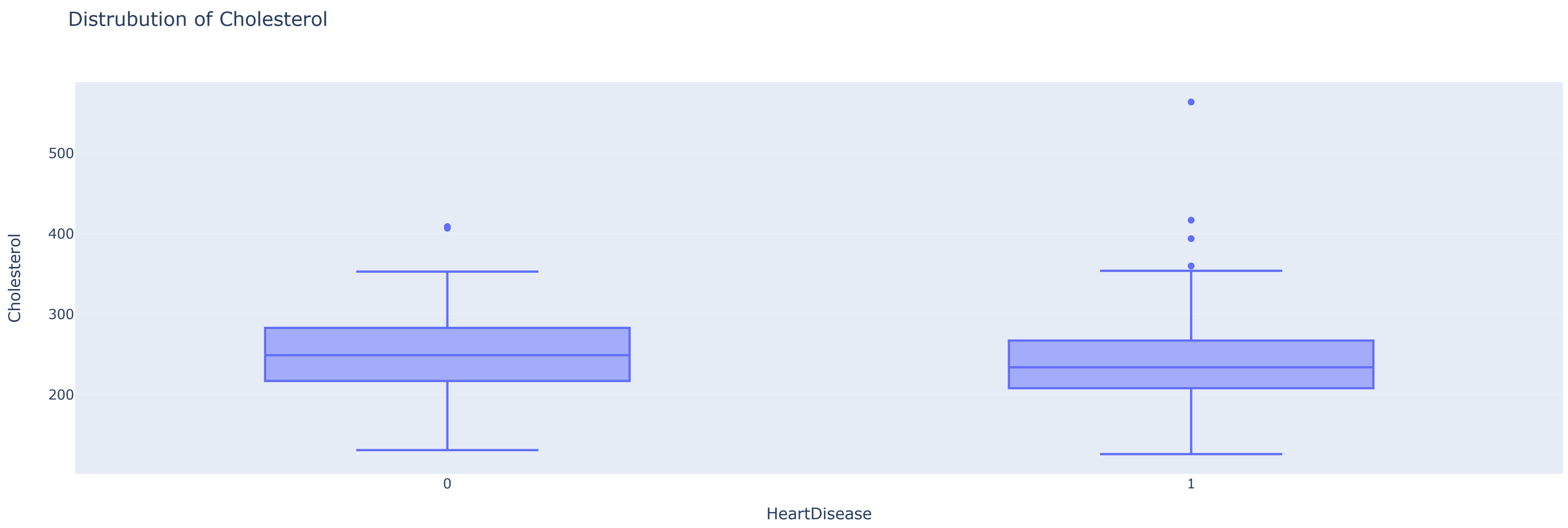
In [16]:
```
fig = px.box(df,y="Age",x="HeartDisease",title=f"Distrubution of Age")
fig.show()
```



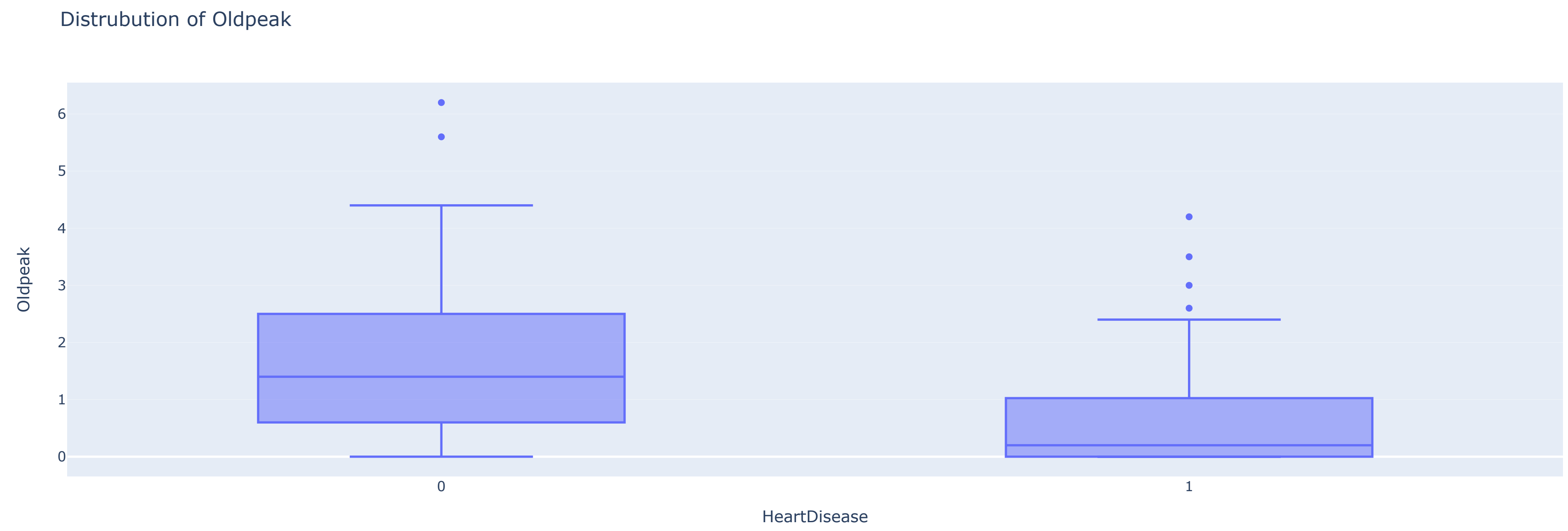
In [17]:
```
fig = px.box(df,y="RestingBP",x="HeartDisease",title=f"Distrubution of RestingBP",color="Sex")
fig.show()
```



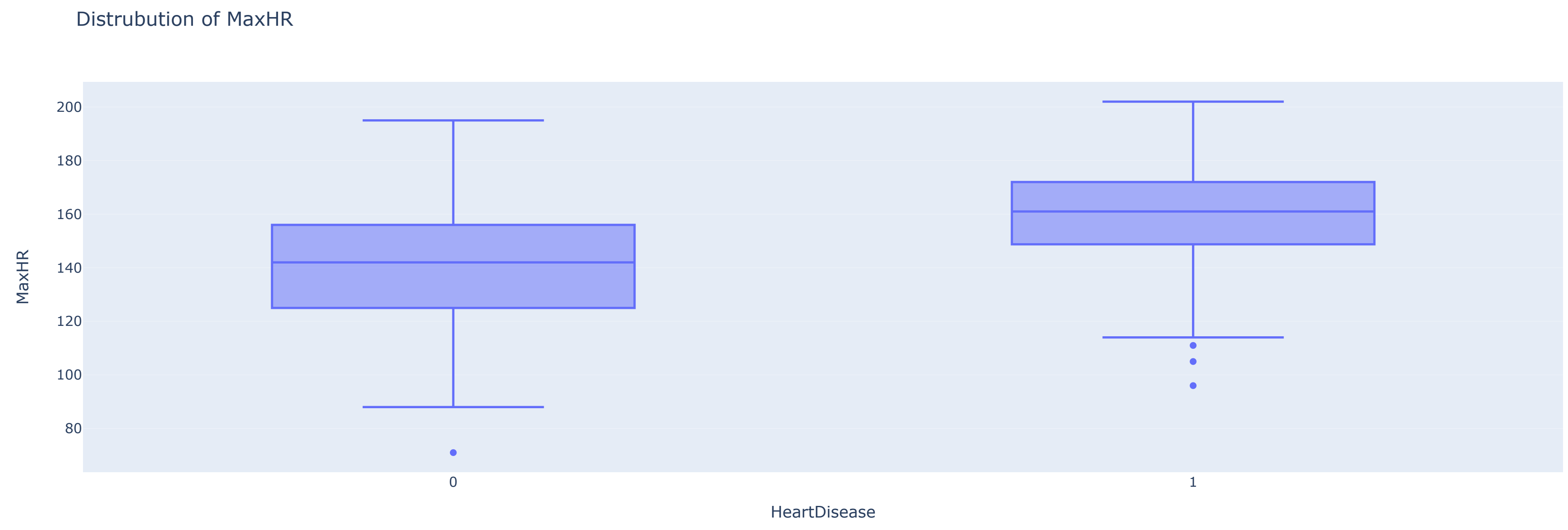
```
In [18]: fig = px.box(df,y="Cholesterol",x="HeartDisease",title=f"Distrubution of Cholesterol")
         fig.show()
```



```
In [19]: fig = px.box(df,y="Oldpeak",x="HeartDisease",title=f"Distrubution of Oldpeak")
         fig.show()
```



```
In [20]: fig = px.box(df,y="MaxHR",x="HeartDisease",title=f"Distrubution of MaxHR")
         fig.show()
```



Data Preprocessing

Data preprocessing is an integral step in Machine Learning as the quality of data and the useful information that can be derived from it directly affects the ability of our model to learn; therefore, it is extremely important that we preprocess our data before feeding it into our model.

The concepts that I will cover in this article are

1. Handling Null Values
2. Feature Scaling
3. Handling Categorical Variables

1. Handling Null Values :

In any real-world dataset, there are always few null values. It doesn't really matter whether it is a regression, classification or any other kind of problem, no model can handle these NULL or NaN values on its own so we need to intervene.

> In python NULL is reprsented with NaN. So don't get confused between these two,they can be used interchangably.

In [21]:
```
# Checking for Type of data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Age             303 non-null    int64  
 1   Sex             303 non-null    string 
 2   ChestPainType   303 non-null    string 
 3   RestingBP       303 non-null    int64  
 4   Cholesterol     303 non-null    int64  
 5   FastingBS       303 non-null    int64  
 6   RestingECG      303 non-null    string 
 7   MaxHR           303 non-null    int64  
 8   ExerciseAngina  303 non-null    string 
 9   Oldpeak         303 non-null    float64
 10  ST_Slope        303 non-null    string 
 11  HeartDisease    303 non-null    int64  
dtypes: float64(1), int64(6), string(5)
memory usage: 28.5 KB
```

In [22]:
```
# Checking for NULLs in the data
df.isnull().sum()
```

Out[22]:
```
Age               0
Sex               0
ChestPainType     0
RestingBP         0
Cholesterol       0
FastingBS         0
RestingECG        0
MaxHR             0
ExerciseAngina    0
Oldpeak           0
ST_Slope          0
HeartDisease      0
dtype: int64
```

So we can see our data does not have any null values but in case we have missing values, we can remove the data as well.

However, it is not the best option to remove the rows and columns from our dataset as it can result in significant information loss. If you have 300K data points then removing 2–3 rows won't affect your dataset much but if you only have 100 data points and out of which 20 have NaN values for a particular field then you can't simply drop those rows. In real-world datasets, it can happen quite often that you have a large number of NaN values for a particular field. Ex — Suppose we are collecting the data from a survey, then it is possible that there could be an optional field which let's say 20% of people left blank. So when we get the dataset then we need to understand that the remaining 80% of data is still useful, so rather than dropping these values we need to somehow substitute the missing 20% values. We can do this with the help of Imputation.

Imputation:

Imputation is simply the process of substituting the missing values of our dataset. We can do this by defining our own customised function or we can simply perform imputation by using the SimpleImputer class provided by sklearn.

For example :

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
imputer = imputer.fit(df[['Weight']])
df['Weight'] = imputer.transform(df[['Weight']])
```

As we do not have any missing data so we will not be using this approch

2. Feature Scaling

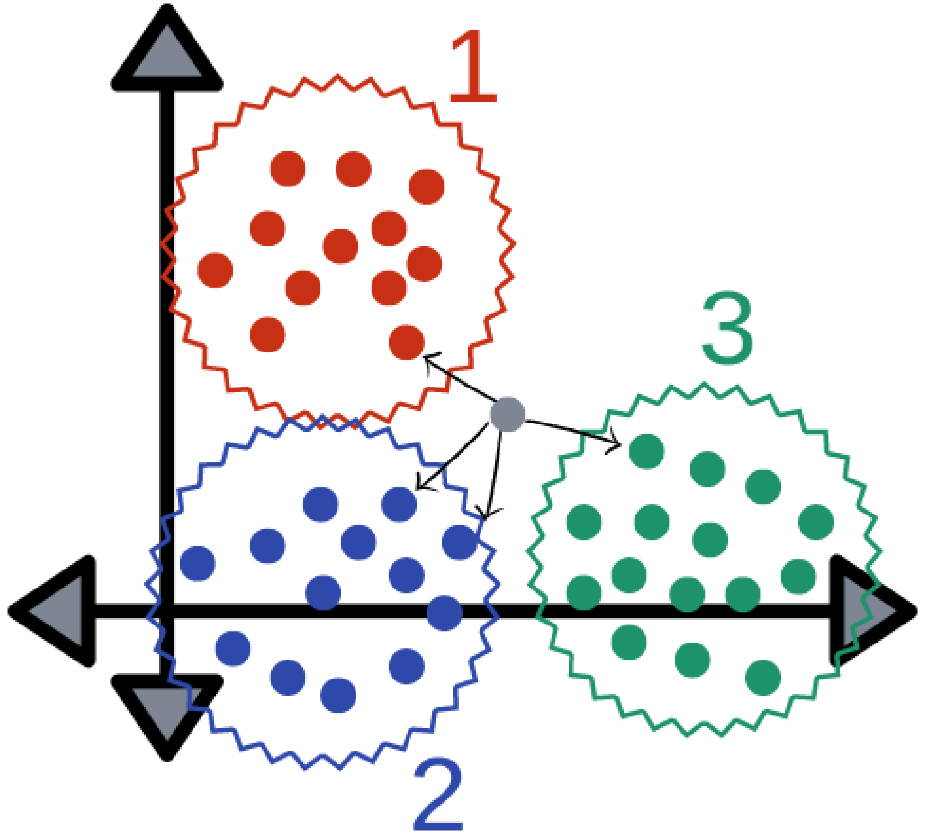
Why Should we Use Feature Scaling?

The first question we need to address – why do we need to scale the variables in our dataset? Some machine learning algorithms are sensitive to feature scaling while others are virtually invariant to it. Let me explain that in more detail.

1. Distance Based Algorithms :

Distance algorithms like **"KNN"**, **"K-means"** and **"SVM"** are most affected by the range of features. This is because behind the scenes they are using distances between data points to determine their similarity. When two features have different scales, there is a chance that higher weightage is given to features with higher magnitude. This will impact the performance of the machine learning algorithm and obviously, we do not want our algorithm to be biassed towards one feature.

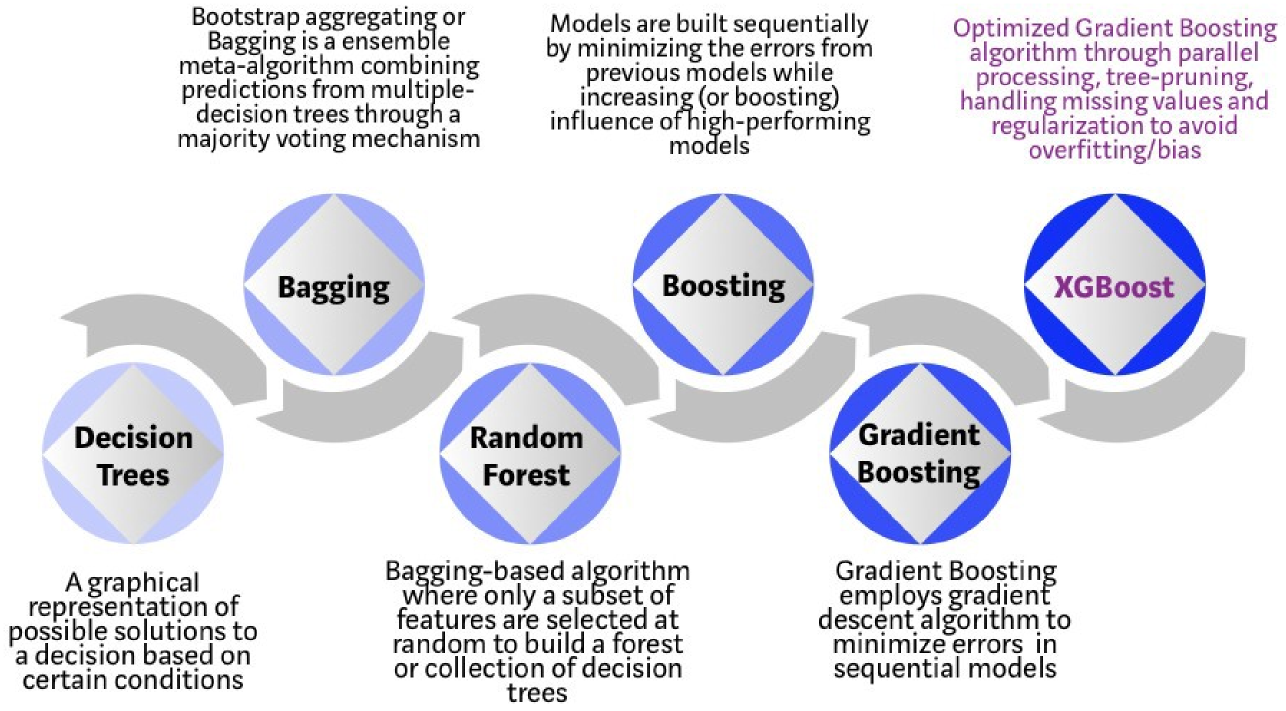
Therefore, we scale our data before employing a distance based algorithm so that all the features contribute equally to the result.



2. Tree-Based Algorithms :

Tree-based algorithms, on the other hand, are fairly insensitive to the scale of the features. Think about it, a decision tree is only splitting a node based on a single feature. The decision tree splits a node on a feature that increases the homogeneity of the node. This split on a feature is not influenced by other features.

So, there is virtually no effect of the remaining features on the split. This is what makes them invariant to the scale of the features!


Bootstrap aggregating or Bagging is a ensemble meta-algorithm combining predictions from multiple-decision trees through a majority voting mechanism
Models are built sequentially by minimizing the errors from previous models while increasing (or boosting) influence of high-performing models
Optimized Gradient Boosting algorithm through parallel processing, tree-pruning, handling missing values and regularization to avoid overfitting/bias
Bagging
Boosting
XGBoost
Decision Trees
Random Forest
Gradient Boosting
A graphical representation of possible solutions to a decision based on certain conditions
Bagging-based algorithm where only a subset of features are selected at random to build a forest or collection of decision trees
Gradient Boosting employs gradient descent algorithm to minimize errors in sequential models

What is Normalization?

Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as Min-Max scaling.

Here's the fromula for normalization :

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

Here, Xmax and Xmin are the maximum and the minimum values of the feature respectively.

When the value of X is the minimum value in the column, the numerator will be 0, and hence X′ is 0 On the other hand, when the value of X is the maximum value in the column, the numerator is equal to the denominator and thus the value of X′ is 1 If the value of X is between the minimum and the maximum value, then the value of X′ is between 0 and 1

What is Standardization?

Standardization is another scaling technique where the values are centered around the mean with a unit standard deviation. This means that the mean of the attribute becomes zero and the resultant distribution has a unit standard deviation

Here's the formula for Standarization:

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

The Big Question – Normalize or Standardize?

Normalization vs. standardization is an eternal question among machine learning newcomers. Let me elaborate on the answer in this section.

- Normalization is good to use when you know that the distribution of your data does not follow a Gaussian distribution(Gaussian distribution is a type of graph that is used to show how data is spread out. It looks like a bell-shaped curve and is used to predict the probability of something happening. For example, if you know the average height of a group of people, you can look at the Gaussian distribution to predict how likely it is that someone in that group is tall or short.). This can be useful in algorithms that do not assume any distribution of the data like K-Nearest Neighbors and Neural Networks.
- Standardization, on the other hand, can be helpful in cases where the data follows a Gaussian distribution. However, this does not have to be necessarily true. Also, unlike normalization, standardization does not have a bounding range. So, even if you have outliers in your data, they will not be affected by standardization.

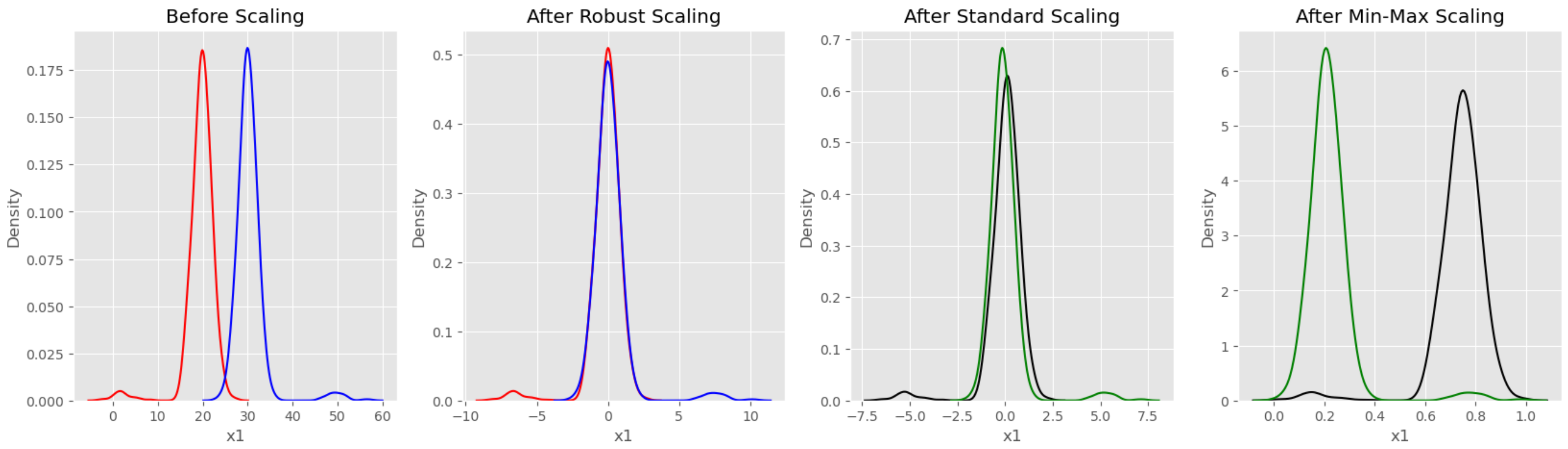
However, at the end of the day, the choice of using normalization or standardization will depend on your problem and the machine learning algorithm you are using. There is no hard and fast rule to tell you when to normalize or standardize your data.

Robust Scaler

When working with outliers we can use Robust Scaling for scaling our data, It scales features using statistics that are robust to outliers. This method removes the median and scales the data in the range between 1st quartile and 3rd quartile. i.e., in between 25th quantile and 75th quantile range. This range is also called an Interquartile range. The median and the interquartile range are then stored so that it could be used upon future data using the transform method. If outliers are present in the dataset, then the median and the interquartile range provide better results and outperform the sample mean and variance. RobustScaler uses the interquartile range so that it is robust to outliers

In [23]:
```python
# data
x = pd.DataFrame({
    # Distribution with lower outliers
    'x1': np.concatenate([np.random.normal(20, 2, 1000), np.random.normal(1, 2, 25)]),
    # Distribution with higher outliers
    'x2': np.concatenate([np.random.normal(30, 2, 1000), np.random.normal(50, 2, 25)]),
})
np.random.normal

scaler = preprocessing.RobustScaler()
robust_df = scaler.fit_transform(x)
robust_df = pd.DataFrame(robust_df, columns =['x1', 'x2'])

scaler = preprocessing.StandardScaler()
standard_df = scaler.fit_transform(x)
standard_df = pd.DataFrame(standard_df, columns =['x1', 'x2'])

scaler = preprocessing.MinMaxScaler()
minmax_df = scaler.fit_transform(x)
minmax_df = pd.DataFrame(minmax_df, columns =['x1', 'x2'])
```

```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(ncols = 4, figsize =(20, 5))
ax1.set_title('Before Scaling')

sns.kdeplot(x['x1'], ax = ax1, color ='r')
sns.kdeplot(x['x2'], ax = ax1, color ='b')
ax2.set_title('After Robust Scaling')

sns.kdeplot(robust_df['x1'], ax = ax2, color ='red')
sns.kdeplot(robust_df['x2'], ax = ax2, color ='blue')
ax3.set_title('After Standard Scaling')

sns.kdeplot(standard_df['x1'], ax = ax3, color ='black')
sns.kdeplot(standard_df['x2'], ax = ax3, color ='g')
ax4.set_title('After Min-Max Scaling')

sns.kdeplot(minmax_df['x1'], ax = ax4, color ='black')
sns.kdeplot(minmax_df['x2'], ax = ax4, color ='g')
plt.show()
```



3. Handling Categorical Variables

Categorical variables/features are any feature type can be classified into two major types:

- Nominal
- Ordinal

Nominal variables are variables that have two or more categories which do not have any kind of order associated with them. For example, if gender is classified into two groups, i.e. male and female, it can be considered as a nominal variable.Ordinal variables, on the other hand, have “levels” or categories with a particular order associated with them. For example, an ordinal categorical variable can be a feature with three different levels: low, medium and high. Order is important.

It is a binary classification problem: the target here is **not skewed** but we use the best metric for this binary classification problem which would be Area Under the ROC Curve (AUC). We can use precision and recall too, but AUC combines these two metrics. Thus, we will be using AUC to evaluate the model that we build on this dataset.

We have to know that computers do not understand text data and thus, we need to convert these categories to numbers. A simple way of doing that can be to use :

- Label Encoding

  ```
  from sklearn.preprocessing import LabelEncoder
  ```

- One Hot Encoding

  ```
  pd.get_dummies()
  ```

but we need to understand where to use which type of label encoding:

For not Tree based Machine Learning Algorithms the best way to go will be to use One-Hot Encoding

- One-Hot-Encoding has the advantage that the result is binary rather than ordinal and that everything sits in an orthogonal vector space.
- The disadvantage is that for high cardinality, the feature space can really blow up quickly and you start fighting with the curse of dimensionality. In these cases, I typically employ one-hot-encoding followed by PCA for dimensionality reduction. I find that the judicious combination of one-hot plus PCA can rarely be beat by other encoding schemes. PCA finds the linear overlap, so will naturally tend to group similar features into the same feature

For Tree based Machine Learning Algorithms the best way to go is with Label Encoding

- LabelEncoder can turn [dog,cat,dog,mouse,cat] into [1,2,1,3,2], but then the imposed ordinality means that the average of dog and mouse is cat. Still there are algorithms like decision trees and random forests that can work with categorical variables just fine and LabelEncoder can be used to store values using less disk space.

In [24]:
```
df[string_col].head()
for col in string_col:
    print(f"The distribution of categorical valeus in the {col} is : ")
    print(df[col].value_counts())
```

```
The distribution of categorical valeus in the Sex is :
M    207
F     96
Name: Sex, dtype: Int64
The distribution of categorical valeus in the ChestPainType is :
TA     143
NAP     87
ATA     50
ASY     23
Name: ChestPainType, dtype: Int64
The distribution of categorical valeus in the RestingECG is :
ST        152
Normal    147
LVH         4
Name: RestingECG, dtype: Int64
The distribution of categorical valeus in the ExerciseAngina is :
N    204
Y     99
Name: ExerciseAngina, dtype: Int64
The distribution of categorical valeus in the ST_Slope is :
Up      142
Flat    140
Down     21
Name: ST_Slope, dtype: Int64
```

In [25]:
```
# As we will be using both types of approches for demonstration lets do First Label Ecoding
# which will be used with Tree Based Algorthms
df_tree = df.apply(LabelEncoder().fit_transform)
df_tree.head()
```

Out[25]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
0	28	0	3	28	97	0	1	59	0	33	0	0
1	28	0	3	39	6	0	1	44	0	39	0	0
2	22	0	3	48	112	1	1	33	1	35	0	0
3	19	1	3	28	34	1	1	54	1	29	0	0
4	25	1	3	42	137	0	1	39	1	31	0	0

We can use this directly in many tree-based models:

- Decision trees
- Random forest
- Extra Trees
- Or any kind of boosted trees model
    - XGBoost
    - GBM
    - LightGBM

This type of encoding cannot be used in linear models, support vector machines or neural networks as they expect data to be normalized (or standardized). For these types of models, we can binarize the data. As shown bellow :

In [26]:
```python
## Creaeting one hot encoded features for working with non tree based algorithms
df_nontree=pd.get_dummies(df,columns=string_col,drop_first=False)
df_nontree.head()
```

Out[26]:

	Age	RestingBP	Cholesterol	FastingBS	MaxHR	Oldpeak	HeartDisease	Sex_F	Sex_M	ChestPainType_ASY	...	ChestPainType_NAP	ChestPainType_TA	RestingECG_LVH	RestingECG_Normal	RestingECG_ST	ExerciseAngina_N	Exercis
0	62	140	268	0	160	3.6	0	1	0	0	...	0	1	0	1	0	1	
1	62	160	164	0	145	6.2	0	1	0	0	...	0	1	0	1	0	1	
2	56	200	288	1	133	4.0	0	1	0	0	...	0	1	0	1	0	0	
3	53	140	203	1	155	3.1	0	0	1	0	...	0	1	0	1	0	0	
4	59	170	326	0	140	3.4	0	0	1	0	...	0	1	0	1	0	0	

5 rows × 21 columns

In [27]:
```python
# Getting the target column at the end
target="HeartDisease"
y=df_nontree[target].values
df_nontree.drop("HeartDisease",axis=1,inplace=True)
df_nontree=pd.concat([df_nontree,df[target]],axis=1)
df_nontree.head()
```

Out[27]:

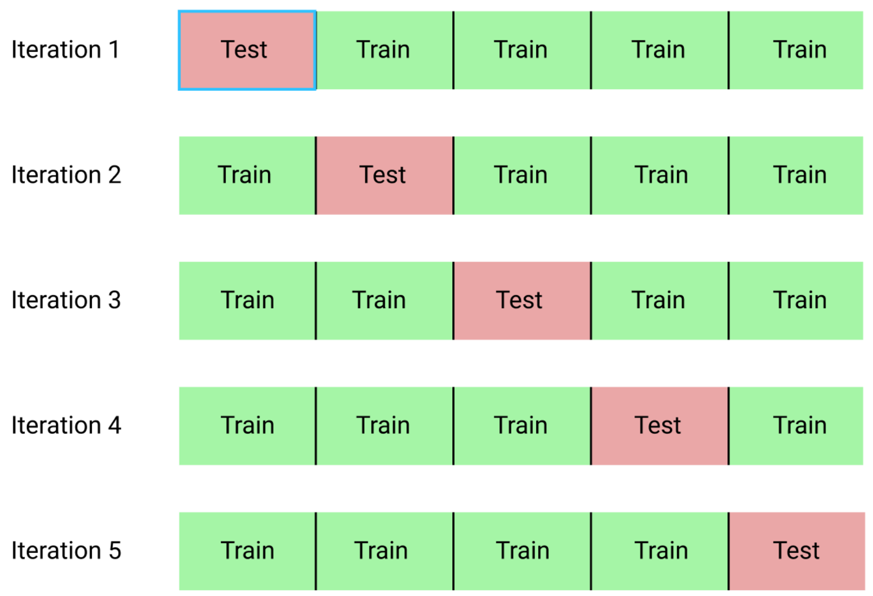
	Age	RestingBP	Cholesterol	FastingBS	MaxHR	Oldpeak	Sex_F	Sex_M	ChestPainType_ASY	ChestPainType_ATA	...	ChestPainType_TA	RestingECG_LVH	RestingECG_Normal	RestingECG_ST	ExerciseAngina_N	ExerciseAngina_Y	ST_
0	62	140	268	0	160	3.6	1	0	0	0	...	1	0	1	0	1	0	
1	62	160	164	0	145	6.2	1	0	0	0	...	1	0	1	0	1	0	
2	56	200	288	1	133	4.0	1	0	0	0	...	1	0	1	0	0	1	
3	53	140	203	1	155	3.1	0	1	0	0	...	1	0	1	0	0	1	
4	59	170	326	0	140	3.4	0	1	0	0	...	1	0	1	0	0	1	

5 rows × 21 columns

Chossing the right Cross-Validation

Choosing the right cross-validation depends on the dataset you are dealing with, and one's choice of cross-validation on one dataset may or may not apply to other datasets. However, there are a few types of cross-validation techniques which are the most popular and widely used. These include:

- k-fold cross-validation
- stratified k-fold cross-validation Cross-validation is dividing training data into a few parts. We train the model on some of these parts and test on the remaining parts



1. K-fold cross-validation :

As you can see, we divide the samples and the targets associated with them. We can divide the data into k different sets which are exclusive of each other. This is known as k-fold cross-validation, We can split any data into k-equal parts using KFold from scikit-learn. Each sample is assigned a value from 0 to k-1 when using k-fold cross validation.

2. Stratified k-fold cross-validation :

If you have a skewed dataset for binary classification with 90% positive samples and only 10% negative samples, you don't want to use random k-fold cross-validation. Using simple k-fold cross-validation for a dataset like this can result in folds with all negative samples. In these cases, we prefer using stratified k-fold cross-validation. Stratified k-fold cross-validation keeps the ratio of labels in each fold constant. So, in each fold, you will have the same 90% positive and 10% negative samples. Thus, whatever metric you choose to evaluate, it will give similar results across all folds.

Training our Machine Learning Model :

NON-TREE BASED ALGORITHMS

So as have talked earlier we have to use different ways to works with categorical data, so we will be using different methods:

1.Using Logistic Regression :

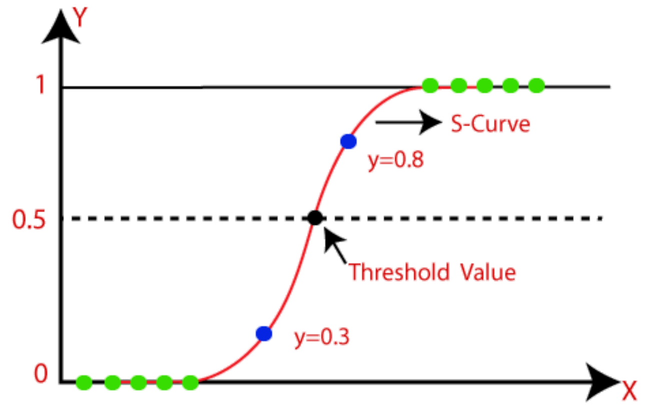
Logistic regression is a calculation used to predict a binary outcome: either something happens, or does not. This can be exhibited as Yes/No, Pass/Fail, Alive/Dead, etc.

Independent variables are analyzed to determine the binary outcome with the results falling into one of two categories. The independent variables can be categorical or numeric, but the dependent variable is always categorical. Written like this:

P(Y=1|X) or P(Y=0|X)

It calculates the probability of dependent variable Y, given independent variable X.

This can be used to calculate the probability of a word having a positive or negative connotation (0, 1, or on a scale between). Or it can be used to determine the object contained in a photo (tree, flower, grass, etc.), with each object given a probability between 0 and 1.



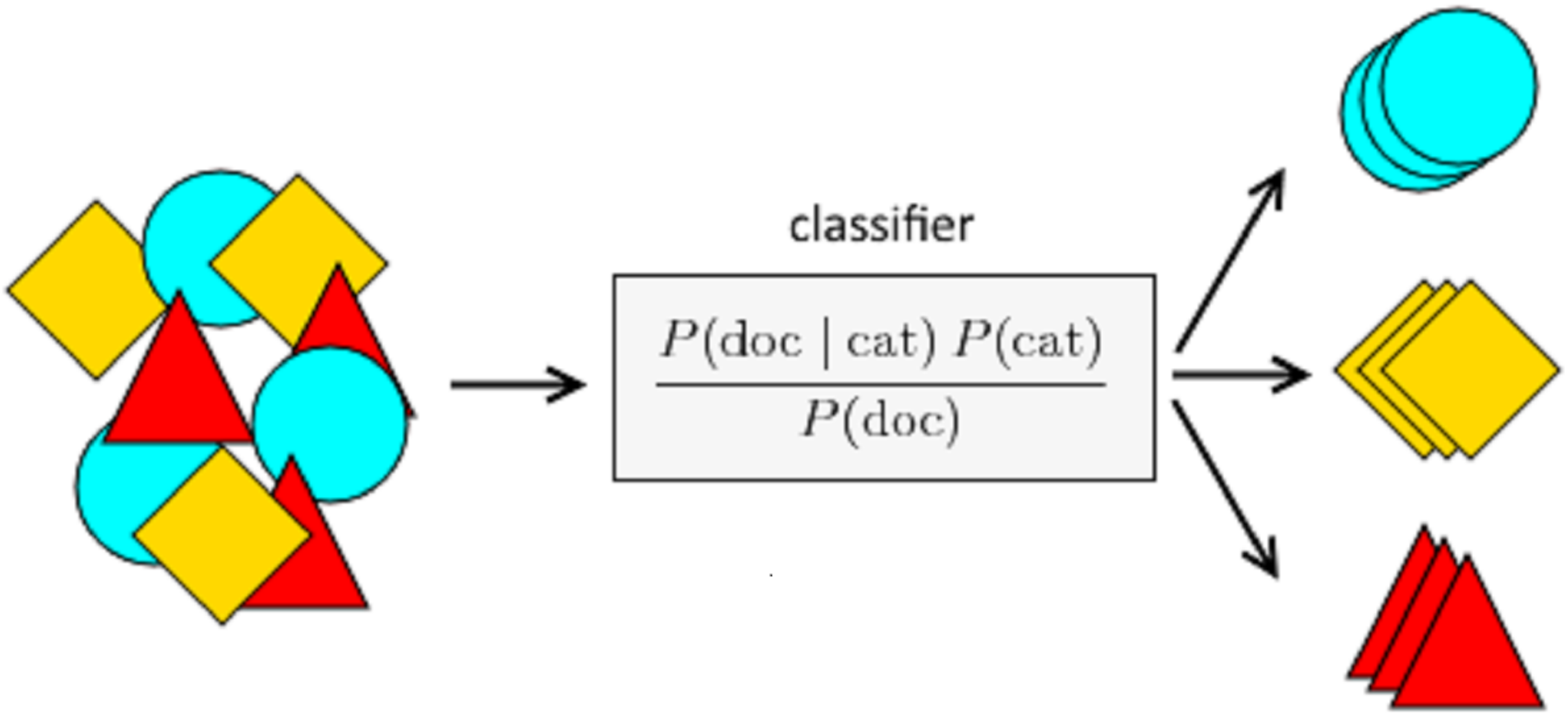
```
In [28]: feature_col_nontree=df_nontree.columns.to_list()
         feature_col_nontree.remove(target)
```

```
In [29]: from sklearn import model_selection
         from sklearn.linear_model import LogisticRegression
         from sklearn.metrics import confusion_matrix,classification_report,accuracy_score,roc_auc_score
         from sklearn.preprocessing import RobustScaler,MinMaxScaler,StandardScaler
         acc_log=[]
```

```python
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_nontree,y=y)):

    X_train=df_nontree.loc[trn_,feature_col_nontree]
    y_train=df_nontree.loc[trn_,target]

    X_valid=df_nontree.loc[val_,feature_col_nontree]
    y_valid=df_nontree.loc[val_,target]

    #print(pd.DataFrame(X_valid).head())
    ro_scaler=MinMaxScaler()
    X_train=ro_scaler.fit_transform(X_train)
    X_valid=ro_scaler.transform(X_valid)


    clf=LogisticRegression()
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_log.append(acc)
    print(f"The accuracy for Fold {fold+1} : {acc}")
    pass
```

```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.51      1.00      0.67        28
           1       1.00      0.18      0.31        33

    accuracy                           0.56        61
   macro avg       0.75      0.59      0.49        61
weighted avg       0.77      0.56      0.48        61

The accuracy for Fold 1 : 0.5909090909090909
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.82      1.00      0.90        28
           1       1.00      0.82      0.90        33

    accuracy                           0.90        61
   macro avg       0.91      0.91      0.90        61
weighted avg       0.92      0.90      0.90        61

The accuracy for Fold 2 : 0.9090909090909092
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.73      0.57      0.64        28
           1       0.69      0.82      0.75        33

    accuracy                           0.70        61
   macro avg       0.71      0.69      0.70        61
weighted avg       0.71      0.70      0.70        61

The accuracy for Fold 3 : 0.6948051948051949
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.75      0.44      0.56        27
           1       0.66      0.88      0.75        33

    accuracy                           0.68        60
   macro avg       0.70      0.66      0.66        60
weighted avg       0.70      0.68      0.67        60

The accuracy for Fold 4 : 0.6616161616161615
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.43      0.11      0.18        27
           1       0.55      0.88      0.67        33

    accuracy                           0.53        60
   macro avg       0.49      0.49      0.43        60
weighted avg       0.49      0.53      0.45        60

The accuracy for Fold 5 : 0.49494949494949503
```

2.Using Naive Bayers



$$\frac{P(\mathrm{doc} \mid \mathrm{cat})\, P(\mathrm{cat})}{P(\mathrm{doc})}$$

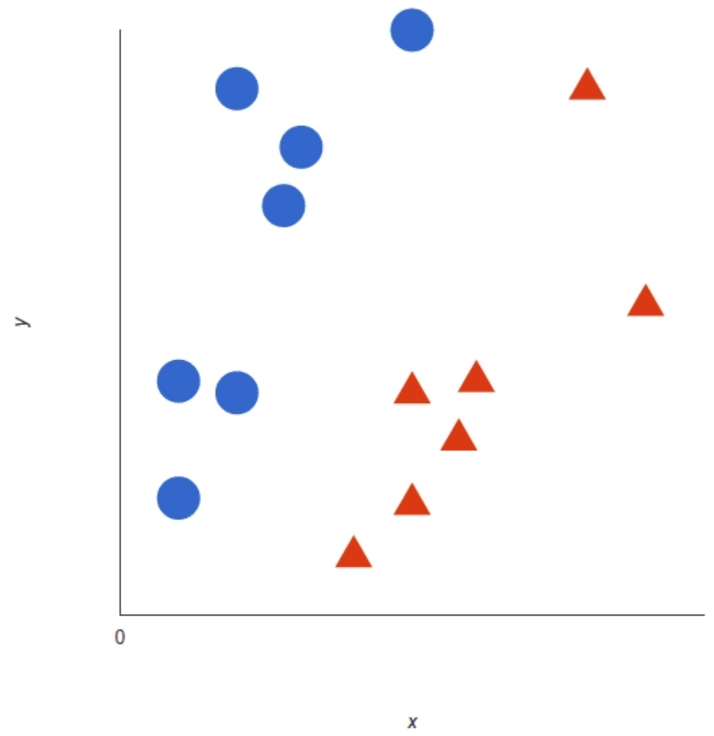
In [30]:
```
from sklearn.naive_bayes import GaussianNB
acc_Gauss=[]
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_nontree,y=y)):

    X_train=df_nontree.loc[trn_,feature_col_nontree]
    y_train=df_nontree.loc[trn_,target]

    X_valid=df_nontree.loc[val_,feature_col_nontree]
    y_valid=df_nontree.loc[val_,target]

    ro_scaler=MinMaxScaler()
    X_train=ro_scaler.fit_transform(X_train)
    X_valid=ro_scaler.transform(X_valid)

    clf=GaussianNB()
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_Gauss.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")

    pass
```

```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.36      0.64      0.46        28
           1       0.09      0.03      0.05        33

    accuracy                           0.31        61
   macro avg       0.23      0.34      0.25        61
weighted avg       0.21      0.31      0.24        61

The accuracy for 1 : 0.33658008658008653
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.90      0.96      0.93        28
           1       0.97      0.91      0.94        33

    accuracy                           0.93        61
   macro avg       0.93      0.94      0.93        61
weighted avg       0.94      0.93      0.93        61

The accuracy for 2 : 0.9366883116883118
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.77      0.82      0.79        28
           1       0.84      0.79      0.81        33

    accuracy                           0.80        61
   macro avg       0.80      0.80      0.80        61
weighted avg       0.81      0.80      0.80        61

The accuracy for 3 : 0.8046536796536796
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.50      0.04      0.07        27
           1       0.55      0.97      0.70        33

    accuracy                           0.55        60
   macro avg       0.53      0.50      0.39        60
weighted avg       0.53      0.55      0.42        60

The accuracy for 4 : 0.5033670033670035
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.35      0.22      0.27        27
           1       0.51      0.67      0.58        33

    accuracy                           0.47        60
   macro avg       0.43      0.44      0.43        60
weighted avg       0.44      0.47      0.44        60

The accuracy for 5 : 0.4444444444444444
```

3.Using SVM(Support Vector Machines):

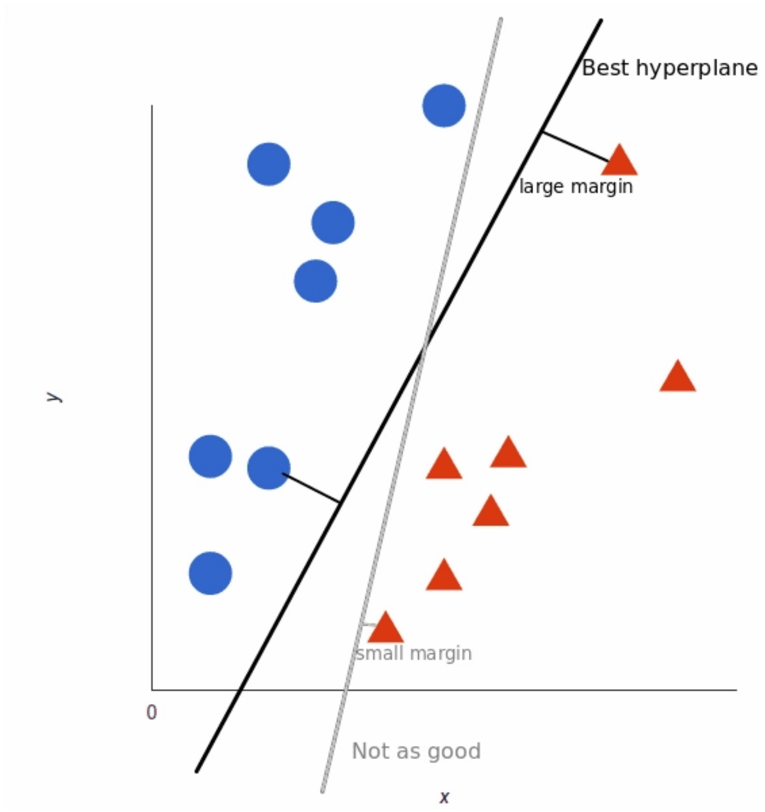
A support vector machine (SVM) uses algorithms to train and classify data within degrees of polarity, taking it to a degree beyond X/Y prediction.

For a simple visual explanation, we'll use two tags: red and blue, with two data features: X and Y, then train our classifier to output an X/Y coordinate as either red or blue.

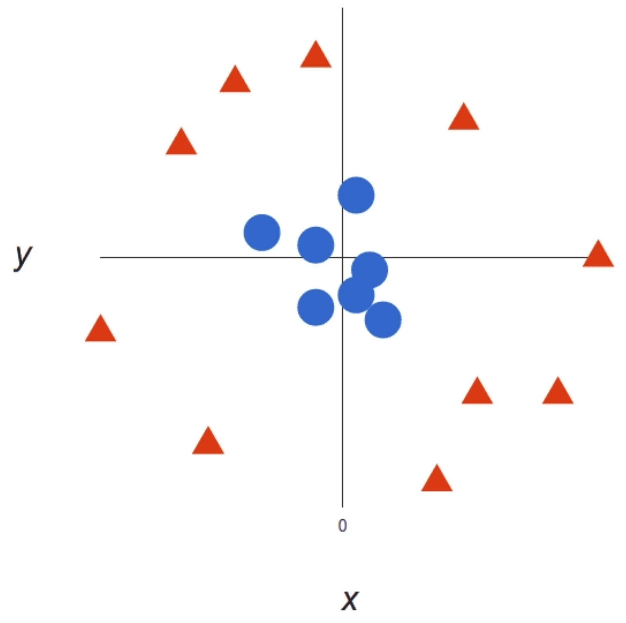


The SVM then assigns a hyperplane that best separates the tags. In two dimensions this is simply a line. Anything on one side of the line is red and anything on the other side is blue. In sentiment analysis, for example, this would be positive and negative.

In order to maximize machine learning, the best hyperplane is the one with the largest distance between each tag:

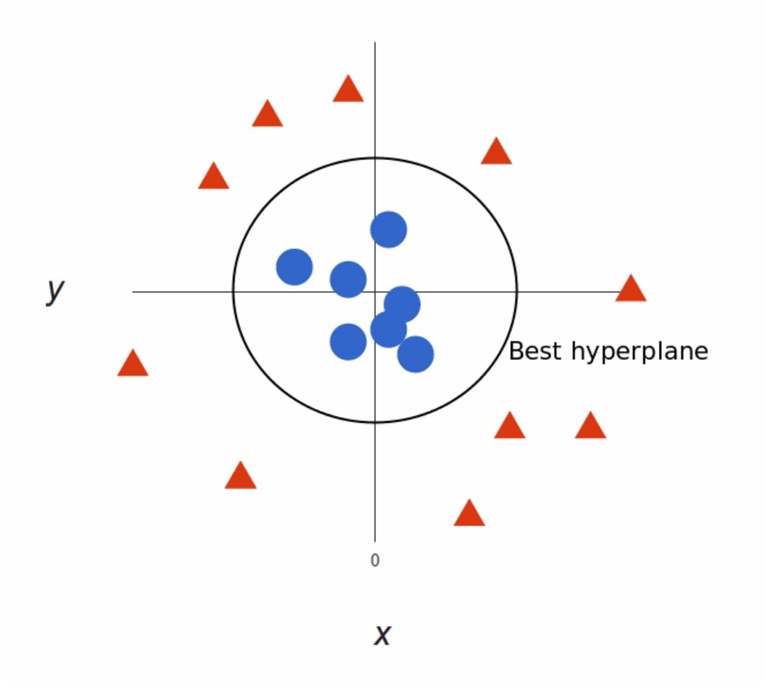


However, as data sets become more complex, it may not be possible to draw a single line to classify the data into two camps:



Using SVM, the more complex the data, the more accurate the predictor will become. Imagine the above in three dimensions, with a Z-axis added, so it becomes a circle.

Mapped back to two dimensions with the best hyperplane, it looks like this



SVM allows for more accurate machine learning because it’s multidimensional.

We need to choose the best Kernel according to our need.

- The linear kernel is mostly preferred for text classification problems as it performs well for large datasets.
- Gaussian kernels tend to give good results when there is no additional information regarding data that is not available.
- Rbf kernel is also a kind of Gaussian kernel which projects the high dimensional data and then searches a linear separation for it.
- Polynomial kernels give good results for problems where all the training data is normalized.

In [31]:
```
# Using Linear Kernel
from sklearn.svm import SVC
acc_svm=[]
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_nontree,y=y)):

    X_train=df_nontree.loc[trn_,feature_col_nontree]
    y_train=df_nontree.loc[trn_,target]

    X_valid=df_nontree.loc[val_,feature_col_nontree]
    y_valid=df_nontree.loc[val_,target]

    ro_scaler=MinMaxScaler()
    X_train=ro_scaler.fit_transform(X_train)
    X_valid=ro_scaler.transform(X_valid)

    clf=SVC(kernel="linear")
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_svm.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")

    pass
```

```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.49      0.89      0.63        28
           1       0.70      0.21      0.33        33

    accuracy                           0.52        61
   macro avg       0.60      0.55      0.48        61
weighted avg       0.60      0.52      0.47        61

The accuracy for 1 : 0.5524891774891775
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.88      1.00      0.93        28
           1       1.00      0.88      0.94        33

    accuracy                           0.93        61
   macro avg       0.94      0.94      0.93        61
weighted avg       0.94      0.93      0.93        61

The accuracy for 2 : 0.9393939393939394
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.74      0.61      0.67        28
           1       0.71      0.82      0.76        33

    accuracy                           0.72        61
   macro avg       0.72      0.71      0.71        61
weighted avg       0.72      0.72      0.72        61

The accuracy for 3 : 0.7126623376623378
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.67      0.30      0.41        27
           1       0.60      0.88      0.72        33

    accuracy                           0.62        60
   macro avg       0.64      0.59      0.56        60
weighted avg       0.63      0.62      0.58        60

The accuracy for 4 : 0.5875420875420876
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.43      0.11      0.18        27
           1       0.55      0.88      0.67        33

    accuracy                           0.53        60
   macro avg       0.49      0.49      0.43        60
weighted avg       0.49      0.53      0.45        60

The accuracy for 5 : 0.49494949494949503
```

In [32]:
```
## Using Sigmoid Kernel
from sklearn.svm import SVC
acc_svm_sig=[]
```

```
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_nontree,y=y)):

    X_train=df_nontree.loc[trn_,feature_col_nontree]
    y_train=df_nontree.loc[trn_,target]

    X_valid=df_nontree.loc[val_,feature_col_nontree]
    y_valid=df_nontree.loc[val_,target]

    ro_scaler=MinMaxScaler()
    X_train=ro_scaler.fit_transform(X_train)
    X_valid=ro_scaler.transform(X_valid)

    clf=SVC(kernel="sigmoid")
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_svm_sig.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")

    pass
```

```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.55      1.00      0.71        28
           1       1.00      0.30      0.47        33

    accuracy                           0.62        61
   macro avg       0.77      0.65      0.59        61
weighted avg       0.79      0.62      0.58        61

The accuracy for 1 : 0.6515151515151515
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.93      1.00      0.97        28
           1       1.00      0.94      0.97        33

    accuracy                           0.97        61
   macro avg       0.97      0.97      0.97        61
weighted avg       0.97      0.97      0.97        61

The accuracy for 2 : 0.9696969696969697
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.92      0.82      0.87        28
           1       0.86      0.94      0.90        33

    accuracy                           0.89        61
   macro avg       0.89      0.88      0.88        61
weighted avg       0.89      0.89      0.88        61

The accuracy for 3 : 0.8804112554112554
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.90      0.67      0.77        27
           1       0.78      0.94      0.85        33

    accuracy                           0.82        60
   macro avg       0.84      0.80      0.81        60
weighted avg       0.83      0.82      0.81        60

The accuracy for 4 : 0.8030303030303031
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.69      0.33      0.45        27
           1       0.62      0.88      0.73        33

    accuracy                           0.63        60
   macro avg       0.65      0.61      0.59        60
weighted avg       0.65      0.63      0.60        60

The accuracy for 5 : 0.6060606060606062
```

In [33]:
```python
## Using RBF kernel
from sklearn.svm import SVC
acc_svm_rbf=[]
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_nontree,y=y)):

    X_train=df_nontree.loc[trn_,feature_col_nontree]
    y_train=df_nontree.loc[trn_,target]
    X_valid=df_nontree.loc[val_,feature_col_nontree]
    y_valid=df_nontree.loc[val_,target]

    ro_scaler=MinMaxScaler()
    X_train=ro_scaler.fit_transform(X_train)
    X_valid=ro_scaler.transform(X_valid)

    clf=SVC(kernel="rbf")
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_svm_rbf.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")

    pass
```

```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.45      0.93      0.60        28
           1       0.33      0.03      0.06        33

    accuracy                           0.44        61
   macro avg       0.39      0.48      0.33        61
weighted avg       0.39      0.44      0.31        61

The accuracy for 1 : 0.47943722943722944
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.59      0.96      0.73        28
           1       0.93      0.42      0.58        33

    accuracy                           0.67        61
   macro avg       0.76      0.69      0.66        61
weighted avg       0.77      0.67      0.65        61

The accuracy for 2 : 0.6942640692640693
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.76      0.57      0.65        28
           1       0.70      0.85      0.77        33

    accuracy                           0.72        61
   macro avg       0.73      0.71      0.71        61
weighted avg       0.73      0.72      0.71        61

The accuracy for 3 : 0.7099567099567099
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        27
           1       0.53      0.91      0.67        33

    accuracy                           0.50        60
   macro avg       0.26      0.45      0.33        60
weighted avg       0.29      0.50      0.37        60

The accuracy for 4 : 0.45454545454545453
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        27
           1       0.54      0.97      0.70        33

    accuracy                           0.53        60
   macro avg       0.27      0.48      0.35        60
weighted avg       0.30      0.53      0.38        60

The accuracy for 5 : 0.48484848484848486
```

In [34]:
```
## Using RBF kernel
from sklearn.svm import SVC
acc_svm_poly=[]
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_nontree,y=y)):

    X_train=df_nontree.loc[trn_,feature_col_nontree]
    y_train=df_nontree.loc[trn_,target]
    X_valid=df_nontree.loc[val_,feature_col_nontree]
    y_valid=df_nontree.loc[val_,target]

    ro_scaler=MinMaxScaler()
    X_train=ro_scaler.fit_transform(X_train)
    X_valid=ro_scaler.transform(X_valid)

    clf=SVC(kernel="poly")
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_svm_poly.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")

    pass
```

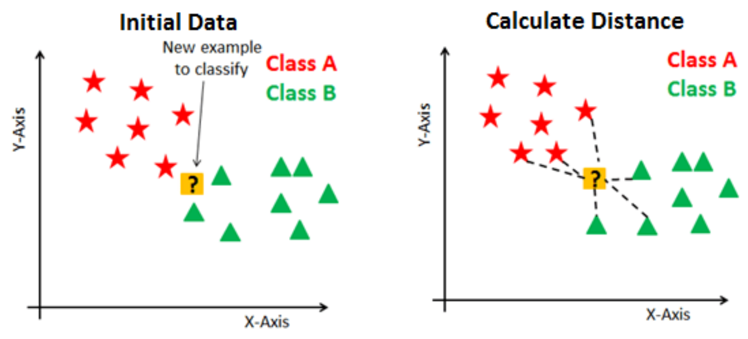
```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.47      0.93      0.63        28
           1       0.67      0.12      0.21        33

    accuracy                           0.49        61
   macro avg       0.57      0.52      0.42        61
weighted avg       0.58      0.49      0.40        61

The accuracy for 1 : 0.5248917748917749
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.49      0.75      0.59        28
           1       0.61      0.33      0.43        33

    accuracy                           0.52        61
   macro avg       0.55      0.54      0.51        61
weighted avg       0.55      0.52      0.50        61

The accuracy for 2 : 0.5416666666666666
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.67      0.43      0.52        28
           1       0.63      0.82      0.71        33

    accuracy                           0.64        61
   macro avg       0.65      0.62      0.62        61
weighted avg       0.65      0.64      0.62        61

The accuracy for 3 : 0.6233766233766235
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.14      0.04      0.06        27
           1       0.51      0.82      0.63        33

    accuracy                           0.47        60
   macro avg       0.33      0.43      0.34        60
weighted avg       0.34      0.47      0.37        60

The accuracy for 4 : 0.42760942760942766
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.25      0.07      0.11        27
           1       0.52      0.82      0.64        33

    accuracy                           0.48        60
   macro avg       0.38      0.45      0.37        60
weighted avg       0.40      0.48      0.40        60

The accuracy for 5 : 0.44612794612794615
```

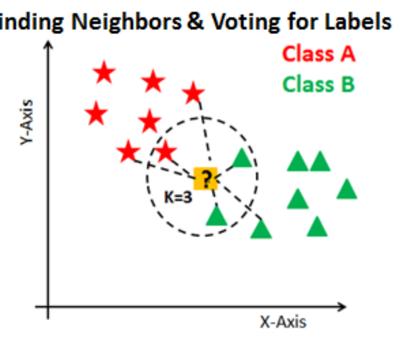
Using K-nearest Neighbors

The optimal K value usually found is the square root of N, where N is the total number of samples

K-nearest neighbors (k-NN) is a pattern recognition algorithm that uses training datasets to find the k closest relatives in future examples.

When k-NN is used in classification, you calculate to place data within the category of its nearest neighbor. If k = 1, then it would be placed in the class nearest 1. K is classified by a plurality poll of its neighbors.





In [35]:
```python
## Using RBF kernel
from sklearn.neighbors import KNeighborsClassifier
acc_KNN=[]
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_nontree,y=y)):

    X_train=df_nontree.loc[trn_,feature_col_nontree]
    y_train=df_nontree.loc[trn_,target]

    X_valid=df_nontree.loc[val_,feature_col_nontree]
    y_valid=df_nontree.loc[val_,target]

    ro_scaler=MinMaxScaler()
    X_train=ro_scaler.fit_transform(X_train)
    X_valid=ro_scaler.transform(X_valid)

    clf=KNeighborsClassifier(n_neighbors=32)
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_KNN.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")

    pass
```

```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.54      0.93      0.68        28
           1       0.85      0.33      0.48        33

    accuracy                           0.61        61
   macro avg       0.69      0.63      0.58        61
weighted avg       0.71      0.61      0.57        61

The accuracy for 1 : 0.6309523809523809
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.87      0.96      0.92        28
           1       0.97      0.88      0.92        33

    accuracy                           0.92        61
   macro avg       0.92      0.92      0.92        61
weighted avg       0.92      0.92      0.92        61

The accuracy for 2 : 0.9215367965367965
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.85      0.79      0.81        28
           1       0.83      0.88      0.85        33

    accuracy                           0.84        61
   macro avg       0.84      0.83      0.83        61
weighted avg       0.84      0.84      0.84        61

The accuracy for 3 : 0.8322510822510822
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.75      0.33      0.46        27
           1       0.62      0.91      0.74        33

    accuracy                           0.65        60
   macro avg       0.69      0.62      0.60        60
weighted avg       0.68      0.65      0.62        60

The accuracy for 4 : 0.6212121212121212
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.53      0.30      0.38        27
           1       0.58      0.79      0.67        33

    accuracy                           0.57        60
   macro avg       0.56      0.54      0.52        60
weighted avg       0.56      0.57      0.54        60

The accuracy for 5 : 0.5420875420875421
```

TREE BASED ALGORITHM

Using Decission tree Classifier

A decision tree is a supervised learning algorithm that is perfect for classification problems, as it’s able to order classes on a precise level. It works like a flow chart, separating data points into two similar categories at a time from the “tree trunk” to “branches,” to “leaves,” where the categories become more finitely similar. This creates categories within categories, allowing for organic classification with limited human supervision.

In [36]:
```
feature_col_tree=df_tree.columns.to_list()
feature_col_tree.remove(target)
```

In [37]:
```
from sklearn.tree import DecisionTreeClassifier
acc_Dtree=[]
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_tree,y=y)):

    X_train=df_tree.loc[trn_,feature_col_tree]
    y_train=df_tree.loc[trn_,target]

    X_valid=df_tree.loc[val_,feature_col_tree]
    y_valid=df_tree.loc[val_,target]

    clf=DecisionTreeClassifier(criterion="entropy")
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_Dtree.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")
```

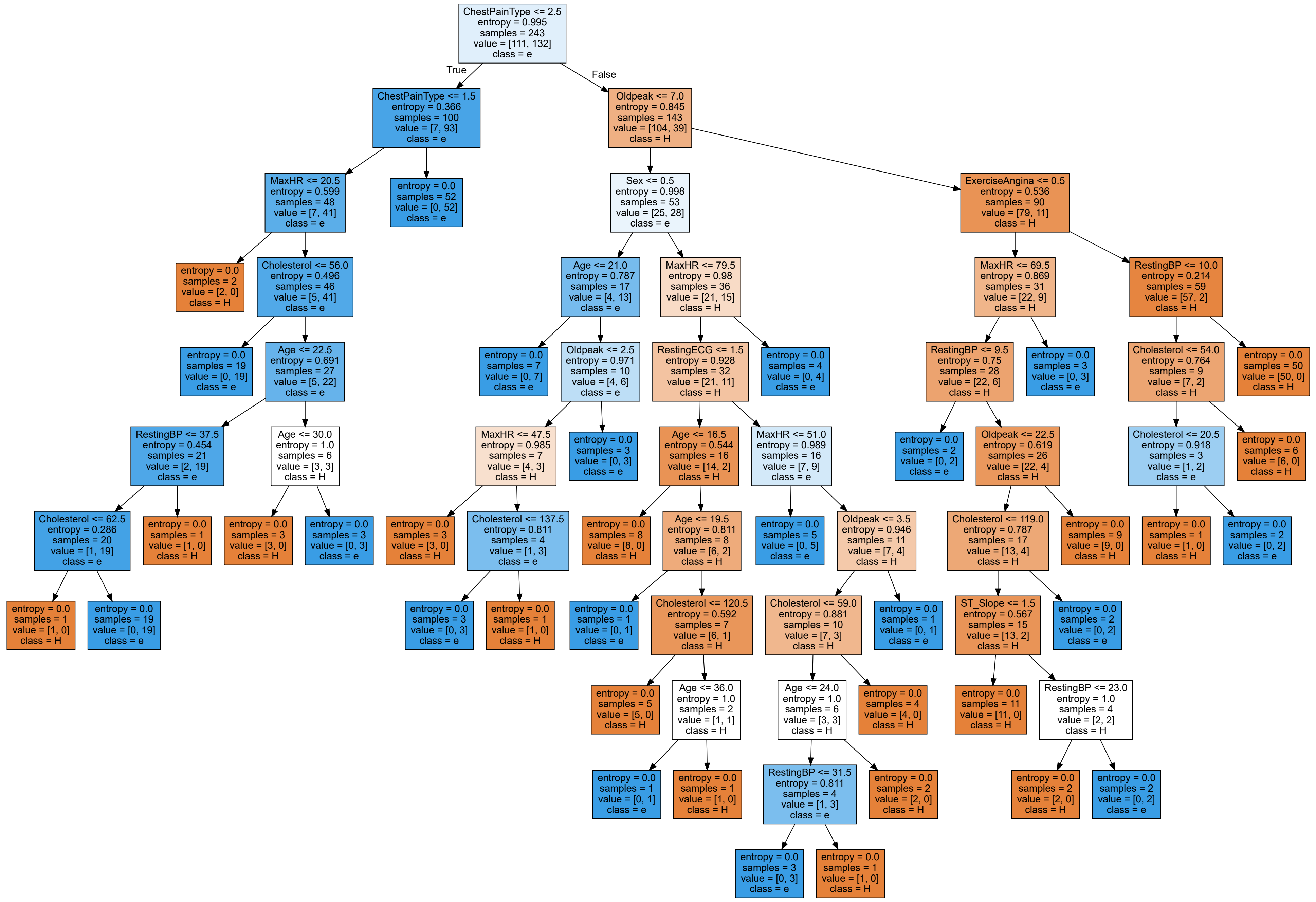
```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.47      1.00      0.64        28
           1       1.00      0.06      0.11        33

    accuracy                           0.49        61
   macro avg       0.74      0.53      0.38        61
weighted avg       0.76      0.49      0.36        61

The accuracy for 1 : 0.5303030303030303
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.63      0.79      0.70        28
           1       0.77      0.61      0.68        33

    accuracy                           0.69        61
   macro avg       0.70      0.70      0.69        61
weighted avg       0.70      0.69      0.69        61

The accuracy for 2 : 0.6958874458874459
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.68      0.68      0.68        28
           1       0.73      0.73      0.73        33

    accuracy                           0.70        61
   macro avg       0.70      0.70      0.70        61
weighted avg       0.70      0.70      0.70        61

The accuracy for 3 : 0.702922077922078
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.35      0.26      0.30        27
           1       0.50      0.61      0.55        33

    accuracy                           0.45        60
   macro avg       0.42      0.43      0.42        60
weighted avg       0.43      0.45      0.44        60

The accuracy for 4 : 0.4326599326599327
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.50      0.22      0.31        27
           1       0.56      0.82      0.67        33

    accuracy                           0.55        60
   macro avg       0.53      0.52      0.49        60
weighted avg       0.53      0.55      0.51        60

The accuracy for 5 : 0.5202020202020202
```

In [38]:
```
!pip install graphviz
import graphviz
```

```
Requirement already satisfied: graphviz in d:\anaconda\lib\site-packages (0.20.1)
```

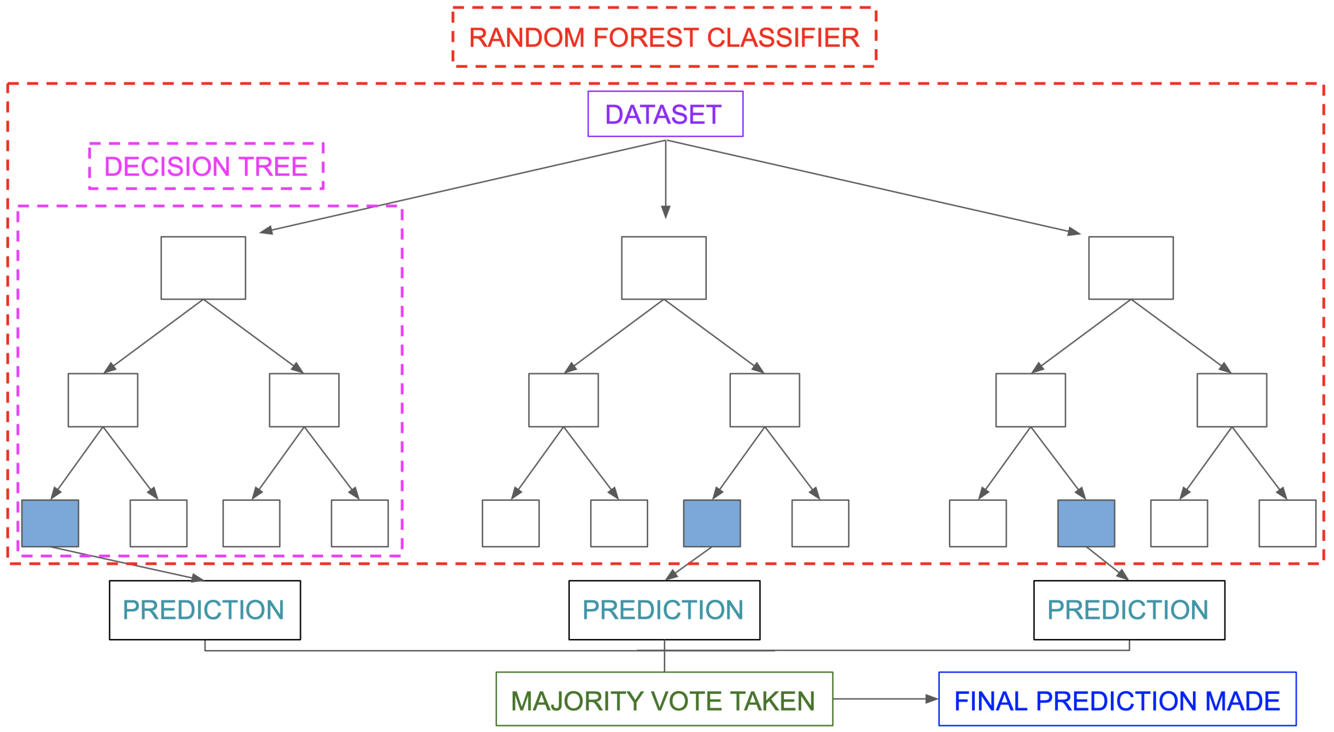
In [39]:
```
from sklearn import tree
# DOT data
dot_data = tree.export_graphviz(clf, out_file=None,
                                feature_names=feature_col_tree,
                                class_names=target,
                                filled=True)

# Draw graph
graph = graphviz.Source(dot_data, format="png")
graph
```

Out[39]:


ChestPainType <= 2.5 entropy = 0.995 samples = 243 value = [111, 132] class = e
True
False
ChestPainType <= 1.5 entropy = 0.366 samples = 100 value = [7, 93] class = e
Oldpeak <= 7.0 entropy = 0.845 samples = 143 value = [104, 39] class = H
MaxHR <= 20.5 entropy = 0.599 samples = 48 value = [7, 41] class = e
entropy = 0.0 samples = 52 value = [0, 52] class = e
Sex <= 0.5 entropy = 0.998 samples = 53 value = [25, 28] class = e
ExerciseAngina <= 0.5 entropy = 0.536 samples = 90 value = [79, 11] class = H
entropy = 0.0 samples = 2 value = [2, 0] class = H
Cholesterol <= 56.0 entropy = 0.496 samples = 46 value = [5, 41] class = e
Age <= 21.0 entropy = 0.787 samples = 17 value = [4, 13] class = e
MaxHR <= 79.5 entropy = 0.98 samples = 36 value = [21, 15] class = H
MaxHR <= 69.5 entropy = 0.869 samples = 31 value = [22, 9] class = H
RestingBP <= 10.0 entropy = 0.214 samples = 59 value = [57, 2] class = H
entropy = 0.0 samples = 19 value = [0, 19] class = e
Age <= 22.5 entropy = 0.691 samples = 27 value = [5, 22] class = e
entropy = 0.0 samples = 7 value = [0, 7] class = e
Oldpeak <= 2.5 entropy = 0.971 samples = 10 value = [4, 6] class = e
RestingECG <= 1.5 entropy = 0.928 samples = 32 value = [21, 11] class = H
entropy = 0.0 samples = 4 value = [0, 4] class = e
RestingBP <= 9.5 entropy = 0.75 samples = 28 value = [22, 6] class = H
entropy = 0.0 samples = 3 value = [0, 3] class = e
Cholesterol <= 54.0 entropy = 0.764 samples = 9 value = [7, 2] class = H
entropy = 0.0 samples = 50 value = [50, 0] class = H
RestingBP <= 37.5 entropy = 0.454 samples = 21 value = [2, 19] class = e
Age <= 30.0 entropy = 1.0 samples = 6 value = [3, 3] class = H
MaxHR <= 47.5 entropy = 0.985 samples = 7 value = [4, 3] class = H
entropy = 0.0 samples = 3 value = [0, 3] class = e
Age <= 16.5 entropy = 0.544 samples = 16 value = [14, 2] class = H
MaxHR <= 51.0 entropy = 0.989 samples = 16 value = [7, 9] class = e
entropy = 0.0 samples = 2 value = [0, 2] class = e
Oldpeak <= 22.5 entropy = 0.619 samples = 26 value = [22, 4] class = H
Cholesterol <= 20.5 entropy = 0.918 samples = 3 value = [1, 2] class = e
entropy = 0.0 samples = 6 value = [6, 0] class = H
Cholesterol <= 62.5 entropy = 0.286 samples = 20 value = [1, 19] class = e
entropy = 0.0 samples = 1 value = [1, 0] class = H
entropy = 0.0 samples = 3 value = [3, 0] class = H
entropy = 0.0 samples = 3 value = [0, 3] class = e
entropy = 0.0 samples = 3 value = [3, 0] class = H
Cholesterol <= 137.5 entropy = 0.811 samples = 4 value = [1, 3] class = e
entropy = 0.0 samples = 8 value = [8, 0] class = H
Age <= 19.5 entropy = 0.811 samples = 8 value = [6, 2] class = H
entropy = 0.0 samples = 5 value = [0, 5] class = e
Oldpeak <= 3.5 entropy = 0.946 samples = 11 value = [7, 4] class = H
Cholesterol <= 119.0 entropy = 0.787 samples = 17 value = [13, 4] class = H
entropy = 0.0 samples = 9 value = [9, 0] class = H
entropy = 0.0 samples = 1 value = [1, 0] class = H
entropy = 0.0 samples = 2 value = [0, 2] class = e
entropy = 0.0 samples = 1 value = [1, 0] class = H
entropy = 0.0 samples = 19 value = [0, 19] class = e
entropy = 0.0 samples = 3 value = [0, 3] class = e
entropy = 0.0 samples = 1 value = [1, 0] class = H
entropy = 0.0 samples = 1 value = [0, 1] class = e
Cholesterol <= 120.5 entropy = 0.592 samples = 7 value = [6, 1] class = H
Cholesterol <= 59.0 entropy = 0.881 samples = 10 value = [7, 3] class = H
entropy = 0.0 samples = 1 value = [0, 1] class = e
ST_Slope <= 1.5 entropy = 0.567 samples = 15 value = [13, 2] class = H
entropy = 0.0 samples = 2 value = [0, 2] class = e
entropy = 0.0 samples = 5 value = [5, 0] class = H
Age <= 36.0 entropy = 1.0 samples = 2 value = [1, 1] class = H
Age <= 24.0 entropy = 1.0 samples = 6 value = [3, 3] class = H
entropy = 0.0 samples = 4 value = [4, 0] class = H
entropy = 0.0 samples = 11 value = [11, 0] class = H
RestingBP <= 23.0 entropy = 1.0 samples = 4 value = [2, 2] class = H
entropy = 0.0 samples = 1 value = [0, 1] class = e
entropy = 0.0 samples = 1 value = [1, 0] class = H
RestingBP <= 31.5 entropy = 0.811 samples = 4 value = [1, 3] class = e
entropy = 0.0 samples = 2 value = [2, 0] class = H
entropy = 0.0 samples = 2 value = [2, 0] class = H
entropy = 0.0 samples = 2 value = [0, 2] class = e
entropy = 0.0 samples = 3 value = [0, 3] class = e
entropy = 0.0 samples = 1 value = [1, 0] class = H

Using Random Forest Classifier

Random forest, like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction (see figure below).



The fundamental concept behind random forest is a simple but powerful one — the wisdom of crowds. In data science speak, the reason that the random forest model works so well is:

> A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models.

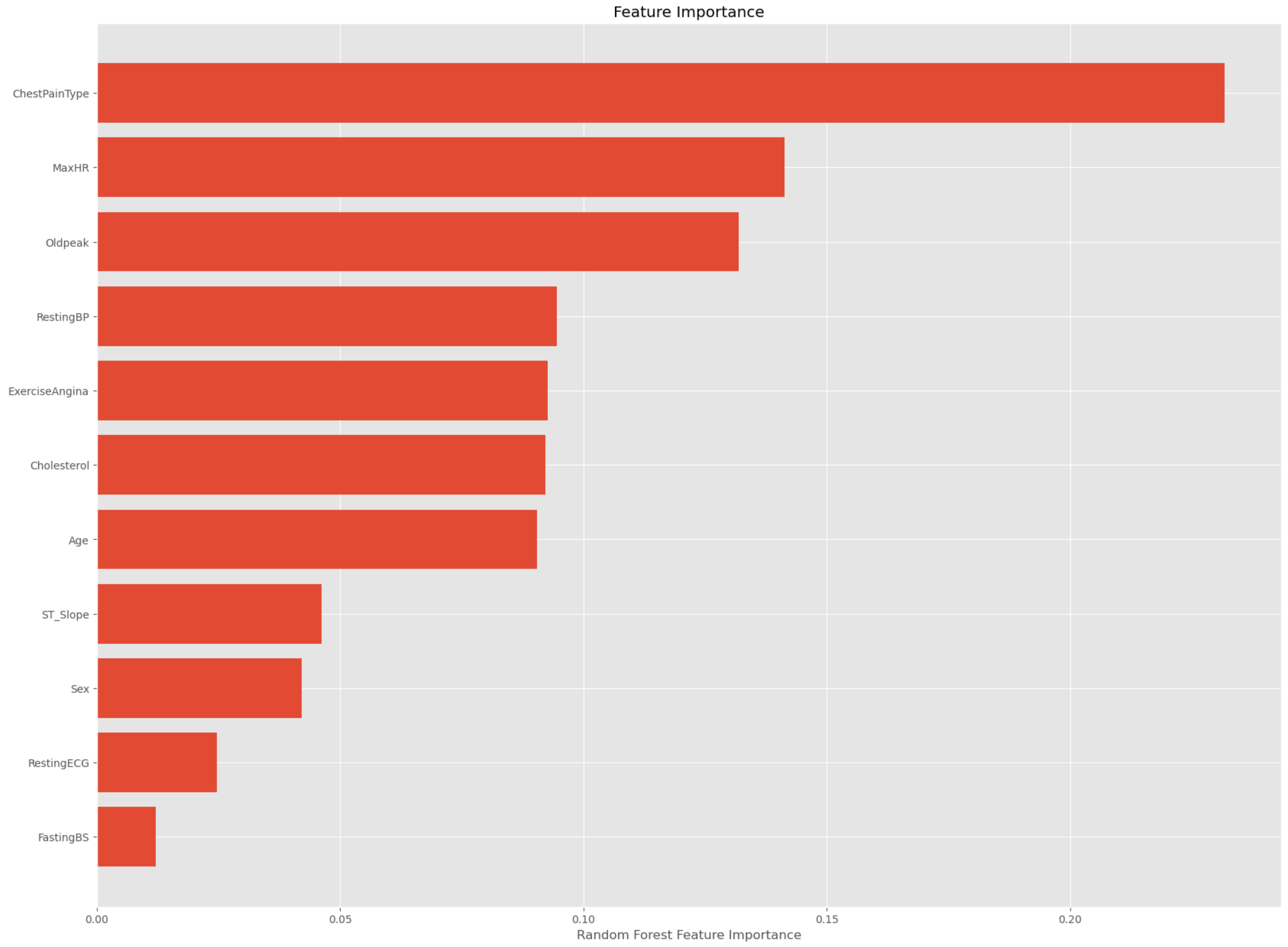
The low correlation between models is the key. Just like how investments with low correlations (like stocks and bonds) come together to form a portfolio that is greater than the sum of its parts, uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. The reason for this wonderful effect is that the trees protect each other from their individual errors (as long as they don't constantly all err in the same direction). While some trees may be wrong, many other trees will be right, so as a group the trees are able to move in the correct direction.

In [40]:
```
from sklearn.ensemble import RandomForestClassifier
acc_RandF=[]
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_tree,y=y)):

    X_train=df_tree.loc[trn_,feature_col_tree]
    y_train=df_tree.loc[trn_,target]

    X_valid=df_tree.loc[val_,feature_col_tree]
    y_valid=df_tree.loc[val_,target]

    clf=RandomForestClassifier(n_estimators=200,criterion="entropy")
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_RandF.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")
```

```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.46      1.00      0.63        28
           1       0.00      0.00      0.00        33

    accuracy                           0.46        61
   macro avg       0.23      0.50      0.31        61
weighted avg       0.21      0.46      0.29        61

The accuracy for 1 : 0.5
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.76      0.89      0.82        28
           1       0.89      0.76      0.82        33

    accuracy                           0.82        61
   macro avg       0.83      0.83      0.82        61
weighted avg       0.83      0.82      0.82        61

The accuracy for 2 : 0.8252164502164503
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.69      0.64      0.67        28
           1       0.71      0.76      0.74        33

    accuracy                           0.70        61
   macro avg       0.70      0.70      0.70        61
weighted avg       0.70      0.70      0.70        61

The accuracy for 3 : 0.7002164502164501
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.56      0.19      0.28        27
           1       0.57      0.88      0.69        33

    accuracy                           0.57        60
   macro avg       0.56      0.53      0.48        60
weighted avg       0.56      0.57      0.50        60

The accuracy for 4 : 0.531986531986532
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.33      0.04      0.07        27
           1       0.54      0.94      0.69        33

    accuracy                           0.53        60
   macro avg       0.44      0.49      0.38        60
weighted avg       0.45      0.53      0.41        60

The accuracy for 5 : 0.48821548821548827
```

In [41]:
```
## Checking Feature importance

plt.figure(figsize=(20,15))
importance = clf.feature_importances_
idxs = np.argsort(importance)
```

```python
plt.title("Feature Importance")
plt.barh(range(len(idxs)),importance[idxs],align="center")
plt.yticks(range(len(idxs)),[feature_col_tree[i] for i in idxs])
plt.xlabel("Random Forest Feature Importance")
#plt.tight_layout()
plt.show()
```



Feature Importance
ChestPainType
MaxHR
Oldpeak
RestingBP
ExerciseAngina
Cholesterol
Age
ST_Slope
Sex
RestingECG
FastingBS
0.00
0.05
0.10
0.15
0.20
Random Forest Feature Importance

Using XGBoost

Unlike many other algorithms, XGBoost is an ensemble learning algorithm meaning that it combines the results of many models, called base learners to make a prediction.

Just like in Random Forests, XGBoost uses Decision Trees as base learners:

However, the trees used by XGBoost are a bit different than traditional decision trees. They are called CART trees (Classification and Regression trees) and instead of containing a single decision in each “leaf” node, they contain real-value scores of whether an instance belongs to a group. After the tree reaches max depth, the decision can be made by converting the scores into categories using a certain threshold.

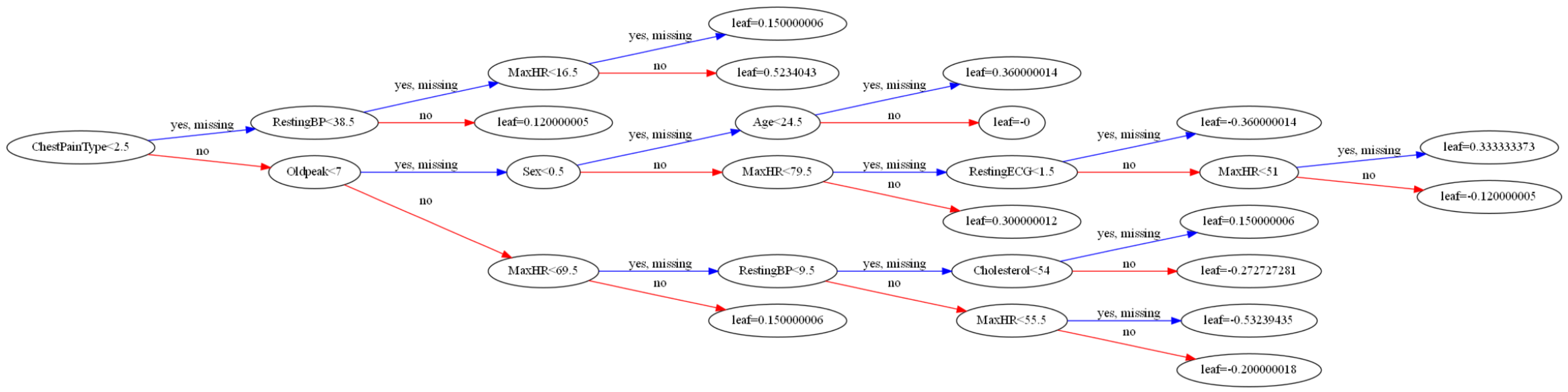
In [42]:

```
from xgboost import XGBClassifier
acc_XGB=[]
kf=model_selection.StratifiedKFold(n_splits=5)
for fold , (trn_,val_) in enumerate(kf.split(X=df_tree,y=y)):

    X_train=df_tree.loc[trn_,feature_col_tree]
    y_train=df_tree.loc[trn_,target]

    X_valid=df_tree.loc[val_,feature_col_tree]
    y_valid=df_tree.loc[val_,target]

    clf=XGBClassifier()
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_valid)
    print(f"The fold is : {fold} : ")
    print(classification_report(y_valid,y_pred))
    acc=roc_auc_score(y_valid,y_pred)
    acc_XGB.append(acc)
    print(f"The accuracy for {fold+1} : {acc}")
```

```
The fold is : 0 :
              precision    recall  f1-score   support

           0       0.47      1.00      0.64        28
           1       1.00      0.06      0.11        33

    accuracy                           0.49        61
   macro avg       0.74      0.53      0.38        61
weighted avg       0.76      0.49      0.36        61

The accuracy for 1 : 0.5303030303030303
The fold is : 1 :
              precision    recall  f1-score   support

           0       0.74      0.82      0.78        28
           1       0.83      0.76      0.79        33

    accuracy                           0.79        61
   macro avg       0.79      0.79      0.79        61
weighted avg       0.79      0.79      0.79        61

The accuracy for 2 : 0.7895021645021645
The fold is : 2 :
              precision    recall  f1-score   support

           0       0.65      0.61      0.63        28
           1       0.69      0.73      0.71        33

    accuracy                           0.67        61
   macro avg       0.67      0.67      0.67        61
weighted avg       0.67      0.67      0.67        61

The accuracy for 3 : 0.6672077922077924
The fold is : 3 :
              precision    recall  f1-score   support

           0       0.80      0.44      0.57        27
           1       0.67      0.91      0.77        33

    accuracy                           0.70        60
   macro avg       0.73      0.68      0.67        60
weighted avg       0.73      0.70      0.68        60

The accuracy for 4 : 0.6767676767676768
The fold is : 4 :
              precision    recall  f1-score   support

           0       0.38      0.11      0.17        27
           1       0.54      0.85      0.66        33

    accuracy                           0.52        60
   macro avg       0.46      0.48      0.42        60
weighted avg       0.46      0.52      0.44        60

The accuracy for 5 : 0.4797979797979799
```

In [43]:

```
fig, ax = plt.subplots(figsize=(30, 30))
from xgboost import plot_tree
plot_tree(clf,num_trees=0,rankdir="LR",ax=ax)
plt.show()
```



Choosing the best Evaluation Matrix:

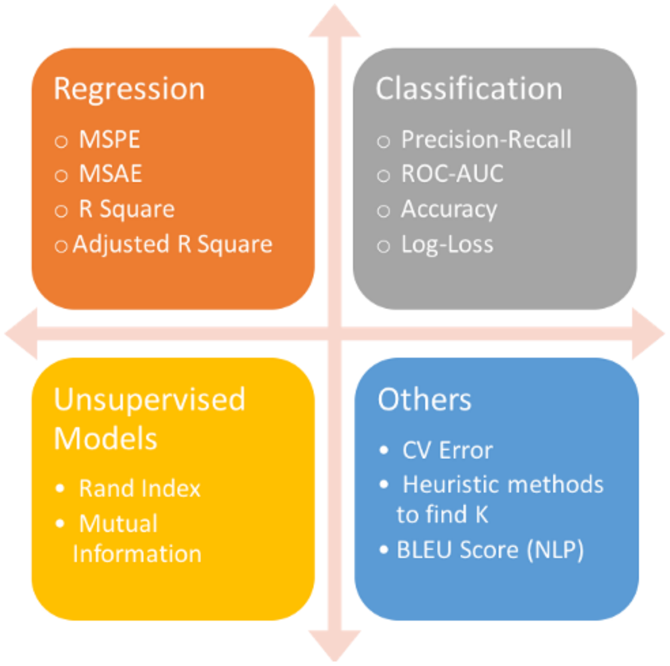
If we talk about classification problems, the most common metrics used are:

- Accuracy
- Precision (P)
- Recall (R)
- F1 score (F1)
- Area under the ROC (Receiver Operating Characteristic) curve or simply AUC (AUC):
    - If we calculate the area under the ROC curve, we are calculating another metric which is used very often when you have a dataset which has skewed binary targets. This metric is known as the Area Under ROC Curve or Area Under Curve or just simply AUC. There are many ways to calculate the area under the ROC curve
    - AUC is a widely used metric for skewed binary classification tasks in the industry,and a metric everyone should know about
- Log loss

> Log Loss = - 1.0 *(target* log(prediction) + (1 - target) * log(1 - prediction))

Most of the metrics that we discussed until now can be converted to a multi-class version. The idea is quite simple. Let's take precision and recall. We can calculate precision and recall for each class in a multi-class classification problem

- **Mcro averaged precision**: calculate precision for all classes individually and then average them
- **Micro averaged precision**: calculate class wise true positive and false positive and then use that to calculate overall precision
- **Weighted precision**: same as macro but in this case, it is weighted average depending on the number of items in each class



Regression
MSPE
MSAE
R Square
Adjusted R Square
Classification
Precision-Recall
ROC-AUC
Accuracy
Log-Loss
Unsupervised Models
Rand Index
Mutual Information
Others
CV Error
Heuristic methods to find K
BLEU Score (NLP)