# - Name: Kendrick Kee

# - UOW ID: 7366814

In [183]:

```
#To import relevant libraries
import pandas as pd
import numpy as np
import random
import sys
import math
```

### Define a function to split dataset

In [184]:

```
# Split the data into training set and testing set
def train_test_split(data, test_size):

    if isinstance(test_size, float):
        test_size = round(test_size * len(data))

    data_index = data.index.tolist()
    test_index = random.sample(population=data_index, k=test_size)

    test_set = data.loc[test_index]
    train_set = data.drop(test_index)

    return train_set, test_set
```

### Define a function to get a summary "dictionary" of dataset as parameter

In [185]:

```python
# Get relevant details about the feature of the dataset for each class label
# If a continuous feature
        ### a tuple of mean, standard deviation and length of dataset is returned
# If data is categorical and is one hot encoded
        ### a tuple of value(1 or 0) and its count with respect to the class (0,1,2)
        ### along with a count of the class is returned
def get_summary(dataset):

    #dictionary to stores details
    #of each label class
    summary = {}

    #loop over unique target values
    for i in dataset.iloc[:,-1].unique():

        #list of details of features for each class (i)
        a = []

        #loop for all the features except label column
        for j in range(len(dataset.columns)-1):

            #store the size of unique values in current feature
            size = len(dataset.iloc[:,j].unique())
            #if size is less than 5, then categorical feature
            if(size < 5):
                lst = list()
                #subset dataset for each class(i)
                #store as df
                df = dataset[dataset.iloc[:,-1]==i]

                #loop for unique values in categorical features
                for k in dataset.iloc[:,j].unique():

                    #for each unique values, store the
                    #value and count of the value in feature
                    lst.append(k)
                    lst.append(len(df[df.iloc[:,j] == k]))

                #make a tuple out of the list
                a.append(tuple([lst[0], lst[1], lst[2], lst[3], len(df)]))

            #else continuous feature
            else:
                a.append((dataset[dataset.iloc[:,-1]==i].mean(axis=0)[j],dataset[dat

        summary[i] = a
    return summary
```

### Define Gaussian Naive Bayes function

In [186]:

```
# Calculate probability for continuous fetaures
# Calculate the Gaussian probability distribution function for x
def calculate_probability_Gaussian(x, mean, stdev,total_rows):
    if stdev == 0 or isnan(stdev):
        return 1/total_rows#if stdev is 0, return the probability of "Add one count'

    exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
    if exponent == 0:
        #number overflow occurs exponent == smallest possibe float by system
        exponent = sys.float_info.min
    return 1 / (sqrt(2 * pi) * stdev) * exponent
```

**To prevent numerical underflow, when exponent underflows out of pythons float precision, exponent will become small possible float by system**

**To prevent the zero frequency/count problem, this function catches occurences of 0/null standard deviation and returns the probablity of 1/number of observations**

### Define Navie Bayes function

In [187]:

```
# Calculate probability for categorical features
def calculate_probability(x, X1, count_1, X2, count_2, class_count,total_rows):
    if x == X1:
        #if zero frequency occurs, add 1 to count and return the probability
        if count_1/class_count == 0:
            return 1/total_rows

        return count_1/class_count
    else:
        #if zero frequency occurs, add 1 to count and return the probability
        if count_2/class_count == 0:
            return 1/total_rows

        return count_2/class_count
```

**Similarly, to prevent 0 frequency error, returning "add one count" probabilty instead of 0**

### Define functions for probability computation

In [188]:

```
lculate the probabilities of predicting each class for a given row
r continuous features use Gaussian probability function
r categorical feature calculate_probability function
calculate_class_probabilities(summaries, row):

#get the length of the dataset
#sum up all the counts of each label class
total_rows = sum([summaries[label][0][2] for label in summaries])

#instantiate a dictionary to store probability
#of each label class for a given row
probabilities = dict()

#get the class value: class_value
#get the summaries for each class: class_summaries
for class_value, class_summaries in summaries.items():

    #get the probability of each label class e.g.
    #if class label 1 has a length of 12345
    #and length of dataset is 234567
    #then this probability is 12345/234567
    probabilities[class_value] = summaries[class_value][0][2]/float(total_rows)

    #loop over all the class_summaries
    #i.e. summaries of each feature
    for i in range(len(class_summaries)):

        #since the summaries for categorical variable contains 5 values,
        #while the ones for continuous variable contains 3 values
        #categorical fetaure
        if len(class_summaries[i]) > 3:
            X1, count_1, X2, count_2, class_count = class_summaries[i]
            probabilities[class_value] = \
            probabilities[class_value] * calculate_probability(row[i],X1, count_1, X
        #continuous feature
        else:
            mean, stdev, _ = class_summaries[i]
            probabilities[class_value] = \
            probabilities[class_value] * calculate_probability_Gaussian(row[i], mean

return probabilities
```

In [189]:

```
# Predict the class for a given row
def predict(summaries, row):
    probabilities = calculate_class_probabilities(summaries, row)

    #instantiate variable to store
    #best label: best_label
    #best probability: best_prob
    best_label, best_prob = None, -1

    #get the best probability
    for class_value, probability in probabilities.items():
        if best_label is None or probability > best_prob:
            best_prob = probability
            best_label = class_value
    return best_label
```

# Driver function for Naive Bayes

In [190]:

```
# Naive Bayes Algorithm
# predict values for the test set
def naive_bayes(train, test):
    summary = get_summary(train)
    predictions = list()
    for row in test.values:
        output = predict(summary, row)
        predictions.append(output)
    return(predictions)
```

## Accuracy function to determine regression metrics

In [191]:

```
# Calculated as:
# check for equality of predicted value and labels in test_set
# calculates the sum of correct prediction
# divides the sum by length of test_set


def accuracy(predictions, data_set):
    y_test = list(data_set.iloc[:,-1])
    correct_count = 0
    sum_error = 0.0
    rsme_error = 0.0
    for i in range(len(y_test)):
        if predictions[i] == y_test[i]:
            correct_count += 1
        sum_error += abs(predictions[i] - y_test[i])
        prediction_error = abs(predictions[i] - y_test[i])
        rsme_error = (prediction_error**2)
    print(f'Number of exact matches in predictions: {correct_count}/{len(y_test)}')
    print(f'MEAN SQUARED ERROR: {np.square(np.subtract(y_test,predictions)).mean()}'
    print(f'ROOT MEAN SQUARED ERROR: {sqrt(rsme_error/float(len(y_test)))}')
    print(f'MEAN ABSOLUTE ERROR: {sum_error/float(len(y_test))}')
    return (round(correct_count/len(data_set)*100,3))
```

**This function computes common regression metrics such as MSE, MAE and RSME to allow for easy evaluation of the model.**

# Loading the Dataset

In [192]:

```
names =  ["Sex", "Length", "Diameter", "Height", "Whole weight",
           "Shucked weight", "Viscera weight","Shell weight", "Rings"]
df = pd.read_csv('abalone.data', header=None, names=names)
df.head()
```

Out[192]:

| | Sex | Length | Diameter | Height | Whole weight | Shucked weight | Viscera weight | Shell weight | Rings |
|---|---|---|---|---|---|---|---|---|---|
| **0** | M | 0.455 | 0.365 | 0.095 | 0.5140 | 0.2245 | 0.1010 | 0.150 | 15 |
| **1** | M | 0.350 | 0.265 | 0.090 | 0.2255 | 0.0995 | 0.0485 | 0.070 | 7 |
| **2** | F | 0.530 | 0.420 | 0.135 | 0.6770 | 0.2565 | 0.1415 | 0.210 | 9 |
| **3** | M | 0.440 | 0.365 | 0.125 | 0.5160 | 0.2155 | 0.1140 | 0.155 | 10 |
| **4** | I | 0.330 | 0.255 | 0.080 | 0.2050 | 0.0895 | 0.0395 | 0.055 | 7 |

Data Preprocessing

In [193]:

```
# Check for missing data and type of data
df.head().info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Sex             5 non-null      object
 1   Length          5 non-null      float64
 2   Diameter        5 non-null      float64
 3   Height          5 non-null      float64
 4   Whole weight    5 non-null      float64
 5   Shucked weight  5 non-null      float64
 6   Viscera weight  5 non-null      float64
 7   Shell weight    5 non-null      float64
 8   Rings           5 non-null      int64
dtypes: float64(7), int64(1), object(1)
memory usage: 488.0+ bytes
```

In [194]:

```
encode = {"Sex": {"M":1,"F":2,"I":3}}
df = df.replace(encode)
df.head()
```

Out[194]:

| | Sex | Length | Diameter | Height | Whole weight | Shucked weight | Viscera weight | Shell weight | Rings |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0.455 | 0.365 | 0.095 | 0.5140 | 0.2245 | 0.1010 | 0.150 | 15 |
| **1** | 1 | 0.350 | 0.265 | 0.090 | 0.2255 | 0.0995 | 0.0485 | 0.070 | 7 |
| **2** | 2 | 0.530 | 0.420 | 0.135 | 0.6770 | 0.2565 | 0.1415 | 0.210 | 9 |
| **3** | 1 | 0.440 | 0.365 | 0.125 | 0.5160 | 0.2155 | 0.1140 | 0.155 | 10 |
| **4** | 3 | 0.330 | 0.255 | 0.080 | 0.2050 | 0.0895 | 0.0395 | 0.055 | 7 |

In [195]:

```python
# Get the statistical info of the numeric features
df.describe()
```

Out[195]:

| | Sex | Length | Diameter | Height | Whole weight | Shucked weight | Visc wei |
|---|---|---|---|---|---|---|---|
| **count** | 4177.000000 | 4177.000000 | 4177.000000 | 4177.000000 | 4177.000000 | 4177.000000 | 4177.0000 |
| **mean** | 1.955470 | 0.523992 | 0.407881 | 0.139516 | 0.828742 | 0.359367 | 0.180 |
| **std** | 0.827815 | 0.120093 | 0.099240 | 0.041827 | 0.490389 | 0.221963 | 0.109 |
| **min** | 1.000000 | 0.075000 | 0.055000 | 0.000000 | 0.002000 | 0.001000 | 0.000 |
| **25%** | 1.000000 | 0.450000 | 0.350000 | 0.115000 | 0.441500 | 0.186000 | 0.093 |
| **50%** | 2.000000 | 0.545000 | 0.425000 | 0.140000 | 0.799500 | 0.336000 | 0.171 |
| **75%** | 3.000000 | 0.615000 | 0.480000 | 0.165000 | 1.153000 | 0.502000 | 0.253 |
| **max** | 3.000000 | 0.815000 | 0.650000 | 1.130000 | 2.825500 | 1.488000 | 0.760 |

In [196]:

```python
df.head()
```

Out[196]:

| | Sex | Length | Diameter | Height | Whole weight | Shucked weight | Viscera weight | Shell weight | Rings |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0.455 | 0.365 | 0.095 | 0.5140 | 0.2245 | 0.1010 | 0.150 | 15 |
| **1** | 1 | 0.350 | 0.265 | 0.090 | 0.2255 | 0.0995 | 0.0485 | 0.070 | 7 |
| **2** | 2 | 0.530 | 0.420 | 0.135 | 0.6770 | 0.2565 | 0.1415 | 0.210 | 9 |
| **3** | 1 | 0.440 | 0.365 | 0.125 | 0.5160 | 0.2155 | 0.1140 | 0.155 | 10 |
| **4** | 3 | 0.330 | 0.255 | 0.080 | 0.2050 | 0.0895 | 0.0395 | 0.055 | 7 |

In [197]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4177 entries, 0 to 4176
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Sex             4177 non-null   int64  
 1   Length          4177 non-null   float64
 2   Diameter        4177 non-null   float64
 3   Height          4177 non-null   float64
 4   Whole weight    4177 non-null   float64
 5   Shucked weight  4177 non-null   float64
 6   Viscera weight  4177 non-null   float64
 7   Shell weight    4177 non-null   float64
 8   Rings           4177 non-null   int64  
dtypes: float64(7), int64(2)
memory usage: 293.8 KB
```

In [198]:

```
# Split the dataset into training and testing
train_set, test_set = train_test_split(df, 0.3)
train_set.describe()
```

Out[198]:

| | Sex | Length | Diameter | Height | Whole weight | Shucked weight | Visc wei |
|---|---|---|---|---|---|---|---|
| **count** | 2924.000000 | 2924.000000 | 2924.000000 | 2924.000000 | 2924.000000 | 2924.000000 | 2924.000 |
| **mean** | 1.962722 | 0.523092 | 0.407030 | 0.139390 | 0.819409 | 0.355485 | 0.178 |
| **std** | 0.830264 | 0.118374 | 0.097669 | 0.042868 | 0.481450 | 0.219325 | 0.108 |
| **min** | 1.000000 | 0.110000 | 0.090000 | 0.000000 | 0.008000 | 0.002500 | 0.000 |
| **25%** | 1.000000 | 0.450000 | 0.350000 | 0.115000 | 0.439750 | 0.183500 | 0.092 |
| **50%** | 2.000000 | 0.540000 | 0.420000 | 0.140000 | 0.787000 | 0.330500 | 0.168 |
| **75%** | 3.000000 | 0.610000 | 0.480000 | 0.165000 | 1.137000 | 0.495000 | 0.249 |
| **max** | 3.000000 | 0.800000 | 0.630000 | 1.130000 | 2.657000 | 1.488000 | 0.590 |

In [199]:

```
test_set.describe()
```

Out[199]:

| | Sex | Length | Diameter | Height | Whole weight | Shucked weight | Visc wei |
|---|---|---|---|---|---|---|---|
| count | 1253.000000 | 1253.000000 | 1253.000000 | 1253.000000 | 1253.000000 | 1253.000000 | 1253.000 |
| mean | 1.938547 | 0.526093 | 0.409868 | 0.139812 | 0.850521 | 0.368428 | 0.185 |
| std | 0.822151 | 0.124035 | 0.102824 | 0.039305 | 0.510173 | 0.227832 | 0.113 |
| min | 1.000000 | 0.075000 | 0.055000 | 0.000000 | 0.002000 | 0.001000 | 0.000 |
| 25% | 1.000000 | 0.455000 | 0.350000 | 0.115000 | 0.449500 | 0.191500 | 0.095 |
| 50% | 2.000000 | 0.550000 | 0.430000 | 0.145000 | 0.820500 | 0.349500 | 0.177 |
| 75% | 3.000000 | 0.620000 | 0.485000 | 0.165000 | 1.177500 | 0.517000 | 0.258 |
| max | 3.000000 | 0.815000 | 0.650000 | 0.250000 | 2.825500 | 1.348500 | 0.760 |

In [200]:

```
# Test the model on training set
train_pred = naive_bayes(train_set, train_set)
print('Accuracy of prediction for training set:', accuracy(train_pred, train_set))
```

```
Number of exact matches in predictions: 485/2924
MEAN SQUARED ERROR: 26.74863201094391
ROOT MEAN SQUARED ERROR: 0.20342484840021002
MEAN ABSOLUTE ERROR: 3.6542407660738716
Accuracy of prediction for training set: 16.587
```

In [201]:

```
# Test the model on testing set
test_pred = naive_bayes(train_set, test_set)
print('Accuracy of prediction for testing set:', accuracy(test_pred, test_set))
```

```
Number of exact matches in predictions: 217/1253
MEAN SQUARED ERROR: 26.62809257781325
ROOT MEAN SQUARED ERROR: 0.0
MEAN ABSOLUTE ERROR: 3.6608140462889067
Accuracy of prediction for testing set: 17.318
```

# Observations

The accuracy of the model is extremely poor at 17% as it is unable to predict the exact number of rings. This is expected as predicting the number of rings poses as more of a regression problem than a classification one due to the continous nature of "Rings" in this context

Therefore, we will observe its other metrics commonly used in regression to evaluate the model.

The MSE of both models are very close for both the training and testing dataset, this suggests that no overfitting occured. However, the MAE for both models is 3.6, this means that the model on average predicts the number of rings to be +- 3.6 which depending on the context and importance of accuracy of getting the exact ring count may deem this model to be good or bad.

The RMSE for the testing dataset appears to be 0 whist its training counterpart has a 0.2 RMSE. This could mean that the model used for the testing dataset is alot more accurate than its training counterpart based on that metric.