CSE315 Assignment 1

Data Preprocessing and Decision Tree (ID3)

Name: Wilbert Osmond
Student ID: 1926308

Major: Exchange (non-UoL)
University: Xi’an Jiaotong-Liverpool University (XJTLU)
Region: Suzhou, Jiangsu, China Mainland

Period: Fall 2019
Date: 28/11/2019 (mitigating circumstances)

Contents

Overview

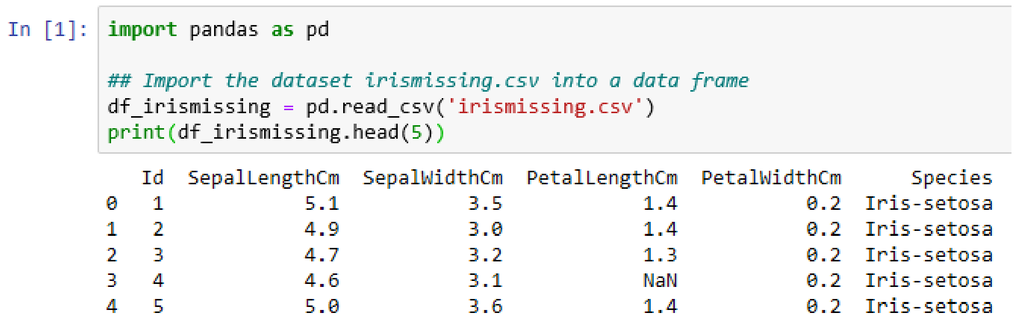
Throughout the following project, data pre-processing and decision tree especially ID3 were explored and implemented in Python code. This project unites our lecture work with that of our laboratory tutorials, showcasing our skills in data-preprocessing, ID3 decision tree and algorithm. Through some manual calculations and Python coding, this project allows us for a full understanding of the step-by-step of each concept. This project displays the importance of the relevant concepts, proper Python coding and improves critical analysis skills of the students completing it.

Note that the Python codes are written in iPython Notebook. It is also important to note that packages that have been imported previously may not be shown to be imported again in the next figures of code.

Task 1: Data Pre-processing

1.1. Import the dataset *irismissing.cs* vinto a data frame and find the row number of each instance that has missing values. [10marks]

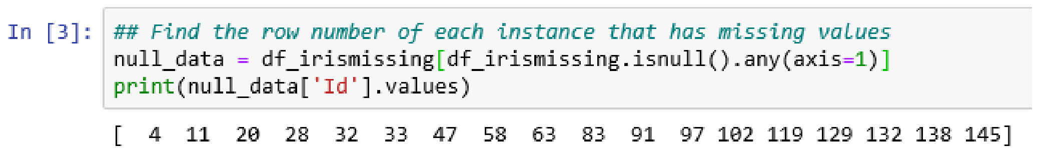
First, I imported the dataset from irismissing.csv to a data frame in Python using pd.read_csv from the pandas library as shown in Figure 1.1.1.



```
In [1]: import pandas as pd

        ## Import the dataset irismissing.csv into a data frame
        df_irismissing = pd.read_csv('irismissing.csv')
        print(df_irismissing.head(5))

           Id  SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm      Species
        0   1            5.1           3.5            1.4           0.2  Iris-setosa
        1   2            4.9           3.0            1.4           0.2  Iris-setosa
        2   3            4.7           3.2            1.3           0.2  Iris-setosa
        3   4            4.6           3.1            NaN           0.2  Iris-setosa
        4   5            5.0           3.6            1.4           0.2  Iris-setosa
```

Figure 1.1.1: Code for importing irismissing.csv into a data frame.

After the data frame is successfully imported, I created a new data frame "null_data" after assigning a True value for any missing values in the data frame by using *.isnull()* syntax and getting any rows which contains at least one True value by using *.any(axis=1)* syntax. Subsequently, I printed the values of the id's (i.e. the row number) of "null_data" which contains only rows with missing values (See Figure 1.1.2)

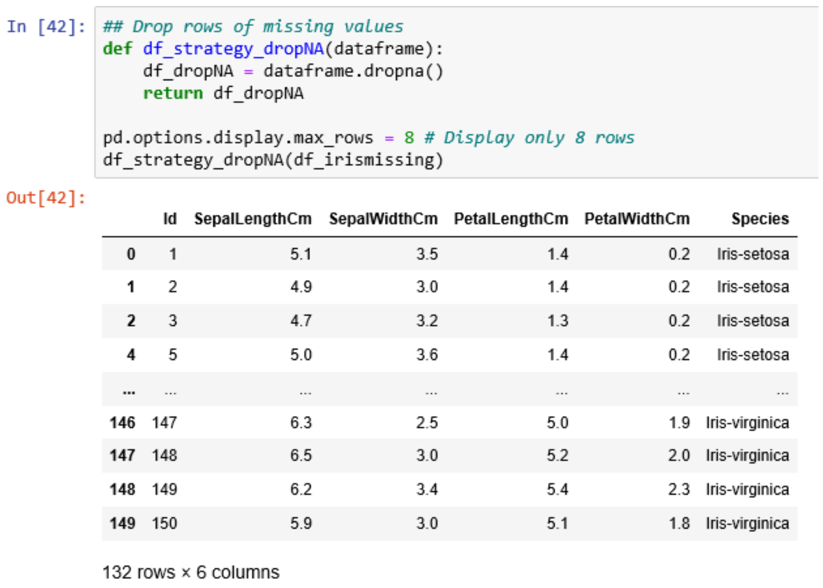


```
In [3]: ## Find the row number of each instance that has missing values
        null_data = df_irismissing[df_irismissing.isnull().any(axis=1)]
        print(null_data['Id'].values)

        [  4  11  20  28  32  33  47  58  63  83  91  97 102 119 129 132 138 145]
```

Figure 1.1.2: Code for finding the row numbers with missing values.

1.2. Write a program to drop missing values, and describe other two strategies (median, mean) for handling missing values and write a function to implement these strategies. [10 marks]

For this sub-task, I implemented three strategies to handle missing values. The first one is by dropping rows of missing values (see Figure 1.2.1). I started by creating a function *df_strategy_dropNa(dataframe)* that returns a new data frame after dropping any rows that contains NaN values in a data frame by using *.dropna()* syntax. I then inputted the *df_irismissing* data frame into the function. As a result, for example, the output in Figure 1.2.1 shows it is apparent that the sequence skipped the row of the 4th id as it contains NaN (refer to Figure 1.1.1) and instead jumped from 3rd to 5th id directly. Since the function drops all the 18 rows (refer to Figure 1.1.2) containing NaN in *df_irismissing*, the new data frame contains only 150 – 18 = 132 rows (see Figure 1.2.1).



```
In [42]: ## Drop rows of missing values
         def df_strategy_dropNA(dataframe):
             df_dropNA = dataframe.dropna()
             return df_dropNA

         pd.options.display.max_rows = 8 # Display only 8 rows
         df_strategy_dropNA(df_irismissing)
```

Out[42]:

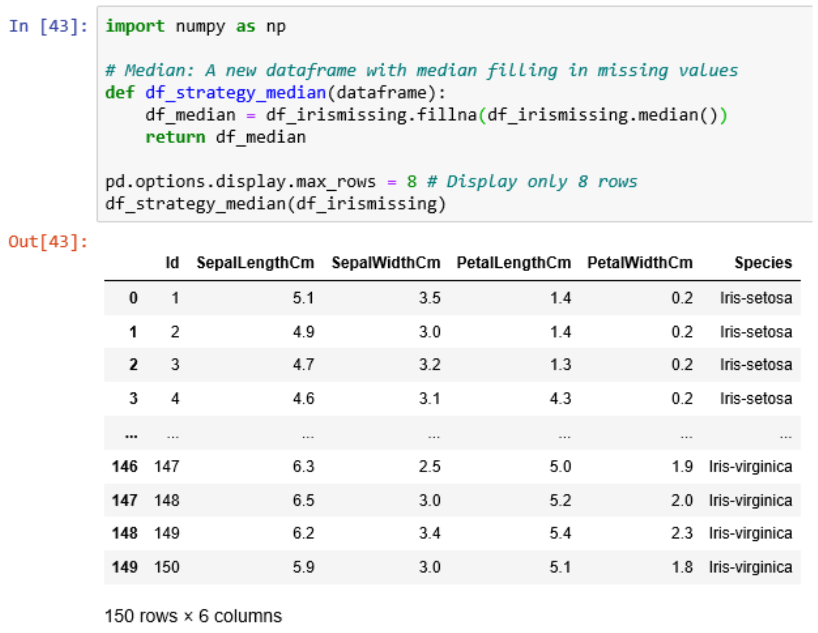
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...	...
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

132 rows × 6 columns

Figure 1.2.1: Code for the first strategy for handling missing values, by dropping rows containing missing values.

Besides the drop function, another strategy to handle the missing values is to impute the missing values with the median as a substituted value. First, I imported the numpy packages to activate the *.median()* syntax. I created such function *df_strategy_median(dataframe)* by computing the median of each column using the *.median()* syntax which is then used to replace NaN in each column using the *.fillna()* syntax, as shown in Figure 1.2.2. For example, as shown in the output in Figure 1.2.2, the output does not skip the 3rd id, unlike the first strategy by dropping rows containing NaN, and instead replace the NaN in the 'PetalLengthCm' column and the 3rd id row with the median of 'PetalLengthCm' column. The number of rows is still the original 150 (see the output in Figure 1.2.2). Furthermore, I printed out the *df_strategy_median(df_irismissing).isnull().sum()*

syntax to ensure that the median strategy leaves all columns with indeed no missing values left (see Figure 1.2.3).

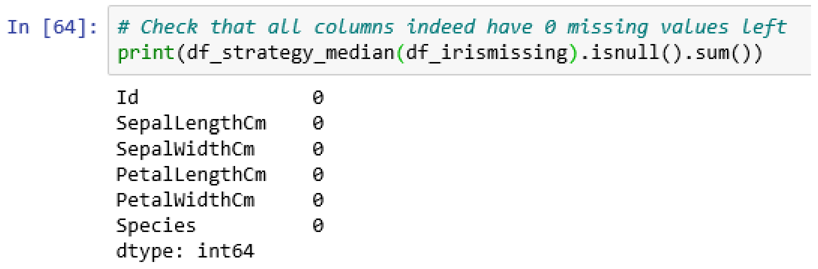


```
In [43]: import numpy as np

         # Median: A new dataframe with median filling in missing values
         def df_strategy_median(dataframe):
             df_median = df_irismissing.fillna(df_irismissing.median())
             return df_median

         pd.options.display.max_rows = 8 # Display only 8 rows
         df_strategy_median(df_irismissing)
Out[43]:
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	4.3	0.2	Iris-setosa
...	...	...	...	...	...	...
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 6 columns

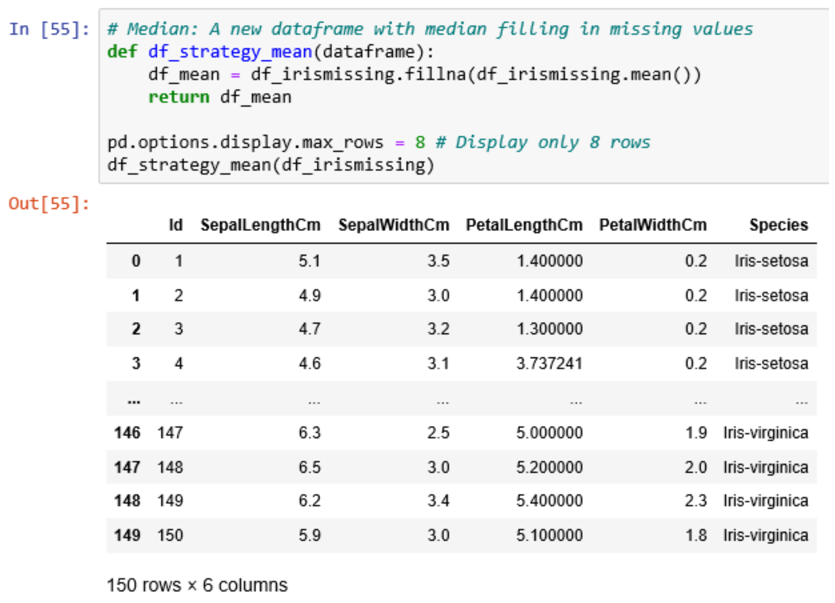
Figure 1.2.2: Code for the second strategy for handling missing values, by imputing NaN with median values.



```
In [64]: # Check that all columns indeed have 0 missing values left
         print(df_strategy_median(df_irismissing).isnull().sum())

         Id               0
         SepalLengthCm    0
         SepalWidthCm     0
         PetalLengthCm    0
         PetalWidthCm     0
         Species          0
         dtype: int64
```

Figure 1.2.3: Code for ensuring zero missing values left in new data frame after the median strategy.

Another imputing strategy to handle the missing values is by using the mean value. This is similar to the median strategy, with the difference only using the mean value instead of the median value to impute NaN values. I created such function *df_strategy_median(dataframe)* by computing the mean of each column using the *.mean()* syntax, after imported the numpy package previously, which is then used to replace NaN in each column using the *.fillna()* syntax, as shown in Figure 1.2.4. For example, as shown in the output in Figure 1.2.4, similar to the median strategy, the output does not skip the 3rd id and instead replace the NaN in the 'PetalLengthCm' column and the 3rd id row with the mean of 'PetalLengthCm' column. The number of rows is also still the original 150 (see the output in Figure 1.2.2). Furthermore, I also printed out the *df_strategy_mean(df_irismissing).sum()* syntax to ensure that the mean strategy leaves all columns with indeed no missing values left (see Figure 1.2.5).



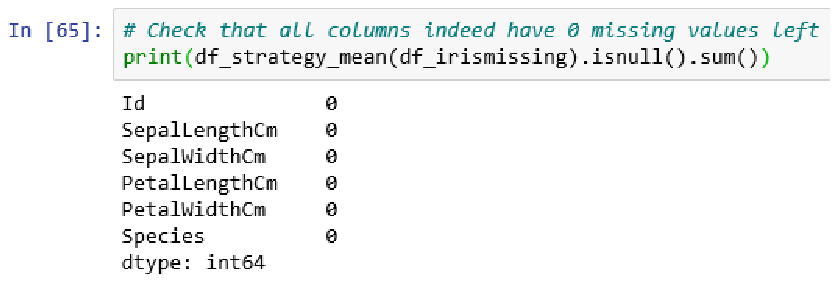
```
In [55]: # Median: A new dataframe with median filling in missing values
         def df_strategy_mean(dataframe):
             df_mean = df_irismissing.fillna(df_irismissing.mean())
             return df_mean

         pd.options.display.max_rows = 8 # Display only 8 rows
         df_strategy_mean(df_irismissing)
```

Out[55]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.400000	0.2	Iris-setosa
1	2	4.9	3.0	1.400000	0.2	Iris-setosa
2	3	4.7	3.2	1.300000	0.2	Iris-setosa
3	4	4.6	3.1	3.737241	0.2	Iris-setosa
...	...	...	...	...	...	...
146	147	6.3	2.5	5.000000	1.9	Iris-virginica
147	148	6.5	3.0	5.200000	2.0	Iris-virginica
148	149	6.2	3.4	5.400000	2.3	Iris-virginica
149	150	5.9	3.0	5.100000	1.8	Iris-virginica

150 rows × 6 columns

Figure 1.2.4: Code for the third strategy for handling missing values, by imputing NaN with mean values.

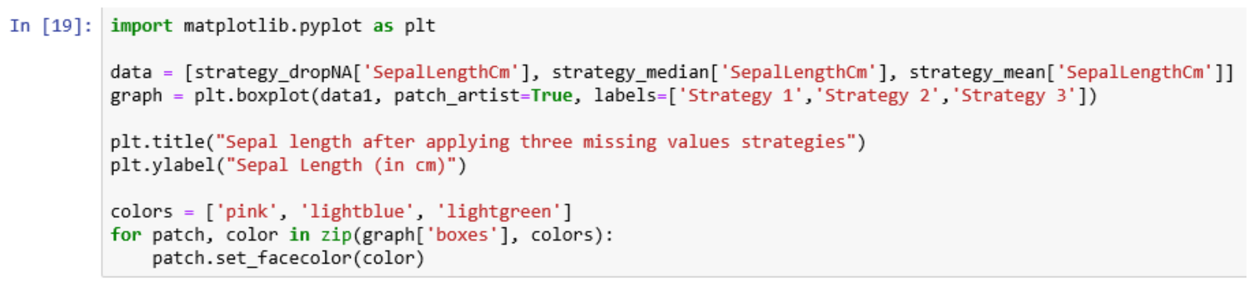


```
In [65]: # Check that all columns indeed have 0 missing values left
         print(df_strategy_mean(df_irismissing).isnull().sum())

         Id               0
         SepalLengthCm    0
         SepalWidthCm     0
         PetalLengthCm    0
         PetalWidthCm     0
         Species          0
         dtype: int64
```

Figure 1.2.3: Code for ensuring zero missing values left in new data frame after the mean strategy.

1.3. Compare the results of applying each missing value strategy using some visualization method. [10 marks]

After creating three functions of each missing value strategy, now I used boxplot to visualize the results comparison. The code starts with importing the *matplotlib.pyplot* package as *plt*. Then, we store the data of an attribute (in Figure 1.3.1 example, the attribute is Sepal Length) after applied with three missing value strategy into an array called *data*. Then, we draw a boxplot graph out of *data*, label the axes accordingly, and set a unique colour to each boxplot that represents each strategy. Figure 1.3.1 shows the code as well as the boxplot visualization for sepal length. Figure 1.3.2, 1.3.3, 1.3.4 show the boxplot visualizations for sepal width, petal length, and petal width respectively.



```
In [19]: import matplotlib.pyplot as plt

         data = [strategy_dropNA['SepalLengthCm'], strategy_median['SepalLengthCm'], strategy_mean['SepalLengthCm']]
         graph = plt.boxplot(data1, patch_artist=True, labels=['Strategy 1','Strategy 2','Strategy 3'])

         plt.title("Sepal length after applying three missing values strategies")
         plt.ylabel("Sepal Length (in cm)")

         colors = ['pink', 'lightblue', 'lightgreen']
         for patch, color in zip(graph['boxes'], colors):
             patch.set_facecolor(color)
```

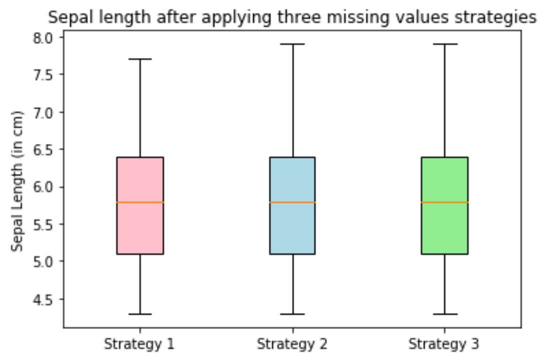


Figure 1.3.1: Code for boxplot visualization of three missing value strategies, i.e. drop values, imputation with median values, and imputation with mean values. An example is given for sepal length attribute in the figure.

As shown in Figure 1.3.1, the three boxplots generally look similar, especially strategies 2 and 3. The only apparent difference is that the upper whisper (i.e. the maximum) of strategy one is apparently lower than the other two strategies.

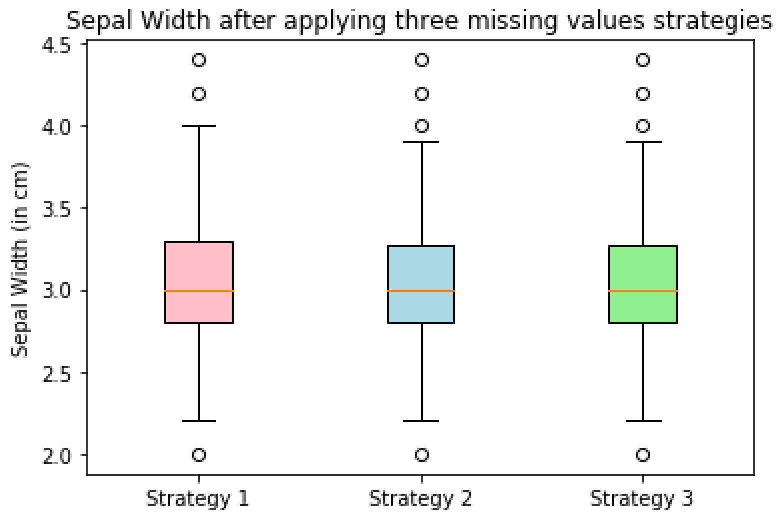


Figure 1.3.2: Boxplot visualization of the sepal width. Each boxplot represents the sepal width data after applied with each of the three missing value strategies, i.e. drop values, imputation with median values, and imputation with mean values.

As shown in Figure 1.3.2, like the case in Figure 1.3.1, the three boxplots also generally look similar, especially strategies 2 and 3. The only apparent differences are that the upper whisper (i.e. the maximum) of strategy one is apparently lower than the other two strategies, and the outliers of strategy one are quite apparently different compared to strategies 2 and 3.

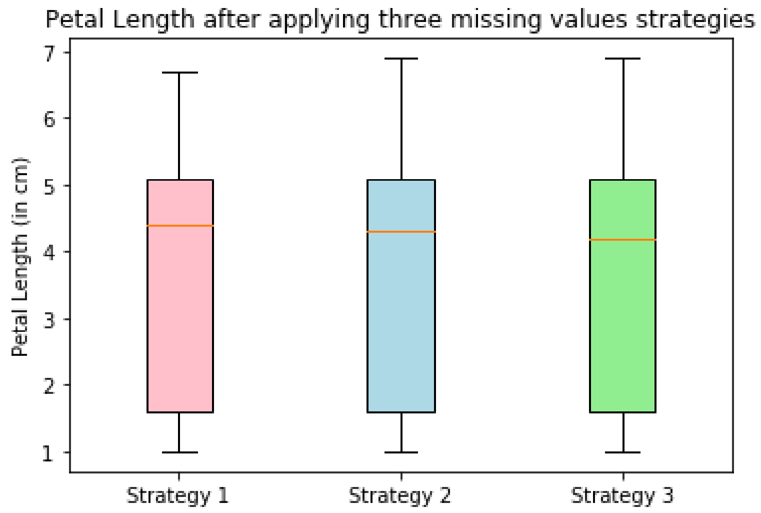


Figure 1.3.3: Boxplot visualization of the petal length. Each boxplot represents the petal length data after applied with each of the three missing value strategies, i.e. drop values, imputation with median values, and imputation with mean values.

Figure 1.3.3 shows that, like the case in Figure 1.3.1 and 1.3.2, the three boxplots also generally look similar, especially strategies 2 and 3. The only apparent differences are that the upper whisper (i.e. the maximum) of strategy one is apparently lower than the other two strategies.

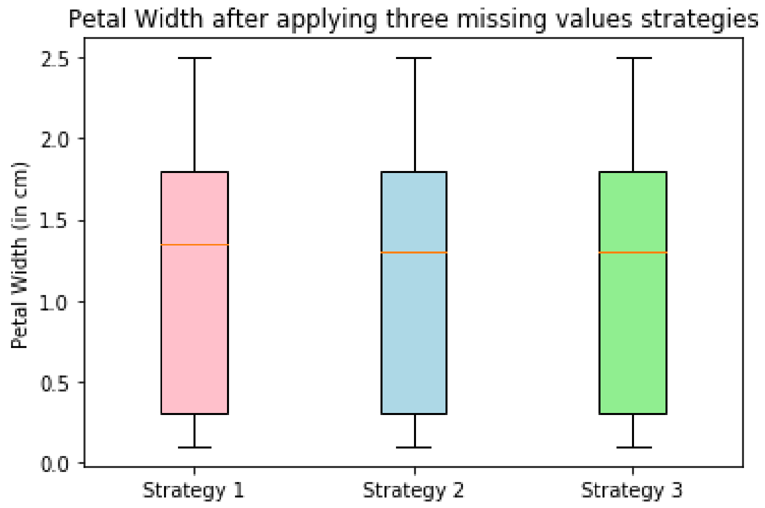


Figure 1.3.4: Boxplot visualization of the petal width. Each boxplot represents the petal width data after applied with each of the three missing value strategies, i.e. drop values, imputation with median values, and imputation with mean values.

Figure 1.3.4 shows that the three boxplots also generally look similar. Unlike the previous figures, however, strategy 1 does not seem too far off strategies 2 and 3.

In conclusion, the three strategies do not seem to significantly differ from each other. Strategies 2 and 3 look especially similar, while strategy 1 can result slightly different from the other two strategies.

Task 2: Decision Tree

2.1. Manually generate the decision tree for the passenger survival dataset below. Use information gain as the split measure. [20 marks]

To generate a decision tree for the passenger survival dataset, information gain is used as the split measure. The information gain equation *IG(S, A)* takes two inputs, the set and the attribute. In the passenger survival dataset, *A = {Class, Sex, Age}*. As can be seen in Equation 2.1.1, the information gain equation requires the computation of entropy in advance. On this coursework, to compute the information gain, the Shannon Entropy will be used which is shown in Equation 2.1.2.

$$IG(S, A) = Entropy(S) - \sum_{i=1} \frac{|S_{u_i}|}{|S|} Entropy(S_{u_i}) \qquad \text{[Equation 2.1.1]}$$

$$Entropy(S) = -\sum_{i=1}^{c} p_i \, log_2 \, p_i \qquad \text{[Equation 2.1.2]}$$

To work on that, I first calculated the frequency of each condition of survival, i.e. the positive and negative survival, and the total passengers. These can be seen in Table 2.1.1.

Number of total passengers (N)	1365
Number of total passengers survived (N_{pos})	548
Number of total passengers not survived (N_{neg})	817

Table 2.1.1: Frequency of total passengers and both survival conditions, i.e. those who survived and those who did not.

After obtaining the frequency of total survival and each condition survival , I computed the entropy of the passenger survival by using the Shannon Entropy (see Equation 2.1.2), in which $p_i = \{\frac{N_{pos}}{N}, \frac{N_{neg}}{N}\}$ so that the specific equation to compute the entropy in terms of passenger survival is as shown in Equation 2.1.3.

$$Entropy(S) = -\left(\frac{N_{pos}}{N} log_2 \frac{N_{pos}}{N} + \frac{N_{neg}}{N} log_2 \frac{N_{neg}}{N}\right) \qquad \text{[Equation 2.1.3]}$$

where: S = Set of an attribute
p_i = Probability of each condition in S
N_{pos} = Frequency of positive survival
N_{neg} = Frequency of negative survival
N = Frequency of total survival

Substituting the values from Table 2.1.1 into Equation 2.1.2, we get:

$$Entropy(S) = -\left(\frac{548}{1365} log_2 \frac{548}{1365} + \frac{817}{1365} log_2 \frac{817}{1365}\right)$$

$$= -0.401465 \, log_2 \, 0.401465 - 0.598535 \, log_2 \, 0.598535$$

$$= 0.9718012324831182 \qquad \text{[Equation 2.1.4]}$$

Besides the entropy of the source $Entropy(S)$, the information gain equation in Equation 2.1.1 also requires the sum of the product of the probability of every attribute $\frac{|S_{u_i}|}{|S|}$ and the entropy of every attribute $Entropy(S_{u_i})$.

As mentioned previously, *A = {Class, Sex, Age}* so we need to compute *IG(S, Class), IG(S, Sex), and IG(S, Age)*. First, we will start with *IG(S, Class)*. Since u_i is the type or branch of each attribute, then for class attribute has $u_i = \{Class1,\ Class2,\ Class3\}$. Second, we have *IG(S, Sex)* where $u_i = \{Male,\ Female\}$. Finally, we have *IG(S, Age)* where $u_i = \{Child, Adult\}$

The information gain equation further requires the computation of the probability and the entropy of each type of attribute u_i, generated from Equation 2.1.3 in advance. Thus, I computed those statistics for the attributes Class, Sex, and Age, as shown in Table 2.1.2, Table 2.1.3, and Table 2.1.4 respectively.

Class Attribute		
Type	Statistical Measure	Result
Class 1	Number of total passengers (N)	349
	Number of total passengers survived (N_{pos})	237
	Number of total passengers not survived (N_{neg})	112
	Entropy = $-\left(\frac{237}{349} log_2 \frac{237}{349} + \frac{112}{349} log_2 \frac{112}{349}\right)$	0.9053757727444973
Class 2	Number of total passengers (N)	275
	Number of total passengers survived (N_{pos})	98
	Number of total passengers not survived (N_{neg})	177
	Entropy = $-\left(\frac{98}{275} log_2 \frac{98}{275} + \frac{177}{275} log_2 \frac{177}{275}\right)$	0.9396232735466368
Class 3	Number of total passengers (N)	741
	Number of total passengers survived (N_{pos})	213
	Number of total passengers not survived (N_{neg})	528
	Entropy = $-\left(\frac{213}{741} log_2 \frac{213}{741} + \frac{528}{741} log_2 \frac{528}{741}\right)$	0.8654036268680916

Table 2.1.2: Statistics of each type of the class attribute, including number of total passengers, frequency of both survival condition, i.e. those who survived and those who did not, and its entropy generated from Equation 2.1.3.

Sex Attribute		
Type	Statistical Measure	Result
Male	Number of total passengers (N)	833
	Number of total passengers survived (N_{pos})	169
	Number of total passengers not survived (N_{neg})	664
	Entropy = $-\left(\frac{169}{833} log_2 \frac{169}{833} + \frac{664}{833} log_2 \frac{664}{833}\right)$	0.7276531089597287
Female	Number of total passengers (N)	532
	Number of total passengers survived (N_{pos})	379
	Number of total passengers not survived (N_{neg})	153
	Entropy = $-\left(\frac{379}{532} log_2 \frac{379}{532} + \frac{153}{532} log_2 \frac{153}{532}\right)$	0.8655929234733335

Table 2.1.3: Statistics of each type of the sex attribute, including number of total passengers, frequency of both survival condition, i.e. those who survived and those who did not, and its entropy generated from Equation 2.1.3.

Age Attribute		
Type	Statistical Measure	Result
Child	Number of total passengers (N)	132
	Number of total passengers survived (N_{pos})	70
	Number of total passengers not survived (N_{neg})	62
	Entropy = $-\left(\frac{70}{132} log_2 \frac{70}{132} + \frac{62}{132} log_2 \frac{62}{132}\right)$	0.997348797918045
Adult	Number of total passengers (N)	1233
	Number of total passengers survived (N_{pos})	478
	Number of total passengers not survived (N_{neg})	755
	Entropy = $-\left(\frac{478}{1233} log_2 \frac{478}{1233} + \frac{755}{1233} log_2 \frac{755}{1233}\right)$	0.963280990286707

Table 2.1.4: Statistics of each type of the age attribute, including number of total passengers, frequency of both survival condition, i.e. those who survived and those who did not, and its entropy generated from Equation 2.1.3.

Using the data in Table 2.1.2, Table 2.1.3, and Table 2.1.4, along with *Entropy(S)* = 0.971. .. from Equation 2.1.4, we can compute *IG(S, Class), IG(S, Sex), IG(S, Age)* respectively corresponding to Equation 2.1.5 as follows:

$$IG(S, Class) = 0.9718012324831182 - \left(\frac{|349|}{|1365|}(0.9053757727444973) + \frac{|275|}{|1365|}(0.9396232735466368) + \frac{|741|}{|1365|}(0.8654036268680916)\right)$$

$$= 0.08122494499417288 \quad \text{[Equation 2.1.6]}$$

$$IG(S, Sex) = 0.9718012324831182 - \left(\frac{|833|}{|1365|}(0.7276531089597287) + \frac{|532|}{|1365|}(0.8655929234733335)\right)$$

$$= 0.19038696504629216 \quad \text{[Equation 2.1.7]}$$

$$IG(S, Age) = 0.9718012324831182 - \left(\frac{|132|}{|1365|}(0.997348797918045) + \frac{|1233|}{|1365|}(0.963280990286707)\right)$$

$$= 0.005225772887007096 \quad \text{[Equation 2.1.8]}$$

From Equations 2.1.6 – 2.1.8, it is apparent that the sex attribute has the highest information gain, *IG(S,Sex)* = 0.19, and so the biggest reduction in entropy. As a result, the sex attribute becomes the root node.

Now, to figure out which of the two other attributes class and age will be the leaf node of each of the sex node branch, we compute the information gain again for the class and age attributes. The difference this time is that of using the general S set as the input in *IG(S,A)*, the set is replaced by S_{Male} and S_{Female}, such as in Equations 2.1.9 and 2.1.10 below.

$$IG(S_{Male}, A) = Entropy(S_{Male}) - \sum_{i=1} \frac{|S_{u_i}|}{|S_{Male}|} Entropy(S_{u_i}) \quad \text{[Equation 2.1.9]}$$

$$IG(S_{Female}, A) = Entropy(S_{Female}) - \sum_{i=1} \frac{|S_{u_i}|}{|S_{Female}|} Entropy(S_{u_i}) \quad \text{[Equation 2.1.10]}$$

Also note that S_{u_i} is also different as the set would have been filtered by 'Sex' == 'Male' and 'Sex' == 'Female'. In Python code, this would be dfPassengerSurvival[dfPassengerSurvival['Sex']=='Male'] and dfPassengerSurvival[dfPassengerSurvival['Sex']=='Female'] respectively. Hence, $Entropy(S_{u_i})$ would also be different from when the set is still the whole dataset. Table 2.1.5 and Table 2.1.6 show the statistics of the class and age attribute respectively of the filtered data frame that contains only male.

(Sex=Male|Class) Attribute		
Type	Statistical Measure	Result
Class 1	Number of total passengers (N)	164
	Number of total passengers survived (N_{pos})	56
	Number of total passengers not survived (N_{neg})	108
	Entropy = $-\left(\frac{56}{164} log_2 \frac{56}{164} + \frac{108}{164} log_2 \frac{108}{164}\right)$	0.9262122127346665
Class 2	Number of total passengers (N)	189
	Number of total passengers survived (N_{pos})	25
	Number of total passengers not survived (N_{neg})	164
	Entropy = $-\left(\frac{25}{189} log_2 \frac{25}{189} + \frac{164}{189} log_2 \frac{164}{189}\right)$	0.5636448924658671
Class 3	Number of total passengers (N)	480
	Number of total passengers survived (N_{pos})	88
	Number of total passengers not survived (N_{neg})	392
	Entropy = $-\left(\frac{88}{480} log_2 \frac{88}{480} + \frac{392}{480} log_2 \frac{392}{480}\right)$	0.6873150928309273

Table 2.1.5: Statistics of each type of the class attribute from (Sex==Male)-filtered data frame, including number of total passengers, frequency of both survival condition, i.e. those who survived and those who did not, and its entropy generated from Equation 2.1.9.

(Sex=Male|Age) Attribute		
Type	Statistical Measure	Result
Child	Number of total passengers (N)	52
	Number of total passengers survived (N_{pos})	27
	Number of total passengers not survived (N_{neg})	25
	Entropy = $-\left(\frac{27}{52} log_2 \frac{27}{52} + \frac{25}{52} log_2 \frac{25}{52}\right)$	0.9989326546260581
Adult	Number of total passengers (N)	781
	Number of total passengers survived (N_{pos})	142
	Number of total passengers not survived (N_{neg})	639

	Entropy = $-\left(\frac{25}{189}log_2\frac{25}{189} + \frac{164}{189}log_2\frac{164}{189}\right)$	0.6840384356390417

Table 2.1.6: Statistics of each type of the age attribute from (Sex==Male)-filtered data frame, including number of total passengers, frequency of both survival condition, i.e. those who survived and those who did not, and its entropy generated from Equation 2.1.9.

Given the data from Table 2.1.5 and 2.1.6, along with *Entropy(S_{Male})* that has been calculated in Table 2.1.3, we compute $IG(S_{Male}, Class)$ and $IG(S_{Male}, Age)$:

$$IG(S_{Male}, Class) = 0.7276531089597287 - \left(\frac{|164|}{|833|}(0.9262122127346665) + \frac{|189|}{|833|}(0.5636448924658671) + \frac{|480|}{|833|}(0.6873150928309273)\right)$$

$$= 0.02136387471797685 \quad \text{[Equation 2.1.11]}$$

$$IG(S_{Male}, Age) = 0.7276531089597287 - \left(\frac{|52|}{|833|}(0.9989326546260581) + \frac{|781|}{|833|}(0.6840384356390417)\right)$$

$$= 0.02395741115102923 \quad \text{[Equation 2.1.12]}$$

Since $IG(S_{Male}, Age)$ is the largest information gain in S_{Male}, then the age attribute becomes the leaf node of the male branch.

To make sure that the age attribute does not also belong as a leaf node to the male branch, we repeat a similar procedure of computing both *IG(S_{Female}, Class)* and *IG(S_{Female}, Age)*. Table 2.1.7 and Table 2.1.8 show the statistics of the class and age attribute respectively of the filtered data frame that contains only female.

(Sex=Female|Class) Attribute		
Type	Statistical Measure	Result
(Sex=Female|Class1)	Number of total passengers (N)	185
	Number of total passengers survived (N_{pos})	181
	Number of total passengers not survived (N_{neg})	4
	Entropy = $-\left(\frac{181}{185}log_2\frac{181}{185} + \frac{4}{185}log_2\frac{4}{185}\right)$	0.15045116018085175
(Sex=Female|Class2)	Number of total passengers (N)	86
	Number of total passengers survived (N_{pos})	73
	Number of total passengers not survived (N_{neg})	13
	Entropy = $-\left(\frac{73}{86}log_2\frac{73}{86} + \frac{13}{86}log_2\frac{13}{86}\right)$	0.6127425554686624
(Sex=Female|Class3)	Number of total passengers (N)	216
	Number of total passengers survived (N_{pos})	125
	Number of total passengers not survived (N_{neg})	136
	Entropy = $-\left(\frac{125}{216}log_2\frac{125}{216} + \frac{136}{216}log_2\frac{136}{216}\right)$	0.9987183260993595

Table 2.1.7: Statistics of each type of the class attribute from (Sex==Female)-filtered data frame, including number of total passengers, frequency of both survival condition, i.e. those who survived and those who did not, and its entropy generated from Equation 2.1.10.

(Sex=Female|Age) Attribute		
Type	Statistical Measure	Result
(Sex=Female|Child)	Number of total passengers (N)	80
	Number of total passengers survived (N_{pos})	43
	Number of total passengers not survived (N_{neg})	37
	Entropy = $-\left(\frac{43}{80}log_2\frac{43}{80} + \frac{37}{80}log_2\frac{37}{80}\right)$	0.9959386076315955
(Sex=Female|Adult)	Number of total passengers (N)	452
	Number of total passengers survived (N_{pos})	336
	Number of total passengers not survived (N_{neg})	116
	Entropy = $-\left(\frac{336}{452}log_2\frac{336}{452} + \frac{116}{452}log_2\frac{116}{452}\right)$	0.8216292954053164

Table 2.1.8: Statistics of each type of the age attribute from (Sex==Female)-filtered data frame, including number of total passengers, frequency of both survival condition, i.e. those who survived and those who did not, and its entropy generated from Equation 2.1.10.

Given the data from Table 2.1.7 and 2.1.8, along with *Entropy*(S_{Female}) that has been calculated in Table 2.1.3, we compute $IG(S_{Female}, Class)$ and $IG(S_{Female}, Age)$:

$$IG(S_{Female}, Class) = 0.8655929234733335 - \left(\frac{|164|}{|833|}(0.15045116018085175) + \frac{|189|}{|833|}(0.6127425554686624) + \frac{|480|}{|833|}(0.9987183260993595)\right)$$

$$= 0.2242493003235302 \qquad \text{[Equation 2.1.11]}$$

$$IG(S_{Female}, Age) = 0.8655929234733335 - \left(\frac{|52|}{|833|}(0.9959386076315955) + \frac{|781|}{|833|}(0.8216292954053164)\right)$$

$$= 0.017751701417448862 \qquad \text{[Equation 2.1.12]}$$

Since $IG(S_{Female}, Class)$ is the largest information gain in S_{Female}, then the class attribute becomes the node of the female branch. This is convenient, since on the contrary, $IG(S_{Male}, Age)$ is the largest information gain in S_{Male}, making the age attribute as the node of the male branch.

Generating a decision tree from this information would result to Figure 2.1.1. The sex attribute becomes the root node because *IG(S, Sex)* is the largest amongst *IG(S, A).* From then on, it is split into the two types of sex: female and male. Amongst $IG(S_{Female}, A)$, $IG(S_{Female}, Class)$ is the largest, making the class attribute as the node of the female branch. On the contrary, $IG(S_{Male}, Age)$ is the largest amongst $IG(S_{Male}, A)$, making the age attribute as the node of the male branch. Each leaf node is dependent on which of N_{pos} and N_{neg} is the majority in $IG(S_{Sex}, A)$, given that sex = {female, male}. The probability of the leaf node is calculated by whichever the majority is from N_{pos} and

N_{neg}, divided by N in $IG(S_{Male}, Age)$ or $IG(S_{Female}, Class)$. Data of attributes (Sex=Male|Age) and (Sex=Female|Class) can be obtained from Tables 2.1.6 and 2.1.7 respectively.

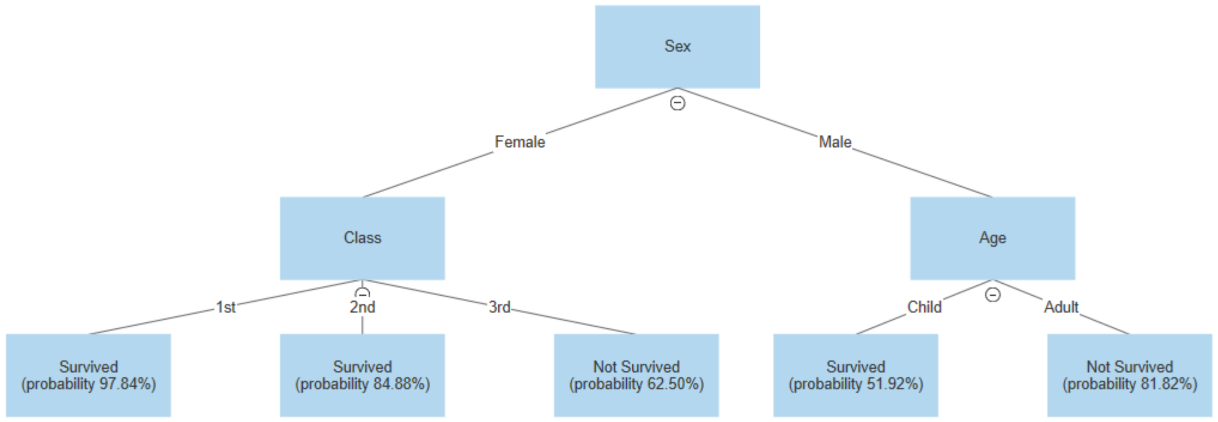

Sex
Female
Male
Class
Age
1st
2nd
3rd
Child
Adult
Survived (probability 97.84%)
Survived (probability 84.88%)
Not Survived (probability 62.50%)
Survived (probability 51.92%)
Not Survived (probability 81.82%)

Figure 2.1.1: Decision tree of the passenger survival dataset.

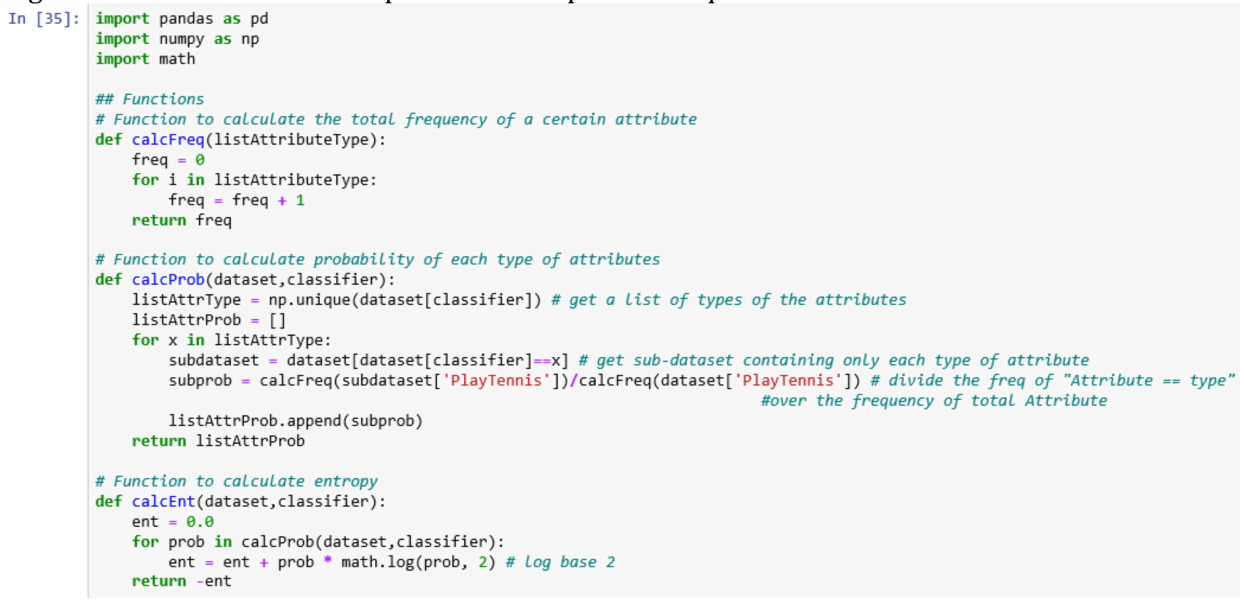
2.2. Tennis data

2.2.1 Write a function that computes the entropy of a set S with $Npos$ positive observations and $Nneg$ negative observations.

To create a function that computes entropy, I created in advance a function to calculate probability, which requires a function to calculate frequency first. Thus, I started with creating a function that can calculate frequency after taking an input of a list of instances of an attribute type. The function starts with a *freq = 0* and for every instance in the list, add 1 into the *freq* variable. Note that this frequency can be replaced by a *count()* syntax to count each instance or row that contains the attribute type for this specific dataset. I just made the frequency calculator function so I will have such generalized function in dispose if, for example, each row in the dataset contains a specified frequency, like the Passenger Survival dataset, instead of just a frequency of one.

Subsequently, I can create the probability calculator function which takes input of the dataset *S* and a classifier for the target, which in this case would be 'PlayTennis'. The function starts by taking a subset of the dataset that only contains the value of the classifier (or the 'PlayTennis') column. Then, it removes duplicates of values in the subset and would only return each unique value, i.e. 'No' and 'Yes' for the 'PlayTennis' column, and store them in a list *listAttrType*. After that, for each type in *listAttrType*, it creates a sub-dataframe in which only contains classifier of each type, e.g. only contains dataframe that 'PlayTennis' == 'No' for one of the attribute types. Then it calculates the PlayTennis frequency of the sub-dataframe, as such as it would be either N_{pos} or N_{neg}, using the frequency calculator function. It also calculates the PlayTennis frequency of the dataframe, as such as it would be *N*. It then divides the survival frequency of the sub-dataframe by of the dataframe, so that it becomes the probability of each attribute type. Each probability is then stored in an empty array which the function returns.

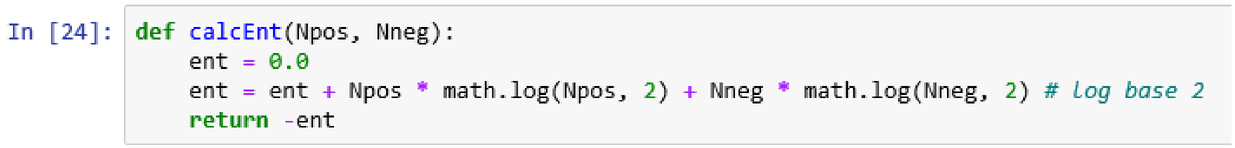
After having the probability calculator function, our entropy calculator function would be simpler. The function starts with creating a variable *ent = 0.0*. Then, for every probability in the probability list returned by the probability calculator function, *ent* is added by $p_i \; log_2 \; p_i$. Note that the second argument in *math.log()* is the log base. After every probability has been computed, it returns the negative of the *ent*. This whole process corresponds to Equation 2.1.2.



```
In [35]: import pandas as pd
         import numpy as np
         import math

         ## Functions
         # Function to calculate the total frequency of a certain attribute
         def calcFreq(listAttributeType):
             freq = 0
             for i in listAttributeType:
                 freq = freq + 1
             return freq

         # Function to calculate probability of each type of attributes
         def calcProb(dataset,classifier):
             listAttrType = np.unique(dataset[classifier]) # get a list of types of the attributes
             listAttrProb = []
             for x in listAttrType:
                 subdataset = dataset[dataset[classifier]==x] # get sub-dataset containing only each type of attribute
                 subprob = calcFreq(subdataset['PlayTennis'])/calcFreq(dataset['PlayTennis']) # divide the freq of "Attribute == type"
                                                                                               #over the frequency of total Attribute
                 listAttrProb.append(subprob)
             return listAttrProb

         # Function to calculate entropy
         def calcEnt(dataset,classifier):
             ent = 0.0
             for prob in calcProb(dataset,classifier):
                 ent = ent + prob * math.log(prob, 2) # log base 2
             return -ent
```

Figure 2.2.1.1: Code for the function to calculate frequency, which is used for the function to calculate probability, which is used for the function to calculate entropy.

Alternatively, if p_i is already listed N_{pos} and N_{neg}, then the entropy calculator function can be more straightforward as follows:



```
In [24]: def calcEnt(Npos, Nneg):
             ent = 0.0
             ent = ent + Npos * math.log(Npos, 2) + Nneg * math.log(Nneg, 2) # log base 2
             return -ent
```

Figure 2.2.1.2: Code for the function to calculate entropy, if N_{pos} and N_{neg} are already given.

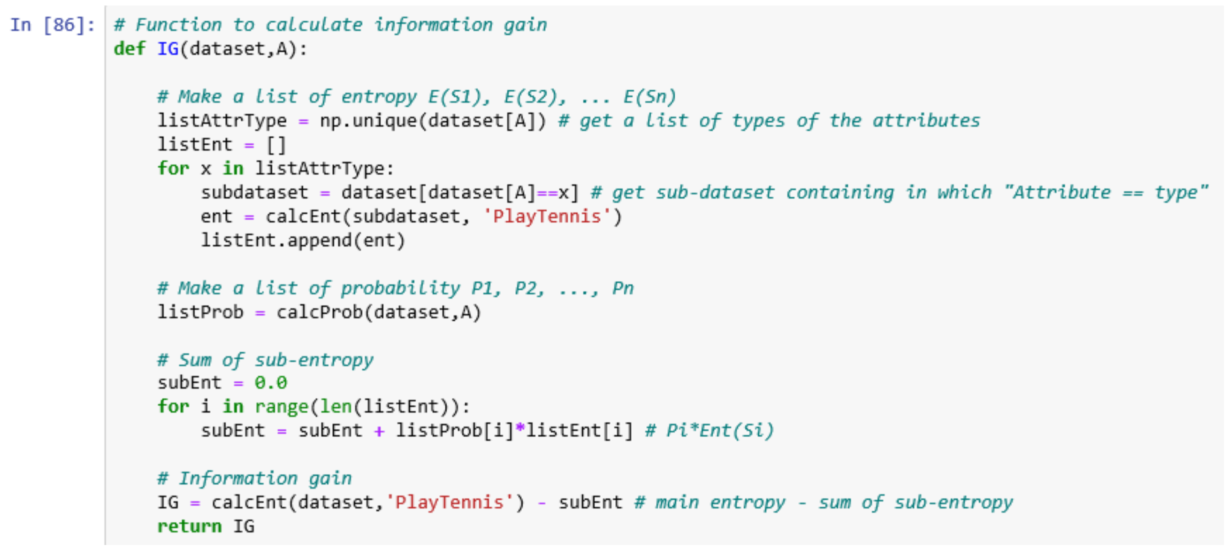
2.2.2 Write a function that takes input as a set *S* of observations and an attribute *A* from these observations, and calculates the information gain denoted as *IG(S,A)*, as if we were to split on that attribute in the context of the ID3 decision tree algorithm. [10 marks]

After creating an entropy function, now we create an information gain function. The function takes input of the dataset and attribute. First, it makes a list of entropy *listEnt* containing *E(S_1), E(S_2), …, E(S_n)*. This is started by finding each unique attribute type in the input attribute column of the dataset and store them into an array *listAttrType*. Then, for every attribute type in *listAttrType*, we get the sub-dataset in which the attribute is of that particular attribute type. The entropy of that sub-dataset is then calculated using the entropy function I created earlier, and then append that into a list of entropy *listEnt*.

Second, we create a list of probability *listProb* containing *P_1, P_2, …, P_n*. This list is obtained by using the probability function of input dataset and attribute, which gives a list of probability of the attribute type in the dataset by default.

Third, we compute the cumulative sum of sub-entropy, i.e. $\sum_{i=1} \frac{|S_{u_i}|}{|S|} Entropy(S_{u_i})$ from Equation 2.1.1. Such sum is stored in a variable *subEnt*, which we initiate by assigning its value to 0.0. We then compute the product of *E(S$_i$)* and *P$_i$* for every *i* as it goes through the list of entropy (or probability, they are of the same length) until it finishes.

After getting the result of the second half of Equation 2.1.1, we compute the information gain by getting the first half, i.e. $Entropy(S)$, deducted by the second half as computed in the previous step. The first half is computed by using the entropy function created in the previous task and inputting the dataset and inputting 'PlayTennis' as the classifier. The whole function then returns the information gain result.

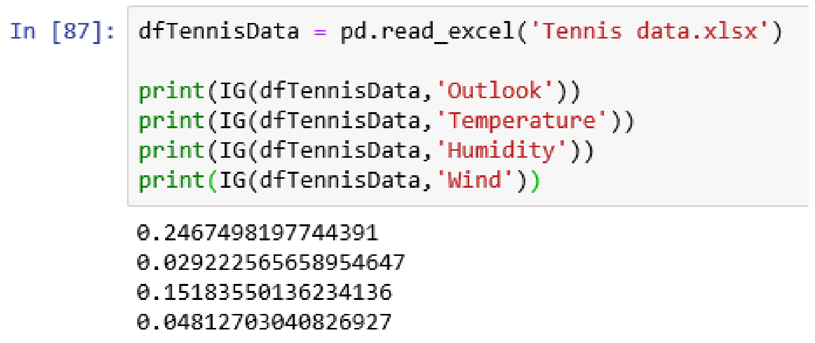


```
In [86]: # Function to calculate information gain
         def IG(dataset,A):

             # Make a list of entropy E(S1), E(S2), ... E(Sn)
             listAttrType = np.unique(dataset[A]) # get a list of types of the attributes
             listEnt = []
             for x in listAttrType:
                 subdataset = dataset[dataset[A]==x] # get sub-dataset containing in which "Attribute == type"
                 ent = calcEnt(subdataset, 'PlayTennis')
                 listEnt.append(ent)

             # Make a list of probability P1, P2, ..., Pn
             listProb = calcProb(dataset,A)

             # Sum of sub-entropy
             subEnt = 0.0
             for i in range(len(listEnt)):
                 subEnt = subEnt + listProb[i]*listEnt[i] # Pi*Ent(Si)

             # Information gain
             IG = calcEnt(dataset,'PlayTennis') - subEnt # main entropy - sum of sub-entropy
             return IG
```

Figure 2.2.1.2: Code for the function to calculate information gain, that takes input of the dataset and attribute.

2.2.3 Estimate the information gain of all the attributes. Which one will you choose for the root node of your decision tree? [10 marks]

As Figure 2.2.3 shows, it is apparent that the outlook attribute has the highest information gain. Consequently, outlook decision will be structured as the root node of the decision tree.



```
In [87]: dfTennisData = pd.read_excel('Tennis data.xlsx')

         print(IG(dfTennisData,'Outlook'))
         print(IG(dfTennisData,'Temperature'))
         print(IG(dfTennisData,'Humidity'))
         print(IG(dfTennisData,'Wind'))

         0.2467498197744391
         0.029222565658954647
         0.15183550136234136
         0.04812703040826927
```

Figure 2.2.3: Code of the results of information gain of all the attributes

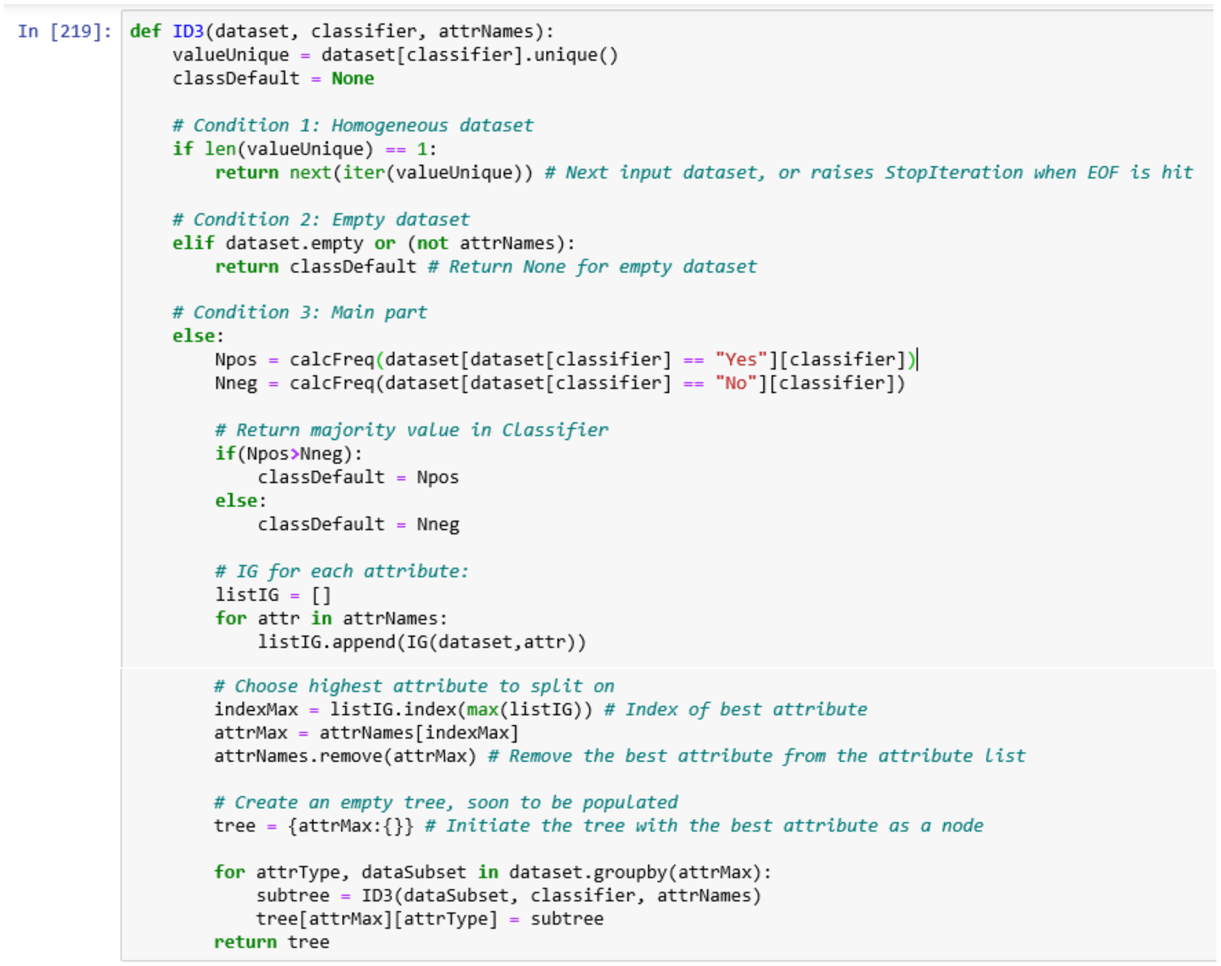
Task 3: ID3 Algorithm

3. Implement the ID3 decision tree from the pseudocode (recursive algorithm) and induce/learn the tree from the tennis data of Task 2.2. [20 marks]

After creating a function of information gain, now we are one step ahead of coding the ID3 decision tree algorithm. I create a function of ID3 which takes input of the dataset, classifier for the target, and a list of attribute names (see Figure 3.1). We start with the first condition that checks whether all observations are of the same class (either only 'Yes' or 'No'), in which case it returns that class. We check this easier by getting the unique values of the classifier into an array and check whether the length is equal to 1. If the observations contain either only 'Yes' or 'No', in which case its array's length is one, then we confirm the dataset is homogenous and returns that only class.

The second condition then checks whether the (sub)dataset (of an attribute) is empty, which then returns the default of class *classDefault* which was previously equated to None.

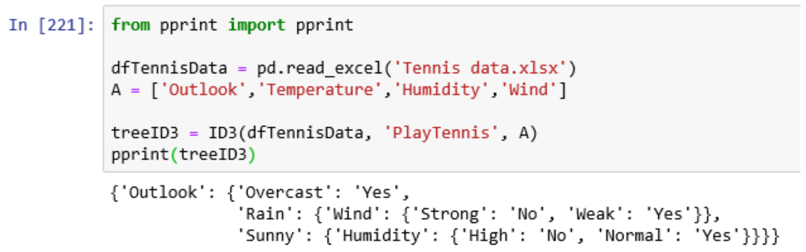
If the case is neither the two previous conditions, then we go to the third condition which is the main part. We start by calculating the frequency of both positive observations N_{pos} and negative observations N_{neg} in the input dataset, using the calcFreq(listAttributeType) function I created in the code shown in Figure 2.2.1.1. We then compare which of N_{pos} and N_{neg} is larger, and updates *classDefault* with whichever the majority happens to be. After that, we move on to calculating the information gain for each attribute. We do that by first creating an empty array called *listIG* = [ ]. Then, for every attribute in the input list of attributes *attrNames*, we find its information gain, given the input dataset and the input attribute, and add it to the *listIG* array. After getting the information gain for every attribute, we then find the highest information gain in *listIG* and then gets its index as *indexMax*. We then find the corresponding index of the highest information gain in the input list of attributes *attrNames*. Afterwards, we remove the attribute of the highest information gain off the list of attributes *attrNames*, so that the next recursion of the tree will not consider such attribute anymore, as it will soon be a node in the decision tree. Subsequently, we create an empty tree which soon will be populated. We initiate the tree with the best attribute as a node. Furthermore, we create an iteration in which we group the dataset according to the best attribute. For example, Outlook = ['Overcast', 'Rain', 'Sunny'], so there will be three subsets, each of which contains of only each type of the outlook attribute. It iterates for every such subset and its attribute type, creating a new subtree as a result of a new call to the ID3 function, only this time it takes input of the sub-dataset according to the attribute type group (instead of the whole dataset), the classifier of the target (same as previous), and the new list of attributes *attrNames* after the best attribute is removed from the list. It then inputs the subtree into the main tree. This repeats for every type of the best attribute. After that, it finally returns the complete tree, which is our finished ID3 decision tree.



```
In [219]: def ID3(dataset, classifier, attrNames):
              valueUnique = dataset[classifier].unique()
              classDefault = None

              # Condition 1: Homogeneous dataset
              if len(valueUnique) == 1:
                  return next(iter(valueUnique)) # Next input dataset, or raises StopIteration when EOF is hit

              # Condition 2: Empty dataset
              elif dataset.empty or (not attrNames):
                  return classDefault # Return None for empty dataset

              # Condition 3: Main part
              else:
                  Npos = calcFreq(dataset[dataset[classifier] == "Yes"][classifier])
                  Nneg = calcFreq(dataset[dataset[classifier] == "No"][classifier])

                  # Return majority value in Classifier
                  if(Npos>Nneg):
                      classDefault = Npos
                  else:
                      classDefault = Nneg

                  # IG for each attribute:
                  listIG = []
                  for attr in attrNames:
                      listIG.append(IG(dataset,attr))

                  # Choose highest attribute to split on
                  indexMax = listIG.index(max(listIG)) # Index of best attribute
                  attrMax = attrNames[indexMax]
                  attrNames.remove(attrMax) # Remove the best attribute from the attribute list

                  # Create an empty tree, soon to be populated
                  tree = {attrMax:{}} # Initiate the tree with the best attribute as a node

                  for attrType, dataSubset in dataset.groupby(attrMax):
                      subtree = ID3(dataSubset, classifier, attrNames)
                      tree[attrMax][attrType] = subtree
                  return tree
```

Figure 3.1: Code for the function of ID3, that takes input of the dataset, classifier for the target, a list of the attributes of the dataset.

Subsequent to creating the function of ID3, now we set the input to the function. The dataset is the tennis dataset imported from an Excel file. The classifier is 'PlayTennis', which is the column name that contains the target in the dataset. The attribute list is ['Outlook', 'Temperature', 'Humidity', 'Wind], which contains all the other column names in the dataset. Then we print the ID3 decision tree created from the ID3 function created as shown above, using *pprint* package which stands for pretty print to make the tree look more organized (see Figure 3.2).



```
In [221]: from pprint import pprint

          dfTennisData = pd.read_excel('Tennis data.xlsx')
          A = ['Outlook','Temperature','Humidity','Wind']

          treeID3 = ID3(dfTennisData, 'PlayTennis', A)
          pprint(treeID3)

          {'Outlook': {'Overcast': 'Yes',
                       'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},
                       'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}
```

Figure 3.2: Code of the result of the generated tree from the ID3 algorithm implemented by the code in Figure 3.1.