CITS5508 Lab Sheet 4:
Ensemble Learning and Random Forest

Name: James Mok
Student number: 21120365
Date created: April 17th 2019
Last modified: April 17th 2019

The aims of this lab can be broken down into the two projects:

Project 1: This project uses an Ensemble Classifier involving concepts such as data cleaning, regularisation, pipelines and more.
Project 2: This project uses a Random Forest regressor focusing on data cleaning, feature importance, dimensionality reduction and more.

Project 1

1.0 Classifying between Healthy vs Parkinson's Disease through using voice measurements.

The data set includes 195 voice recordings from individuals where particular voice measurements are the features. The aim of the data is to discriminate healthy people from those with Parkinsin's disease.

This dataset is a range of biomedical voice measurements from 31 people, 23 with Parkinson's disease.

1.1 Understanding the 'Parkinsons' Dataset

From `parkinsons.names` , the relevant details have been shown below:

Number of Attributes for parkinsons dataset: 23

It is also important to note that `Status` :
A value of 1 = Unhealthy (Parkinsons Disease)
A value of 0 = Healthy

Based on the the provided descriptions we can **remove** `name` since it is irrelevant to the prediction.

In [1]:

```
#Importing libraries and data files

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#Importing parkinsons data
data = pd.read_csv('parkinsons.data.txt')
data = data.drop(['name'], axis=1)

#Quick initial visualisation of the data
print('The dimensions of the data set is: \n'  + str(data.shape))
data.head(5)
```

```
The dimensions of the data set is:
(195, 23)
```

Out[1]:

	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)	MDVP:RAP
0	119.992	157.302	74.997	0.00784	0.00007	0.00370
1	122.400	148.650	113.819	0.00968	0.00008	0.00465
2	116.682	131.111	111.555	0.01050	0.00009	0.00544
3	116.676	137.871	111.366	0.00997	0.00009	0.00502
4	116.014	141.781	110.655	0.01284	0.00011	0.00655

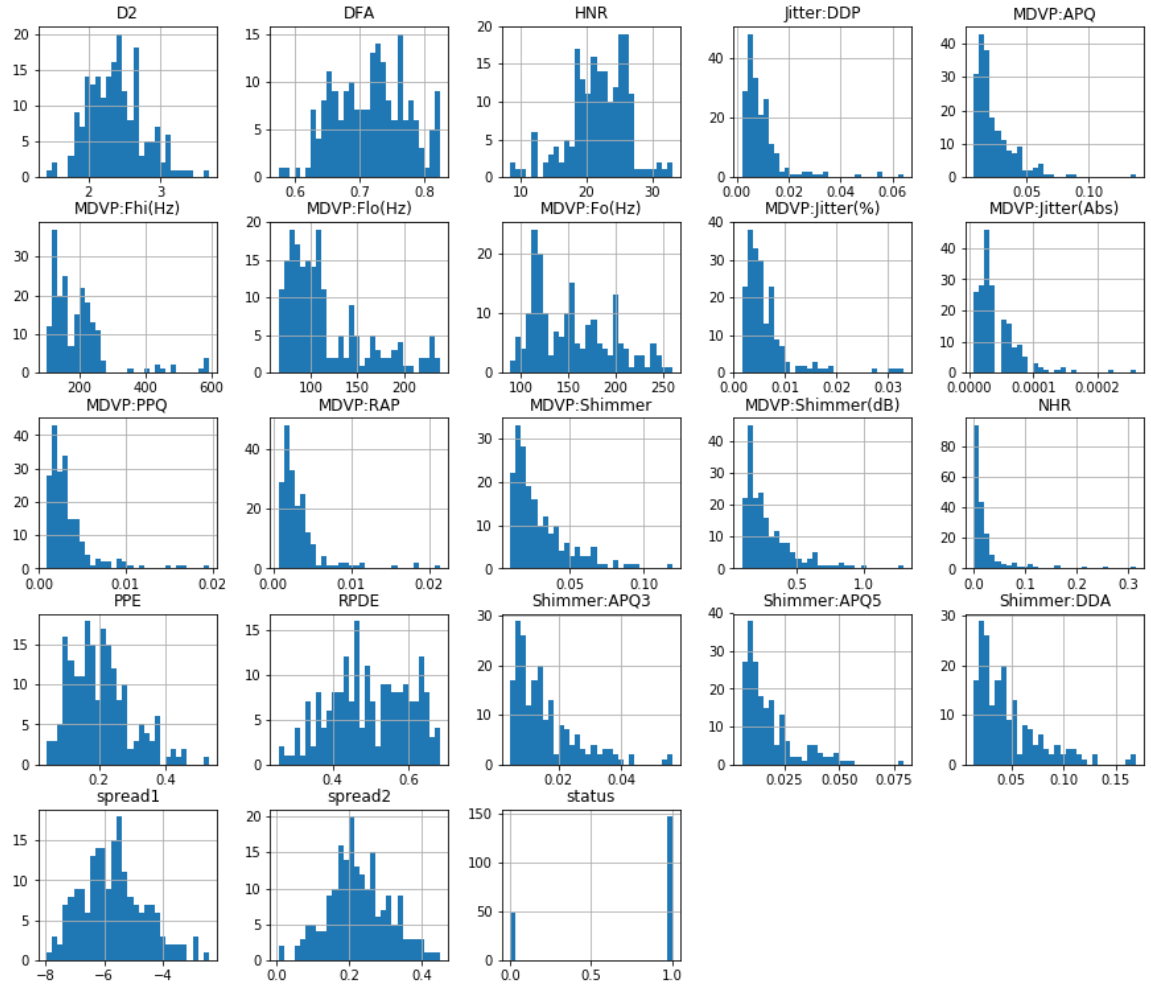
5 rows × 23 columns

1.2 Visualising data

First we will plot the data using histograms to identify the appropriate scaling technique

In [2]:

```
# Plotting histograms to visualise data
data.hist(bins=30, figsize=(15,13))
plt.show()
```



Comments on plots

Interesting to note that though there are some which follow a normal distribution, there are many features which are heavily skewed left. It will be very important to scale the data, especially since an SVM Classifier will be used.

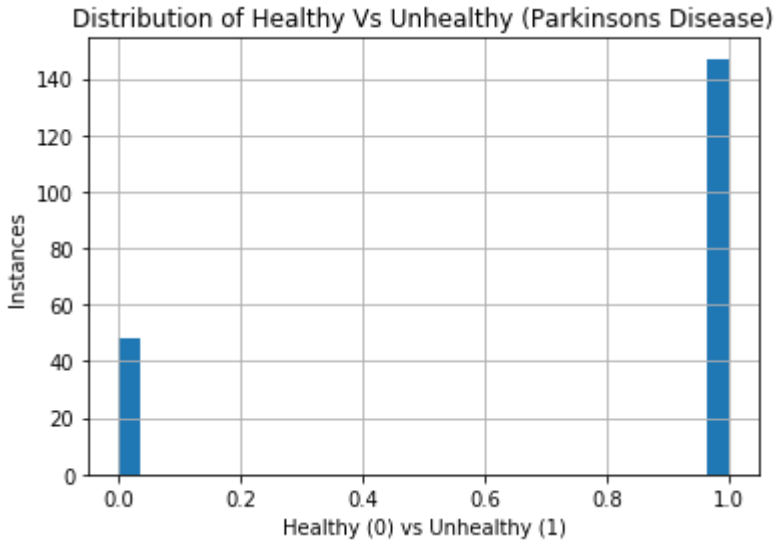
Check for imbalance

In [3]:

```
#Visualising ring value distribution
ax1 = data['status'].hist(bins = 29);

title = 'Distribution of Healthy Vs Unhealthy (Parkinsons Disease)'
ax1.set_title(title)
ax1.set_ylabel('Instances')
ax1.set_xlabel('Healthy (0) vs Unhealthy (1)')
```

Out[3]:

```
Text(0.5, 0, 'Healthy (0) vs Unhealthy (1)')
```



Comments on class distribution - imbalanced?

In this dataset, there is much more data on individuals that are unhealthy (1) compared to healthy (0). However it will be concluded that this dataset is not imbalanced as the difference is not significant enough.

Selecting the kernels

One type of visualisation that can be made is to plot the features against each other showing the healthy/unhealthy points on the graph. This allows us to see if the data is linearly seperable - which then allows us to determine the most suitable kernel.

However, an easy way to solve this is to use gridsearch which will find the most suitable kernel.

Another way to do this is using `seaborn.pairplot` , but have omitted this due to computation times and it is very difficult to interpret.

In [4]:

```
## Code is avaliable for interest
#import seaborn as sns
#sns.pairplot(data)
```

1.3 Splitting training and test set

In [5]:

```
from sklearn.model_selection import train_test_split

# Segregating data into X and y
y = data['status']
X = data.drop('status', axis = 1)

# Splitting the data set into training and test set, 80/20 split respectively
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2
0)
```

1.4 Building the Classifier

Below is an ensemble classifier comprising of an SVM and Logistic Regressor

Pipeline

A pipeline will be created to normalise, and classify the data with GridSearch

In [6]:

```
#importing libraries
from sklearn.model_selection import  GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import VotingClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix, f1_score

import warnings

# Ignore warnings
warnings.filterwarnings("ignore")

# Defining Scaler
scaler= StandardScaler()

# Defining the classifiers
LReg_clf = LogisticRegression(random_state=20)
SVC_clf = SVC()

# Defining pipeline using scaler and Voting Classifier
pipe1 = Pipeline([
    ('scale', scaler),
    ('VC', VotingClassifier([('LR', LReg_clf),('SVC', SVC_clf)])),
])

# Defining parameters
parameters = {
    'VC__LR__penalty':('l1', 'l2'),
    'VC__LR__C': [0.25, 0.5, 0.75],
    'VC__SVC__kernel':('rbf', 'linear'),
    'VC__SVC__C': [1, 1.5, 2]
}

# Gridsearch
VC_clf = GridSearchCV(pipe1, parameters, cv=3, scoring='f1')
VC_clf = VC_clf.fit(X_train,y_train)

# Scoring
print('Best score and parameters:')
print(VC_clf.best_score_)
print(VC_clf.best_params_)
```

```
Best score and parameters:
0.9146162479495812
{'VC__LR__C': 0.5, 'VC__LR__penalty': 'l1', 'VC__SVC__C': 1.5, 'VC__SVC__k
ernel': 'rbf'}
```

Explanation of Pipeline

Here the pipeline is used for:

- Scaling
- GridSearch (Best parameters)
- Classifying

Note: It is important to note that now using the pipeline we don't need to explicitly call fit_transform and transform from `StandardScaler`. This streamlines the process in writing the code.

Hyperparameters

The important hyperparameters to test here are:

- Kernel
- C
- Penalty

Kernel
As describe earlier, it is important to find the correct kernel to seperate the data. Here `rbf` was found to be most suitable

C
Helps to prevent overfitting of the training set through regularisation which is very important to tune correctly.

Penalty

- l1 = Lasso Regression
- l2 = Ridge Regression

The main difference is that Lasso reduces the features to 0 which ends up cutting off the irrelevant features. Ridge does not, which just reduces them. From `GridSearch` Lasso seems to be more suitable.

F1 score - Seperate Classifiers vs Voting Classifier

Here we will use the seperate classifiers to then provide a comparison

In [7]:

```
# Defining classifiers with Appropriate hyperparameters
pipe2 = Pipeline([
        ('scale', scaler),
        ('LR', LogisticRegression(penalty= 'l1', C = 0.5, random_state=20))
])

pipe3 = Pipeline([
        ('scale', scaler),
        ('SVC', SVC(kernel= 'rbf', C = 1.5))
])

# Fitting classifiers
LR_clf = pipe2.fit(X_train,y_train)
SVC_clf = pipe3.fit(X_train,y_train)

# Predictions
VC_pred = VC_clf.predict(X_test)
LR_pred = LR_clf.predict(X_test)
SVC_pred = SVC_clf.predict(X_test)

#Scoring
name = ['Voting Classifier','Logistic Regression', 'SVC']

predicted =[VC_pred, LR_pred, SVC_pred]

#for loop to generate scores using average=weighted method (used due to help relatively
imbalanced datasets)
for i in range(3):
    print(str(name[i]))
    print('F1 Score: '+ str("%.2f" % (f1_score(y_test, predicted[i], average='weighted'
)*100)) + '%' +'\n')
```

```
Voting Classifier
F1 Score: 81.24%

Logistic Regression
F1 Score: 81.24%

SVC
F1 Score: 86.04%
```

Comments on results

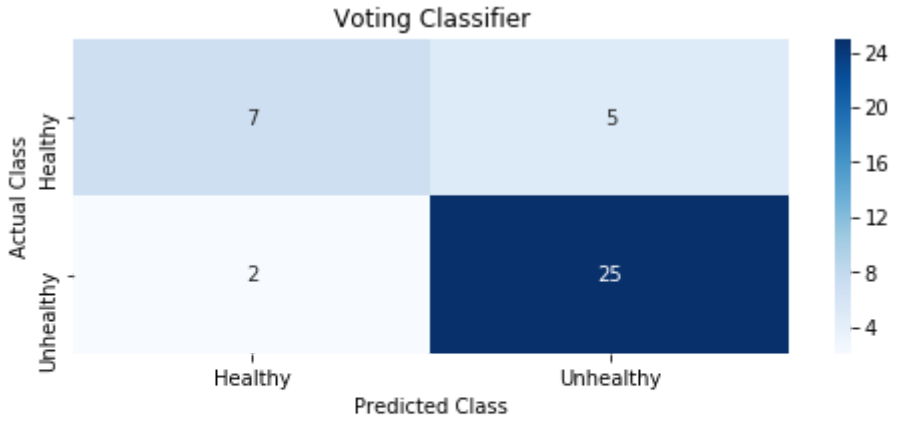
Firstly, it is interesting to note that the F1 Score of Voting Classifier and Logistic Regression are the exact same. Secondly, that the Voting Classifier performed worse. This is interesting as ensemble learning tries to find the best of both classifiers, however instances like this situation do occur.

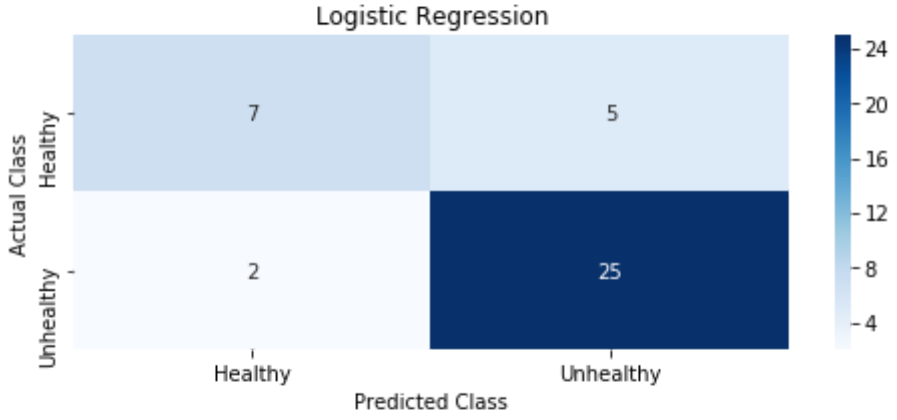
Confusion matricies

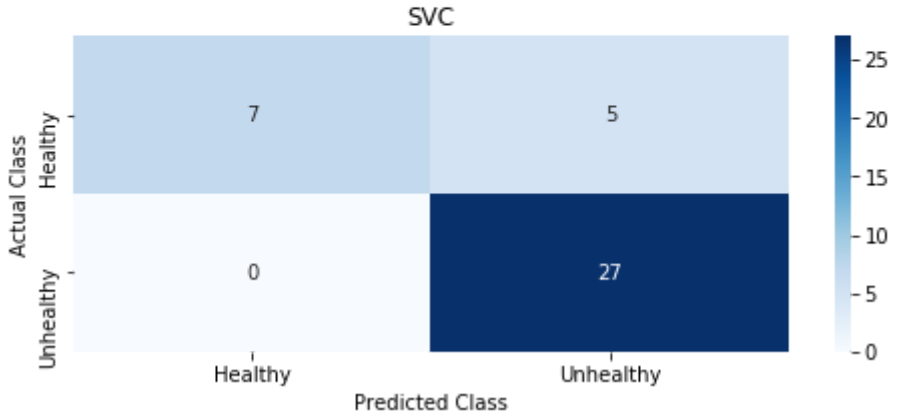
Below are the confusion matricies

In [8]:

```
# Creating Confusion Matricies

#Importing Lirbaries
import seaborn as sns

#Confusion matrix
cm_VC = confusion_matrix(y_test, VC_pred)
cm_LR = confusion_matrix(y_test, LR_pred)
cm_SVC = confusion_matrix(y_test, SVC_pred)

#Defining labels to go into confusion matrix
cm = [cm_VC,cm_LR,cm_SVC]
title = ['Voting Classifier','Logistic Regression',
         'SVC']

#Creating figure
fig = plt.figure(figsize=(8,11))
fig.subplots_adjust(hspace = 0.4, wspace=0.3)

#Defining labels
labels = ['Healthy','Unhealthy']

#For loop to cycle through and create the confusion matrices
for i in range(3):

    #Creating confusion matrix
    ax = fig.add_subplot(3, 1, i+1)
    sns.heatmap(cm[i], annot = True, xticklabels= labels, yticklabels=labels, cmap='Blu
es')
    #Plot labels
    plt.title(str(title[i]))
    plt.xlabel('Predicted Class');
    plt.ylabel('Actual Class');
```


Voting Classifier
7
5
2
25
Healthy
Unhealthy
Actual Class
Healthy
Unhealthy
Predicted Class
24
20
16
12
8
4


Logistic Regression
7
5
2
25
Healthy
Unhealthy
Actual Class
Healthy
Unhealthy
Predicted Class
24
20
16
12
8
4


SVC
7
5
0
27
Healthy
Unhealthy
Actual Class
Healthy
Unhealthy
Predicted Class
25
20
15
10
5
0

Observations - Confusion matricies

One interesting observation is that the SVC classifier did not have any false positives which the other two did. Also, correct Unhealthy predictions was the highest. This may be influenced by the data size with favourably more unhealthy data.

Again it is interesting how the Voting Classifier has the same results as Logistic Regression where the voting classifier performed worse.

1.5 Conclusion

F1 Scores:

- Voting Classifier - 81.24%
- Logistic Regression - 81.24%
- SVC - 86.04%

The results show that the Support Vector Machine Classifier performed the best by ~5% and also from the confusion matrix less prone to false positives.

It is an interesting result as voting classifiers usually combines the best of both models. However in this situation the SVC performed best.

Project 2

2.0 Predicting age of abalone using physical measurements.

The project aims to predict the age of abalone from physical measurements. This dataset contains 8 different attributes and 29 different ring values.

2.1 Understanding the 'abalone' Dataset

From `abalone.names` , the relevant details have been shown below:

Number of attributes: 8 Number of ring values: 29

> It is important to note that age is +1.5 years added to ring values

The **headings** for the columns were also added based on `abalone.names`

In [9]:

```
#Importing libraries and data files

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#Importing abalone data
data = pd.read_csv('abalone.data.txt')
data.columns = ['Sex', 'Length', 'Diameter', 'Height',
                'Whole weight', 'Shucked weight', 'Viscera weight',
                'Shell weight', 'Rings']

#Quick initial visualisation of the data
print('The dimensions of the data set is: \n'  + str(data.shape))
data.head(5)
```

```
The dimensions of the data set is:
(4176, 9)
```

Out[9]:

	Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
0	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
1	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
2	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
3	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7
4	I	0.425	0.300	0.095	0.3515	0.1410	0.0775	0.120	8

2.2 Encoding

As we can see from the data, the `'Sex'` column contains string values. This needs to be encoded into numerical values to be used in the model.

One-Hot Encoding

One Hot Encoding is used where the categorical data have no ordinal relationship between each other. To do this, One Hot Encoding creates new columns for each category and assigns a 1 to the class and 0 to the other classes.

For this reason, we will use One-Hot Encoding

In [10]:

```
from sklearn.preprocessing import OneHotEncoder

#Defining encoder
enc = OneHotEncoder(handle_unknown='ignore')

#Encoding
S = enc.fit_transform(data['Sex'].values.reshape(-1,1)).toarray()

#Checking order of categories to label
print(enc.categories_)
OneHot = pd.DataFrame(S, columns = ['Sex F','Sex I','Sex M'])

#Adding new columns that are one-hot encoded
data_enc = pd.concat([data, OneHot], axis=1)
data_enc = data_enc.drop(['Sex'], axis=1)

#Visualisation
data_enc.head()
```

```
[array(['F', 'I', 'M'], dtype=object)]
```

Out[10]:

	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings	Sex F	Sex I	Sex M
0	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7	0.0	0.0	1.0
1	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9	1.0	0.0	0.0
2	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10	0.0	0.0	1.0
3	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7	0.0	1.0	0.0
4	0.425	0.300	0.095	0.3515	0.1410	0.0775	0.120	8	0.0	1.0	0.0

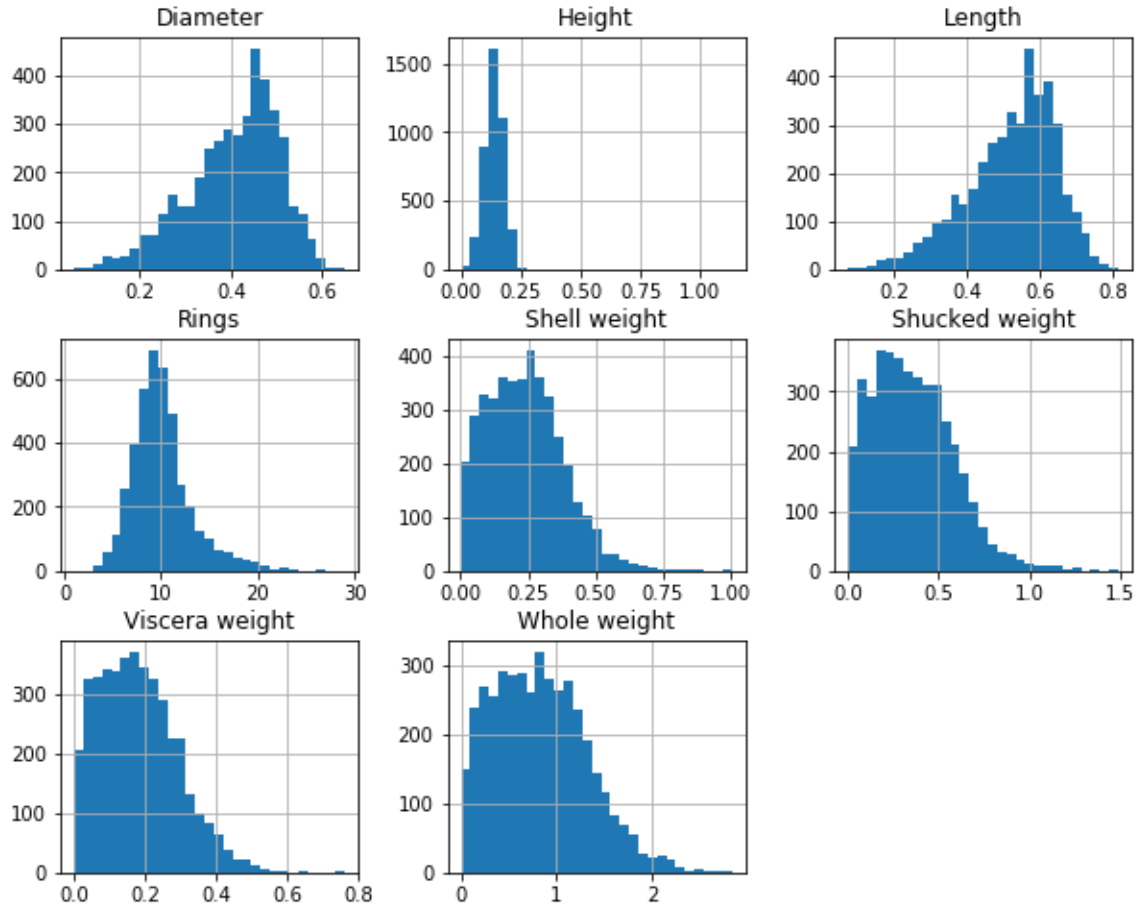
It is important to note that encoding must come before scaling to ensure that the new columns don't adversely affect the model.

2.3 Visualising data

Below are histograms plotting to further understanding the distribution of the data

In [11]:

```
# Plotting historrams to visualise data
data.hist(bins=29, figsize=(10,8))
plt.show()
```



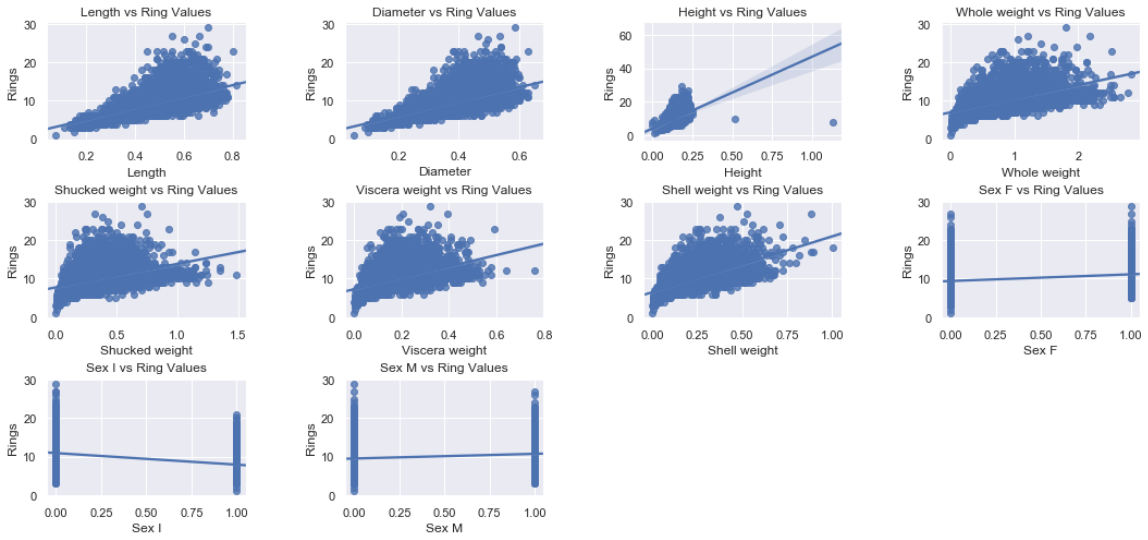
These look normally distributed, therefore `StandardScaler` will be used

Visualisation - Regression of each feature vs Ring Values

Below, plots will be created to understand the relationship between the Ring Values and each feature seperately.

In [12]:

```
# Plotting the features vs Ring Values

#Using data with encoded features
data = data_enc

#Importing visualisation tools
import seaborn as sns; sns.set(color_codes=True)
tips = sns.load_dataset("tips")

#Plotting each feature vs Ring Values
feat = ['Length', 'Diameter','Height', 'Whole weight',
        'Shucked weight' ,'Viscera weight' ,'Shell weight',
       'Sex F', 'Sex I', 'Sex M']
#Creating figures
fig = plt.figure(figsize=(18,8))
fig.subplots_adjust(hspace = 0.5, wspace=0.5)

#for loop to create plots
for i in range(len(feat)):
    #using replot to create scatter plot with line of best fit
    f = fig.add_subplot(3,4,i+1)
    ax = sns.regplot(x = data[feat[i]], y = data['Rings'])
    plt.title(feat[i] + ' vs Ring Values')
```



Observations

In this case, it can be seen that most features exhibit something similar to a linear relationship to Ring Values. However, it is evident that it is very hard to determine a relationship for `Sex`.

The `Sex I` plot which is the feature for infants, does however show a bit more of a relationship with a stronger slope. This plot shows that if it is an infant (1) the Rings Values are lower (Lower age) - and this makes sense.

2.4 Distribution of the labelled data

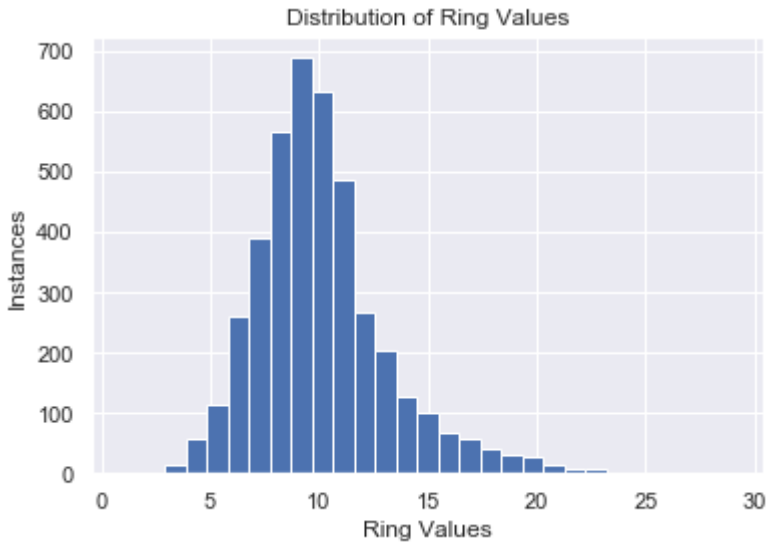
Below is a plot to visualise the labelled data (y set) to check for for imbalance of data.

In [13]:

```
#Visualising ring value distribution
ax1 = data['Rings'].hist(bins = 29);

title = 'Distribution of Ring Values'
ax1.set_title(title)
ax1.set_ylabel('Instances')
ax1.set_xlabel('Ring Values')
```

Out[13]:

```
Text(0.5, 0, 'Ring Values')
```



Observations

Very imbalanced set, especially near the ends of the plot.

We will now look at this in further detail.

In [14]:

```
# Using the new data after cncoding (no difference for now)
data = data_enc

# Display ring values and instances in descending order
data['Rings'].value_counts()
```

Out[14]:

```
9     689
10    634
8     568
11    487
7     391
12    267
6     259
13    203
14    126
5     115
15    102
16     67
17     58
4      57
18     42
19     32
20     26
3      15
21     14
23      9
22      6
24      2
27      2
1       1
25      1
2       1
26      1
29      1
Name: Rings, dtype: int64
```

Comments - Grouping Ring Values with Low Instances

I have to decided to group the Ring Values with low instances instead of removing. I think it is important to still include this data.

But how?

> I will group >=21 as well as <=4.

- This will retain the data of the ring values that are very high/low since we aren't cutting them out.

However since the problem is regression, >=21 and <=4 will become a average numerical value for the purposes of the model. If it is not an average, the model can not intepret the number correctly.

In [15]:

```
#Grouping >=21 into an average
high_group = ((21*14)+(22*6)+(23*9)+(24*2)+25+26+(27*2)+29)/(14+6+9+2+2+2+1)
print('Grouping of greater than or equal to 21: ' + str('%.2f' % high_group))

#Grouping <= 4 into an average
low_group = (1+2+(3*15)+(4*57))/(1+1+15+57)
print('Grouping of less than or equal to 4: ' + str('%.2f' % low_group))
```

```
Grouping of greater than or equal to 21: 22.64
Grouping of less than or equal to 4: 3.73
```

In [16]:

```
# for loop to group values less than or equal to 4 as '3.73'
#                          greater than or equal to 21 as '22.64'

for index in range(len(data)):
    if data['Rings'].loc[index] >= 21:
        data['Rings'].loc[index] = 22.64
    elif data['Rings'].loc[index] <= 4:
        data['Rings'].loc[index] = 3.73
```

In [17]:

```
# New Ring Value Distribution
data['Rings'].value_counts()
```

Out[17]:

```
9.00     689
10.00    634
8.00     568
11.00    487
7.00     391
12.00    267
6.00     259
13.00    203
14.00    126
5.00     115
15.00    102
3.73      74
16.00     67
17.00     58
18.00     42
22.64     36
19.00     32
20.00     26
Name: Rings, dtype: int64
```

2.5 Splitting train and test set

In [18]:

```
#importing library
from sklearn.model_selection import train_test_split

# Segregating data into X and y
y = data['Rings']
X = data.drop('Rings', axis = 1)

# Splitting the data set into training and test set, 90/10 split respectively
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=2
0)
```

2.6 Normalisation of data

As mentioned earlier, we will be using `StandardScaler`

In [19]:

```
from sklearn.preprocessing import StandardScaler
#Defining scaler
scaler = StandardScaler()

# Scaled training and test sets
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

2.7 Random Forest Regressor

Below is the Random Forest Regressor where we will compute the feature importance and RMSE

In [20]:

```
#Importing libraries
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from math import sqrt
import warnings
# Ignore warnings
warnings.filterwarnings("ignore")

# Defining the random forest regressor
Reg = RandomForestRegressor(max_depth=5, min_samples_split=5, min_samples_leaf = 30, ra
ndom_state=20)

# fitting, predicting the model + listing feature importance
Reg = Reg.fit(X_train, y_train)
y_pred = Reg.predict(X_test)
print('Feature Importance:'+ str(Reg.feature_importances_))

#scoring using RMSE
RMSE = sqrt(mean_squared_error(y_test, y_pred))
print('Random Forest RMSE = ' + str(RMSE))
```

```
Feature Importance:[0.00732359 0.00360365 0.00732082 0.01627673 0.14349496
0.00368158
 0.79639661 0.         0.02069205 0.00121001]
Random Forest RMSE = 2.171077062350583
```

Hyperparameters

- max_depth - Depth of each tree in the forest. The deeper the tree captures more information of the dataset.
- min_samples_split - Minimum number of samples required to split an internal node.
- min_samples_leaf - Minimum number of samples required to be at a leaf node, the base of the tree.

I have avoided implementing GridSearch which will increase computation times and have found the previously used values optimal.

2.8 Dimensionailty Reduction - Feature importance

Now we will remove features that are below 95% total feature importance.

This removal of features were indexed based off the feature importance output earlier, where all features were removed except for `Shucked weight` and `Shell weight` .

In [21]:

```
# Removing features that do not contribute to 95% of total feature importance
X_train_new = np.delete(X_train,(0,1,2,3,5,7,8,9),axis=1)
X_test_new = np.delete(X_test,(0,1,2,3,5,7,8,9),axis=1)

print(str(X_train_new.shape))
print('There are 2 features left')
```

```
(3758, 2)
There are 2 features left
```

Note: It is importance to realise that index 7,8,9 relates to the `Sex` feature from one hot encoding therefore all these features will be removed together.

Comparison of RMSE

Here we computer the model using the data **with** dimensionailty reduction.

A comparison of the two differences of RMSE and Absolute error will be shown below.

In [22]:

```
Reg2 = Reg.fit(X_train_new, y_train)
y_pred2 = Reg.predict(X_test_new)
print('Feature Importance:'+ str(Reg2.feature_importances_))

RMS = sqrt(mean_squared_error(y_test, y_pred2))

print('Random Forest RMSE (Reduced dimensions) = ' + str(RMS))
```

```
Feature Importance:[0.15522942 0.84477058]
Random Forest RMSE (Reduced dimensions) = 2.209247014276737
```

RMSE Comparison - Results

> Random Forest RMSE (Before reduction) = 2.17
> Random Forest RMSE (Reduced features) = 2.21

As expected `RMSE` was higher **after** dimensionality reduction.

This means that removing most of the features provided worse results. This would makes sense as a lot of data is removed and less for the model to intepret. However, this only increased the RMSE by a very small amount which drastically **improves computation times**.

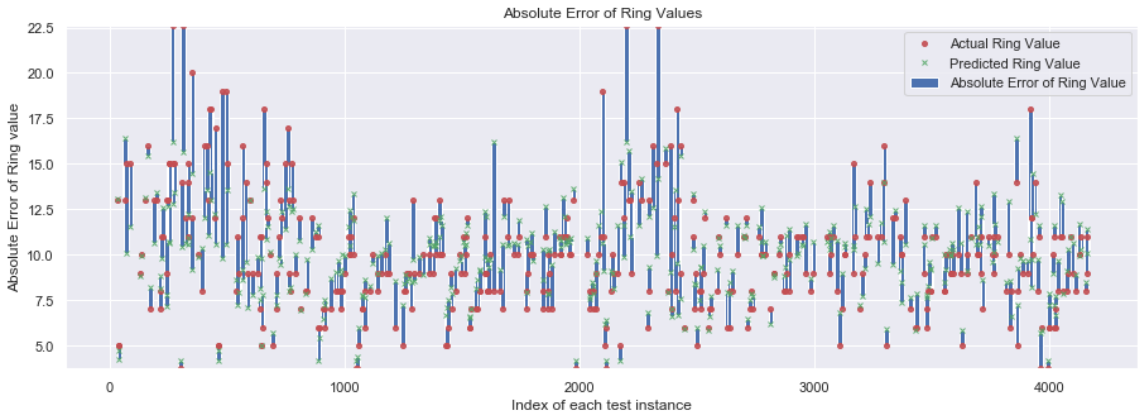
Therefore, in this case, **dimensionailty reduction** has **decreased** the models results but improves computation times.

Plot of Absolute Error of Ring Values

Below is a plot of the absolute error of Ring Values after dimensionality reduction

In [23]:

```
import matplotlib.pyplot as plt

# Calculating error to be used in addition to the baseline of actual ring values
error = y_pred2 - y_test

#Creating figure
fig = plt.figure(figsize=(35,5))
fig.subplots_adjust(hspace = 0.5, wspace=0.3)
ax = fig.add_subplot(1, 2, 1)

#Bar plot - Absolute Error - p1 = abs error, p2 =  Actual Ring values, p3 = Predicted R
ing Values
p1 = plt.bar(y_test.index.values, error,bottom = y_test.values , width = 20)
p2 = plt.plot(y_test.index.values, y_test.values,
              marker="o",linestyle="", alpha=0.9,markersize=4, color="r")
p3 = plt.plot(y_test.index.values, y_test + error,
              marker="x", linestyle="", alpha=0.9,markersize=4, color="g")

#Defining labels
plt.ylabel('Absolute Error of Ring value')
plt.xlabel('Index of each test instance')
plt.title('Absolute Error of Ring Values')
plt.legend(('Actual Ring Value','Predicted Ring Value','Absolute Error of Ring Value')
           ,loc='upper right')
plt.show()
```



Observations of plot

It is interesting to see that the model has lower amounts of error for predicting Ring Values that are low (Ring values of 6-10). The reason for this is simply that there is more data in that range which helps the predictions. Also, Vice-versa, the model does not predict as well in the higher range of Ring Values.

Dimensionality Reduction - Principal Component Analysis (PCA)

PCA will be used and compared with the two RMSE resuts above.

In [24]:

```
from sklearn.decomposition import PCA

# PCA to retain 99% of variance
pca = PCA(n_components=0.99)
pca.fit(X_train)

# Transforming training and test set
X_train_new2 = pca.transform(X_train)
X_test_new2 = pca.transform(X_test)

#Printing to show 99% variance retained, and 6/10 componenets retained.
print('Cumulative sum of variance retention:' + str(pca.explained_variance_ratio_.cumsu
m()))
```

```
Cumulative sum of variance retention:[0.69242931 0.84486526 0.93543352 0.9
6347746 0.98015549 0.99161368]
```

PCA - Computing for RMSE

Here the data after PCA will be computerd for RMSE

In [25]:

```
# Defining regressor
Reg = RandomForestRegressor(max_depth=5, min_samples_split=5, min_samples_leaf = 30, ra
ndom_state=20)

#Fitting with PCA used on data set + feature importance
Reg = Reg.fit(X_train_new2, y_train)
y_pred3 = Reg.predict(X_test_new2)
print('Feature Importance:'+ str(Reg.feature_importances_))


RMSE = sqrt(mean_squared_error(y_test, y_pred3))
print('Random Forest RMSE (After PCA) = ' + str(RMSE))
```

```
Feature Importance:[0.58025328 0.00937608 0.00597063 0.03897606 0.02752953
0.33789442]
Random Forest RMSE (After PCA) = 2.2175086017460055
```

2.9 Conclusions

Conclusions about the two ways of performing dimensionality reduction

> Random Forest RMSE (Before reduction) = 2.17
> Random Forest RMSE (Reduced features) = 2.21
> Random Forest RMSE (After PCA) = 2.22

PCA performed slightly worse, and in this case PCA kept 6 features compared to 2 which is very interesting as more data suggest better results. Though slightly worse, it was very simliar to using 95% of feature importance.

In this case, using feature importance performs better than PCA.

3.0 Summary

This lab was another great experience but also provided many challenges!

My key takeaways were:

- Intuition behind dimensionality reduction
- Practically using Pipelines
- Implementing Ensemble learning
- Visualation of data - improving plotting methods

Thank you for reading!