In [1]:

```
from IPython import display
display.Image("/content/gdrive/MyDrive/JohannesStotter-660x390.jpg")
```

Out[1]:

Logistic Regration Model : LR workes really well on Binary classification problem. This is a supervised learning algorithm.

Work Flow For this Project.

1. Data Pre-Processing
2. Train Test split
3. Train our model

Note :- Objective is to find "Given object is Rock(R) or Mine(M).

Following steps:

1. Import libraries and Load dataset
2. Analyze & Visualize Data(Descriptive Statistics)
3. Validation Dataset
4. Evaluate Algorithms
5. Algorithm Tuning
6. Ensemble Methods
7. Finalize Model
8. Summary

1. Importing libraries

In [2]:

```
import pandas as pd
import numpy as np
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content
/gdrive", force_remount=True).
```

In [3]:

```
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
```

2. Data Collection and Data Processing

In [4]:

```
df= pd.read_csv('/content/gdrive/MyDrive/ML_projects_data/sonar.all-data.csv', header=None)
```

In [5]:

```
df.head()
```

Out[5]:

	0	1	2	3	4	5	6	7	8	9	...	51	52	53	54	55	56	
0	0.0200	0.0371	0.0428	0.0207	0.0954	0.0986	0.1539	0.1601	0.3109	0.2111	...	0.0027	0.0065	0.0159	0.0072	0.0167	0.0180	0.0
1	0.0453	0.0523	0.0843	0.0689	0.1183	0.2583	0.2156	0.3481	0.3337	0.2872	...	0.0084	0.0089	0.0048	0.0094	0.0191	0.0140	0.0
2	0.0262	0.0582	0.1099	0.1083	0.0974	0.2280	0.2431	0.3771	0.5598	0.6194	...	0.0232	0.0166	0.0095	0.0180	0.0244	0.0316	0.0
3	0.0100	0.0171	0.0623	0.0205	0.0205	0.0368	0.1098	0.1276	0.0598	0.1264	...	0.0121	0.0036	0.0150	0.0085	0.0073	0.0050	0.0
4	0.0762	0.0666	0.0481	0.0394	0.0590	0.0649	0.1209	0.2467	0.3564	0.4459	...	0.0031	0.0054	0.0105	0.0110	0.0015	0.0072	0.0

5 rows × 61 columns

In [6]:

```
# Number of rows and columns
df.shape
```

Out[6]:

```
(208, 61)
```

In [7]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 208 entries, 0 to 207
Data columns (total 61 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       208 non-null    float64
 1   1       208 non-null    float64
 2   2       208 non-null    float64
 3   3       208 non-null    float64
 4   4       208 non-null    float64
 5   5       208 non-null    float64
 6   6       208 non-null    float64
 7   7       208 non-null    float64
 8   8       208 non-null    float64
 9   9       208 non-null    float64
 10  10      208 non-null    float64
 11  11      208 non-null    float64
 12  12      208 non-null    float64
 13  13      208 non-null    float64
 14  14      208 non-null    float64
 15  15      208 non-null    float64
 16  16      208 non-null    float64
 17  17      208 non-null    float64
 18  18      208 non-null    float64
 19  19      208 non-null    float64
 20  20      208 non-null    float64
 21  21      208 non-null    float64
 22  22      208 non-null    float64
 23  23      208 non-null    float64
 24  24      208 non-null    float64
 25  25      208 non-null    float64
 26  26      208 non-null    float64
 27  27      208 non-null    float64
 28  28      208 non-null    float64
 29  29      208 non-null    float64
 30  30      208 non-null    float64
 31  31      208 non-null    float64
 32  32      208 non-null    float64
 33  33      208 non-null    float64
 34  34      208 non-null    float64
 35  35      208 non-null    float64
 36  36      208 non-null    float64
 37  37      208 non-null    float64
 38  38      208 non-null    float64
 39  39      208 non-null    float64
 40  40      208 non-null    float64
 41  41      208 non-null    float64
 42  42      208 non-null    float64
 43  43      208 non-null    float64
 44  44      208 non-null    float64
 45  45      208 non-null    float64
 46  46      208 non-null    float64
 47  47      208 non-null    float64
 48  48      208 non-null    float64
 49  49      208 non-null    float64
 50  50      208 non-null    float64
 51  51      208 non-null    float64
 52  52      208 non-null    float64
 53  53      208 non-null    float64
 54  54      208 non-null    float64
 55  55      208 non-null    float64
 56  56      208 non-null    float64
 57  57      208 non-null    float64
 58  58      208 non-null    float64
 59  59      208 non-null    float64
 60  60      208 non-null    object 
dtypes: float64(60), object(1)
memory usage: 99.2+ KB
```

In [8]:

```
# Staticcals Measures of the data.
df.describe()
```

Out[8]:

	0	1	2	3	4	5	6	7	8	9	...	
count	208.000000	208.000000	208.000000	208.000000	208.000000	208.000000	208.000000	208.000000	208.000000	208.000000	...	208.(
mean	0.029164	0.038437	0.043832	0.053892	0.075202	0.104570	0.121747	0.134799	0.178003	0.208259	...	0.(
std	0.022991	0.032960	0.038428	0.046528	0.055552	0.059105	0.061788	0.085152	0.118387	0.134416	...	0.(
min	0.001500	0.000600	0.001500	0.005800	0.006700	0.010200	0.003300	0.005500	0.007500	0.011300	...	0.(
25%	0.013350	0.016450	0.018950	0.024375	0.038050	0.067025	0.080900	0.080425	0.097025	0.111275	...	0.(
50%	0.022800	0.030800	0.034300	0.044050	0.062500	0.092150	0.106950	0.112100	0.152250	0.182400	...	0.(
75%	0.035550	0.047950	0.057950	0.064500	0.100275	0.134125	0.154000	0.169600	0.233425	0.268700	...	0.(
max	0.137100	0.233900	0.305900	0.426400	0.401000	0.382300	0.372900	0.459000	0.682800	0.710600	...	0.

8 rows × 60 columns

In [9]:

```
df.value_counts().sum()
```

Out[9]:

```
208
```

In [10]:

```
# objects are reasonably balanced between M (mines) and R (rocks).
df[60].value_counts()
```

Out[10]:

```
M    111
R     97
Name: 60, dtype: int64
```

In [11]:

```
# Mean values for all the 60 columns; diffrences between Rocks and Mine value determine the object is "whether mi
ne or rock".
df.groupby(60).mean()
```

Out[11]:

	0	1	2	3	4	5	6	7	8	9	...	50	51	5
60														
M	0.034989	0.045544	0.050720	0.064768	0.086715	0.111864	0.128359	0.149832	0.213492	0.251022	...	0.019352	0.016014	0.01164
R	0.022498	0.030303	0.035951	0.041447	0.062028	0.096224	0.114180	0.117596	0.137392	0.159325	...	0.012311	0.010453	0.00964

2 rows × 60 columns

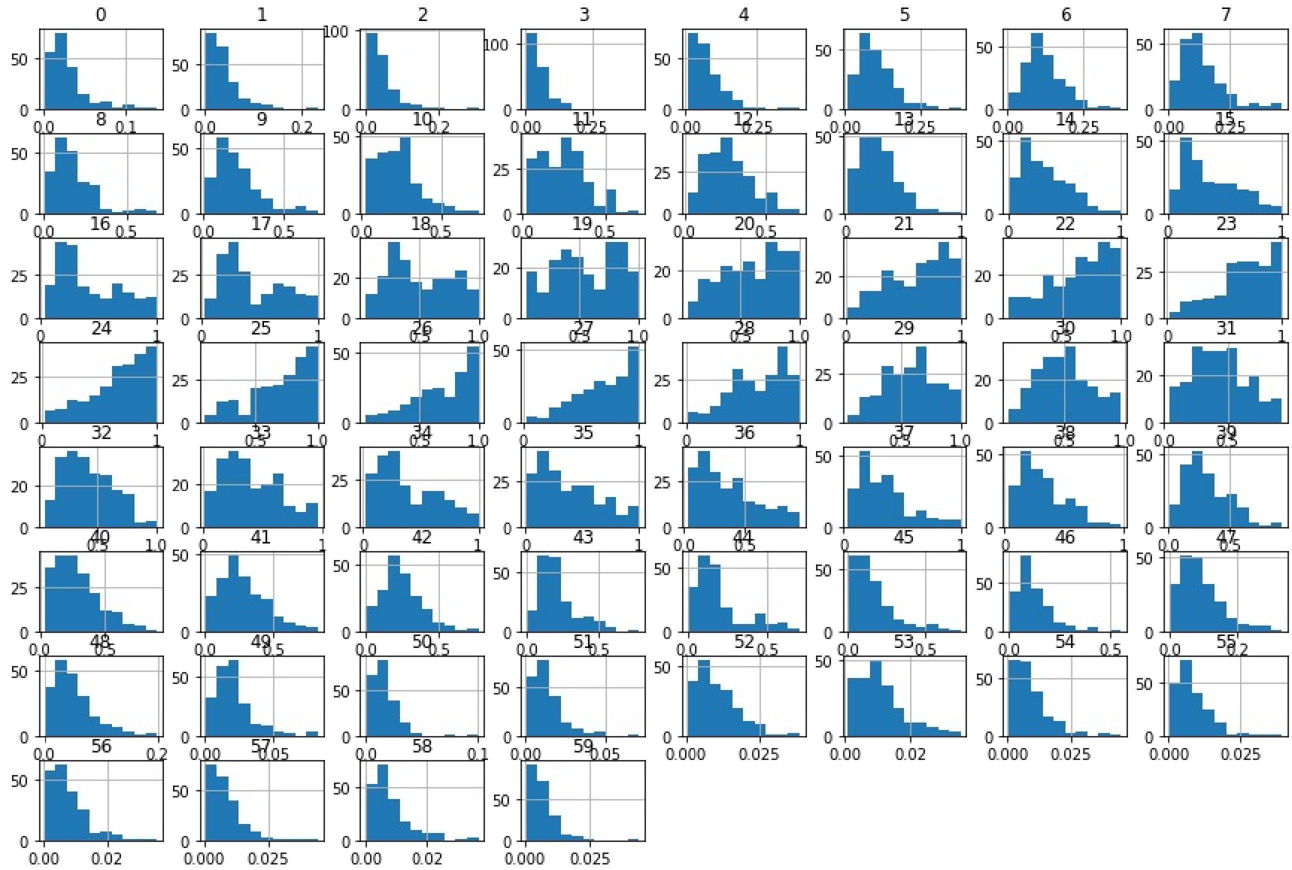
3. Visualizations

In [12]:

```
# Let's look at visualizations of individual attributes
```

In [13]:

```
df.hist(figsize=(15,10))
plt.show()
```



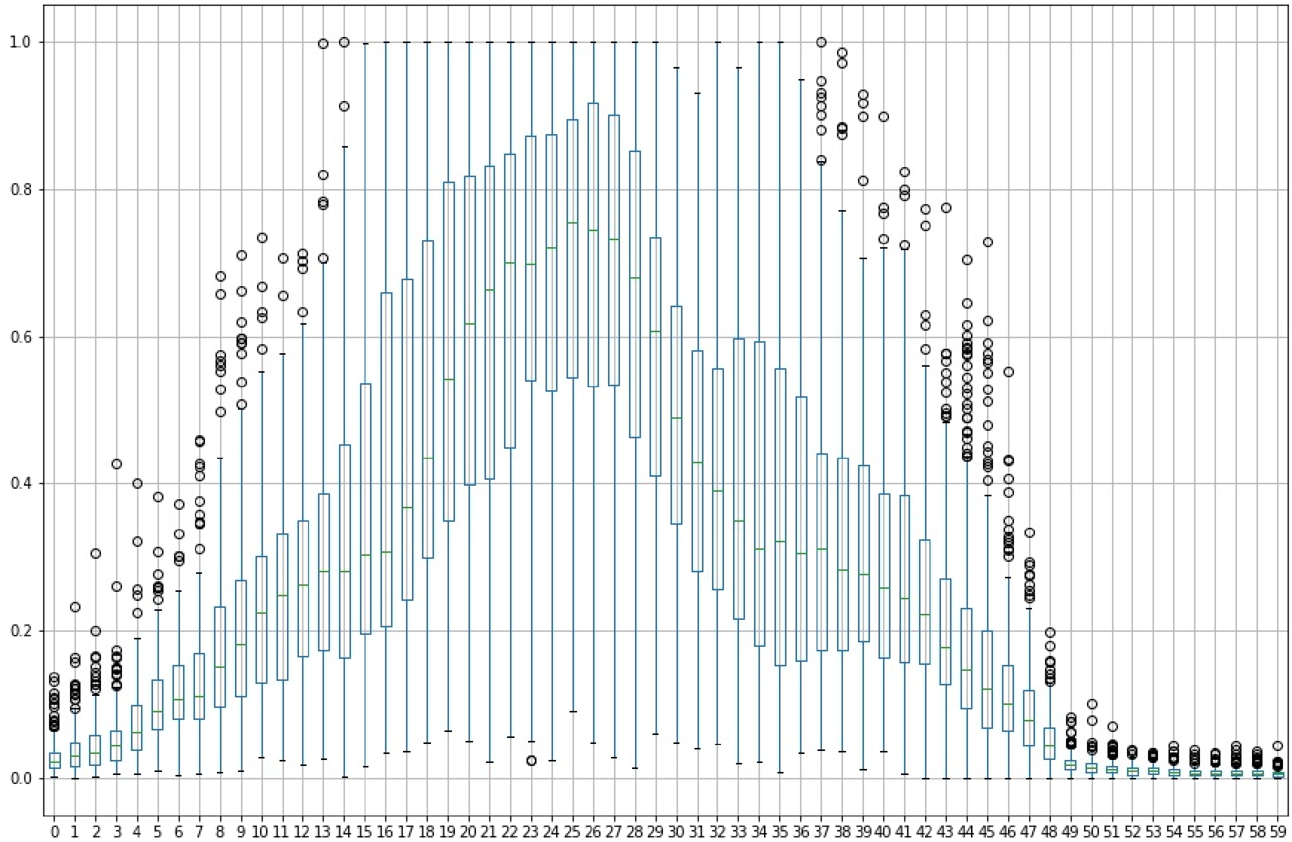
- we can see that there are a lot of Gaussian-like distributions.

In [14]:

```
# Box and whisker plots
df.boxplot(figsize=(15,10))
plt.show
```

Out[14]:

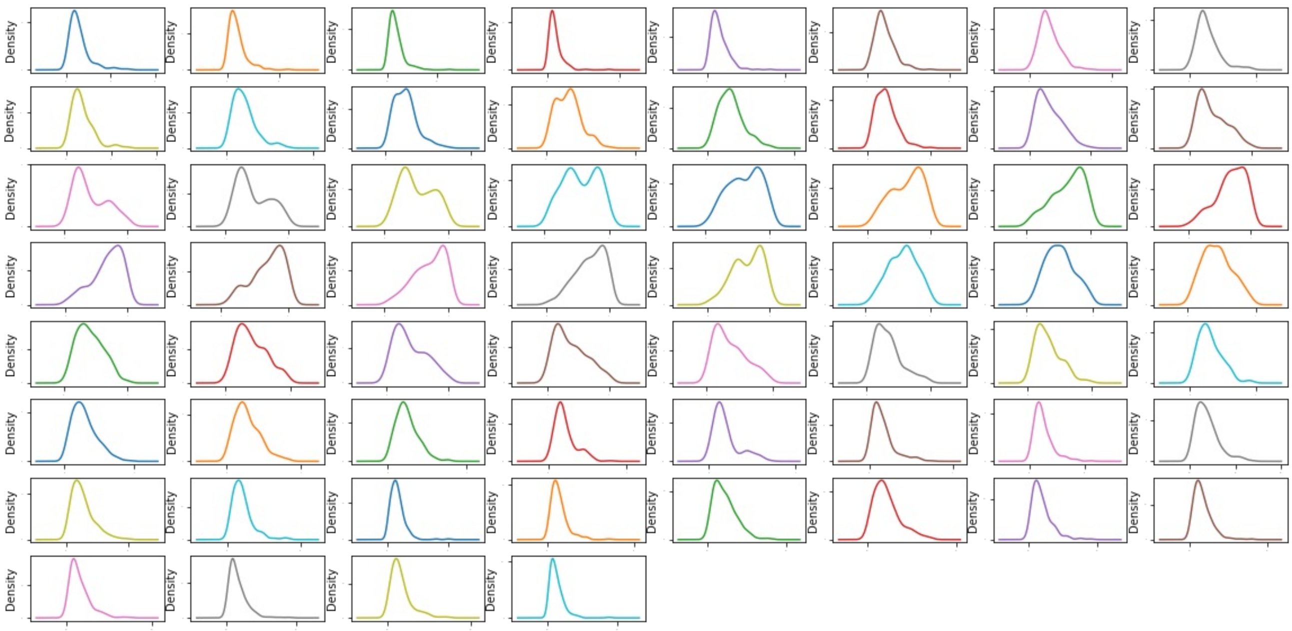
```
<function matplotlib.pyplot.show(*args, **kw)>
```



- We can see that attributes do have quite different spreads.

In [15]:

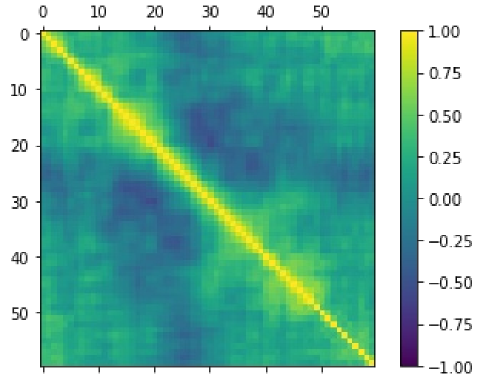
```
# density
df.plot(kind='density', subplots=True, layout=(8,8), sharex=False, legend=False, fontsize=1,figsize=(20,10))

plt.show()
```



- We can see that many of the attributes have a skewed distribution.
- A power transform like a Box-Cox transform that can correct for the skew in distributions might be useful.

In [16]:

```
# correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(df.corr(), vmin=-1, vmax=1, interpolation='none')
fig.colorbar(cax)
plt.show()
```



- The yellow around the diagonal suggests that attributes that are next to each other are generally more correlated with each other.
- The green & dark green patches also suggest some moderate negative correlation from each other in the ordering.

4. Data Validation

In [17]:

```
# seprating data and Labels
X=df.drop(columns=60,axis=1)
Y=df[60]
validation_size =0.20
```

In [18]:

```
print (X)
print(Y)
```

```
          0       1       2       3       4       5       6       7       8   \
0    0.0200  0.0371  0.0428  0.0207  0.0954  0.0986  0.1539  0.1601  0.3109
1    0.0453  0.0523  0.0843  0.0689  0.1183  0.2583  0.2156  0.3481  0.3337
2    0.0262  0.0582  0.1099  0.1083  0.0974  0.2280  0.2431  0.3771  0.5598
3    0.0100  0.0171  0.0623  0.0205  0.0205  0.0368  0.1098  0.1276  0.0598
4    0.0762  0.0666  0.0481  0.0394  0.0590  0.0649  0.1209  0.2467  0.3564
..      ...     ...     ...     ...     ...     ...     ...     ...     ...
203  0.0187  0.0346  0.0168  0.0177  0.0393  0.1630  0.2028  0.1694  0.2328
204  0.0323  0.0101  0.0298  0.0564  0.0760  0.0958  0.0990  0.1018  0.1030
205  0.0522  0.0437  0.0180  0.0292  0.0351  0.1171  0.1257  0.1178  0.1258
206  0.0303  0.0353  0.0490  0.0608  0.0167  0.1354  0.1465  0.1123  0.1945
207  0.0260  0.0363  0.0136  0.0272  0.0214  0.0338  0.0655  0.1400  0.1843

          9   ...      50      51      52      53      54      55      56  \
0    0.2111  ...  0.0232  0.0027  0.0065  0.0159  0.0072  0.0167  0.0180
1    0.2872  ...  0.0125  0.0084  0.0089  0.0048  0.0094  0.0191  0.0140
2    0.6194  ...  0.0033  0.0232  0.0166  0.0095  0.0180  0.0244  0.0316
3    0.1264  ...  0.0241  0.0121  0.0036  0.0150  0.0085  0.0073  0.0050
4    0.4459  ...  0.0156  0.0031  0.0054  0.0105  0.0110  0.0015  0.0072
..      ...  ...     ...     ...     ...     ...     ...     ...     ...
203  0.2684  ...  0.0203  0.0116  0.0098  0.0199  0.0033  0.0101  0.0065
204  0.2154  ...  0.0051  0.0061  0.0093  0.0135  0.0063  0.0063  0.0034
205  0.2529  ...  0.0155  0.0160  0.0029  0.0051  0.0062  0.0089  0.0140
206  0.2354  ...  0.0042  0.0086  0.0046  0.0126  0.0036  0.0035  0.0034
207  0.2354  ...  0.0181  0.0146  0.0129  0.0047  0.0039  0.0061  0.0040

          57      58      59
0    0.0084  0.0090  0.0032
1    0.0049  0.0052  0.0044
2    0.0164  0.0095  0.0078
3    0.0044  0.0040  0.0117
4    0.0048  0.0107  0.0094
..      ...     ...     ...
203  0.0115  0.0193  0.0157
204  0.0032  0.0062  0.0067
205  0.0138  0.0077  0.0031
206  0.0079  0.0036  0.0048
207  0.0036  0.0061  0.0115

[208 rows x 60 columns]
0      R
1      R
2      R
3      R
4      R
      ..
203    M
204    M
205    M
206    M
207    M
Name: 60, Length: 208, dtype: object
```

In [19]:

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=validation_size, stratify=Y, random_state=1)
```

In [20]:

```
print(X.shape,X_train.shape, X_test.shape)
```

```
(208, 60) (166, 60) (42, 60)
```

5. Evaluate Algorithms

In [21]:

```
model= LogisticRegression()
```

In [22]:

```
# Training the LR model with training dat
model.fit(X_train,Y_train)
```

Out[22]:

```
LogisticRegression()
```

In [23]:

```
# Accuracy on  training data
X_train_predicton = model.predict(X_train)
training_data_accuracy=accuracy_score(X_train_predicton,Y_train)
```

In [24]:

```
print('Accuracy on training data: ',training_data_accuracy)
```

```
Accuracy on training data:  0.8433734939759037
```

In [25]:

```
# Accuracy on  test data
X_test_prediction = model.predict(X_test)
test_data_accuracy = accuracy_score(X_test_prediction,Y_test)
```

In [26]:

```
print('Accuracy on test data: ',test_data_accuracy)
```

```
Accuracy on test data:  0.6904761904761905
```

Use some other algorithums on this classification problem.

- Linear Algorithms: Logistic Regression (LR) and Linear Discriminant Analysis (LDA).
- Nonlinear Algorithms: Classification and Regression Trees (CART), Support Vector Machines (SVM), Gaussian Naive Bayes (NB) and k-Nearest Neighbors (KNN).

In [27]:

```
# Check Algorithms
models=[]
models.append(('LR',LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
```

Let's compare the algorithms. We will display the mean and standard deviation of accuracy for each algorithm. On this type of dataset distance based algorithms like k-Nearest Neighbors and Support Vector Machines may do well.We will also use 10-fold cross validation.

In [28]:

```
# Evaluation Metrics & 10-fold cross validation
num_folds = 10
scoring = 'accuracy'
```

In [29]:

```
results = []
names = []
for name, model in models:
    kfold = KFold(n_splits=num_folds)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

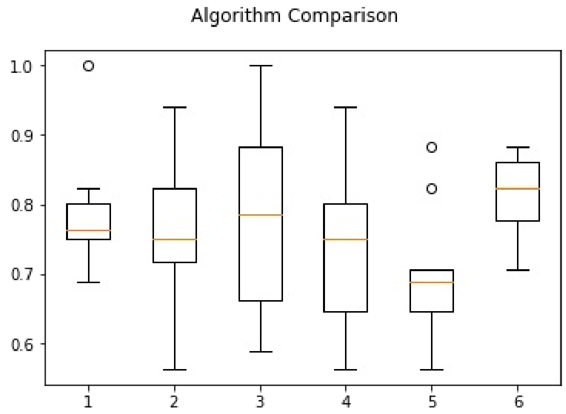
```
LR: 0.782353 (0.082369)
LDA: 0.751838 (0.107706)
KNN: 0.784191 (0.134441)
CART: 0.740441 (0.111455)
NB: 0.697426 (0.088862)
SVM: 0.813603 (0.055320)
```

The results suggest that k-Nearest Neighbors, Support Vector Machines and Logistic Regression may be worth further study.

It is always wise to look at the distribution of accuracy values calculated across cross validation folds. We can do that graphically using box and whisker plots.

In [30]:

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
plt.boxplot(results)
plt.show()
```



Let's evaluate the same algorithms with a standardized copy of the dataset. This is where the data is transformed such that each attribute has a mean value of zero and a standard deviation of one. We also need to avoid data leakage when we transform the data. A good way to avoid leakage is to use pipelines that standardize the data and build the model for each fold in the cross validation test harness.

In [31]:

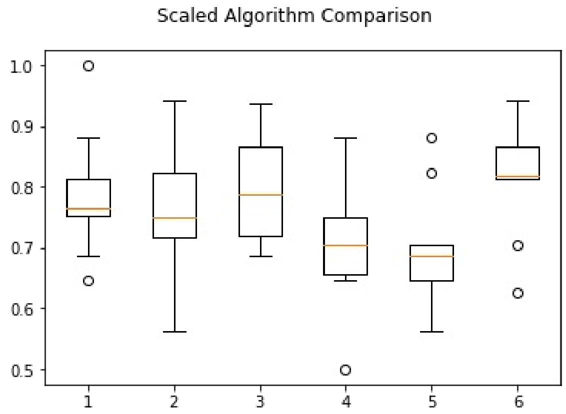
```
# Standardize the dataset
pipelines = []
pipelines.append(('ScaledLR', Pipeline([('Scaler', StandardScaler()),('LR', LogisticRegression())])))
pipelines.append(('ScaledLDA', Pipeline([('Scaler', StandardScaler()),('LDA', LinearDiscriminantAnalysis())])))
pipelines.append(('ScaledKNN', Pipeline([('Scaler', StandardScaler()),('KNN', KNeighborsClassifier())])))
pipelines.append(('ScaledCART', Pipeline([('Scaler', StandardScaler()),('CART', DecisionTreeClassifier())])))
pipelines.append(('ScaledNB', Pipeline([('Scaler', StandardScaler()),('NB', GaussianNB())])))
pipelines.append(('ScaledSVM', Pipeline([('Scaler', StandardScaler()),('SVM', SVC())])))

results = []
names = []
for name, model in pipelines:
    kfold = KFold(n_splits=num_folds)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

```
ScaledLR: 0.788603 (0.093851)
ScaledLDA: 0.751838 (0.107706)
ScaledKNN: 0.794853 (0.083413)
ScaledCART: 0.709926 (0.099175)
ScaledNB: 0.697426 (0.088862)
ScaledSVM: 0.812132 (0.085488)
```

- KNN is still doing well, even better than before.
- The standardization of the data has lifted the skill of SVM to be the most accurate algorithm tested so far.
- Plot the distribution of the accuracy scores using box and whisker plots.

In [32]:

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Scaled Algorithm Comparison')
plt.boxplot(results)
plt.show()
```



The results suggest digging deeper into the SVM and KNN algorithms.

6.0 Algorithm Tuning

- In this section we investigate tuning the parameters for two algorithms KNN and SVM.

6.1 Tuning KNN

In [33]:

```
# Tune scaled KNN
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
neighbors = [1,3,5,7,9,11,13,15,17,19,21]
param_grid = dict(n_neighbors=neighbors)
model = KNeighborsClassifier()
kfold = KFold(n_splits=num_folds)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.841912 using {'n_neighbors': 1}
0.841912 (0.079398) with: {'n_neighbors': 1}
0.813971 (0.098227) with: {'n_neighbors': 3}
0.807353 (0.069992) with: {'n_neighbors': 5}
0.789338 (0.067891) with: {'n_neighbors': 7}
0.758824 (0.061480) with: {'n_neighbors': 9}
0.740809 (0.046540) with: {'n_neighbors': 11}
0.728309 (0.073003) with: {'n_neighbors': 13}
0.739706 (0.079667) with: {'n_neighbors': 15}
0.751838 (0.085263) with: {'n_neighbors': 17}
0.739706 (0.083898) with: {'n_neighbors': 19}
0.739706 (0.088934) with: {'n_neighbors': 21}
```

We have printed the configuration that resulted in the highest accuracy as well as the accuracy of all values tried.

6.2 Tuning SVM

In [34]:

```
# Tune scaled SVM
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
c_values = [0.1, 0.3, 0.5, 0.7, 0.9, 1.0, 1.3, 1.5, 1.7, 2.0]
kernel_values = ['linear', 'poly', 'rbf', 'sigmoid']
param_grid = dict(C=c_values, kernel=kernel_values)
model = SVC()
kfold = KFold(n_splits=num_folds)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.854779 using {'C': 2.0, 'kernel': 'rbf'}
0.829779 (0.098198) with: {'C': 0.1, 'kernel': 'linear'}
0.584926 (0.138124) with: {'C': 0.1, 'kernel': 'poly'}
0.543382 (0.094605) with: {'C': 0.1, 'kernel': 'rbf'}
0.764338 (0.052316) with: {'C': 0.1, 'kernel': 'sigmoid'}
0.788235 (0.087577) with: {'C': 0.3, 'kernel': 'linear'}
0.667647 (0.067355) with: {'C': 0.3, 'kernel': 'poly'}
0.757721 (0.065364) with: {'C': 0.3, 'kernel': 'rbf'}
0.781618 (0.085768) with: {'C': 0.3, 'kernel': 'sigmoid'}
0.763971 (0.088370) with: {'C': 0.5, 'kernel': 'linear'}
0.782721 (0.079355) with: {'C': 0.5, 'kernel': 'poly'}
0.805882 (0.074693) with: {'C': 0.5, 'kernel': 'rbf'}
0.781985 (0.075704) with: {'C': 0.5, 'kernel': 'sigmoid'}
0.776471 (0.076939) with: {'C': 0.7, 'kernel': 'linear'}
0.795221 (0.089950) with: {'C': 0.7, 'kernel': 'poly'}
0.812132 (0.081340) with: {'C': 0.7, 'kernel': 'rbf'}
0.769485 (0.079840) with: {'C': 0.7, 'kernel': 'sigmoid'}
0.764338 (0.064195) with: {'C': 0.9, 'kernel': 'linear'}
0.789706 (0.088446) with: {'C': 0.9, 'kernel': 'poly'}
0.806250 (0.082423) with: {'C': 0.9, 'kernel': 'rbf'}
0.769853 (0.074428) with: {'C': 0.9, 'kernel': 'sigmoid'}
0.764338 (0.064195) with: {'C': 1.0, 'kernel': 'linear'}
0.783824 (0.095501) with: {'C': 1.0, 'kernel': 'poly'}
0.806250 (0.082423) with: {'C': 1.0, 'kernel': 'rbf'}
0.758088 (0.083105) with: {'C': 1.0, 'kernel': 'sigmoid'}
0.758456 (0.066544) with: {'C': 1.3, 'kernel': 'linear'}
0.784191 (0.105086) with: {'C': 1.3, 'kernel': 'poly'}
0.836029 (0.088773) with: {'C': 1.3, 'kernel': 'rbf'}
0.751471 (0.103023) with: {'C': 1.3, 'kernel': 'sigmoid'}
0.770221 (0.076295) with: {'C': 1.5, 'kernel': 'linear'}
0.778309 (0.113876) with: {'C': 1.5, 'kernel': 'poly'}
0.854412 (0.068706) with: {'C': 1.5, 'kernel': 'rbf'}
0.745956 (0.079457) with: {'C': 1.5, 'kernel': 'sigmoid'}
0.752206 (0.079180) with: {'C': 1.7, 'kernel': 'linear'}
0.778309 (0.113876) with: {'C': 1.7, 'kernel': 'poly'}
0.848529 (0.068580) with: {'C': 1.7, 'kernel': 'rbf'}
0.733088 (0.107479) with: {'C': 1.7, 'kernel': 'sigmoid'}
0.752206 (0.064756) with: {'C': 2.0, 'kernel': 'linear'}
0.820221 (0.116996) with: {'C': 2.0, 'kernel': 'poly'}
0.854779 (0.073387) with: {'C': 2.0, 'kernel': 'rbf'}
0.769853 (0.074428) with: {'C': 2.0, 'kernel': 'sigmoid'}
```

We have printed the best configuration, the accuracy as well as the accuracies for all configuration combinations.

- We can see the most accurate configuration was SVM with an RBF kernel and a C value of 2.0.
- The accuracy 85% is seemingly better than what KNN could achieve.

7. Ensemble Methods

Another way that we can improve the performance of algorithms on this problem is by using ensemble methods. In this section we will evaluate four different ensemble machine learning algorithms, two boosting and two bagging methods:

- Boosting Methods: AdaBoost (AB) and Gradient Boosting (GBM).
- Bagging Methods: Random Forests (RF) and Extra Trees (ET).
- No data standardization is used in this case because all four ensemble algorithms are based on decision trees that are less sensitive to data distributions.

In [35]:

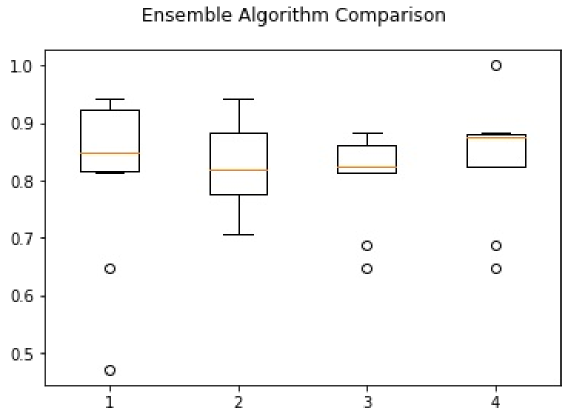
```
# ensembles
ensembles = []
ensembles.append(('AB', AdaBoostClassifier()))
ensembles.append(('GBM', GradientBoostingClassifier()))
ensembles.append(('RF', RandomForestClassifier()))
ensembles.append(('ET', ExtraTreesClassifier()))
results = []
names = []
for name, model in ensembles:
    kfold = KFold(n_splits=num_folds)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

```
AB: 0.815074 (0.141790)
GBM: 0.831250 (0.074904)
RF: 0.806985 (0.075314)
ET: 0.837132 (0.096853)
```

- We can see that both boosting techniques provide strong accuracy scores.
- We can plot the distribution of accuracy scores across the cross validation folds.

In [36]:

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Ensemble Algorithm Comparison')
plt.boxplot(results)
plt.show()
```



The results suggest ET may be worthy of further study, with a strong mean and a spread that skews up towards high accuracy.

8. Finalize Model

- The SVM showed the most promise as a low complexity and stable model for this problem.
- A part of the findings was that SVM performs better when the dataset is standardized so that all attributes have a mean value of zero and a standard deviation of one.

In [37]:

```
# prepare the model
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
model = SVC(C=1.5)
model.fit(rescaledX, Y_train)
# estimate accuracy on validation dataset
rescaledValidationX = scaler.transform(X_test)
predictions = model.predict(rescaledValidationX)
print(accuracy_score(Y_test, predictions))
print(confusion_matrix(Y_test, predictions))
print(classification_report(Y_test, predictions))
```

```
0.8333333333333334
[[18  4]
 [ 3 17]]
              precision    recall  f1-score   support

           M       0.86      0.82      0.84        22
           R       0.81      0.85      0.83        20

    accuracy                           0.83        42
   macro avg       0.83      0.83      0.83        42
weighted avg       0.83      0.83      0.83        42
```

We can see that we achieve an accuracy of nearly 83% on the held-out validation dataset.

9. Summary

We have covered the following points:

- Problem Definition (Sonar return data).
- Loading the Dataset.
- Analyze Data (same scale but different distributions of data).
- Evaluate Algorithms (SVM and KNN looked good).
- Evaluate Algorithms with Standardization (SVM and KNN looked good).
- Algorithm Tuning (K=1 for KNN was good, SVM with an RBF kernel and C=1.5 was best).
- Ensemble Methods (Bagging and Boosting, not quite as good as SVM).
- Finalize Model (use all training data and confirm using validation dataset).

In [40]:

```
#!jupyter nbconvert --to html
```