e-prediction-using-8-models-edited

February 21, 2024

0.1 Importing the Dependencies

[ ]:
```python
import numpy as np #for computations
import pandas as pd #for data storage
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

0.2 Data Collection and Data Processing

[ ]:
```python
#loading the dataset to a pandas Dataframe
sonar_data = pd.read_csv('Datasets/sonar data.csv', header=None)
```

[ ]:
```python
sonar_data.head()
```

[ ]:
```
        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

        9   …      51      52      53      54      55      56      57  \
0  0.2111  …  0.0027  0.0065  0.0159  0.0072  0.0167  0.0180  0.0084
1  0.2872  …  0.0084  0.0089  0.0048  0.0094  0.0191  0.0140  0.0049
2  0.6194  …  0.0232  0.0166  0.0095  0.0180  0.0244  0.0316  0.0164
3  0.1264  …  0.0121  0.0036  0.0150  0.0085  0.0073  0.0050  0.0044
4  0.4459  …  0.0031  0.0054  0.0105  0.0110  0.0015  0.0072  0.0048

        58      59  60
0  0.0090  0.0032   R
1  0.0052  0.0044   R
2  0.0095  0.0078   R
3  0.0040  0.0117   R
4  0.0107  0.0094   R

[5 rows x 61 columns]
```

[ ]:
```
sonar_data.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: 99.3+ KB
```

[ ]:
```
# number of rows and columns
sonar_data.shape
```

[ ]:
```
(208, 61)
```

There are 60 features and 208 data points - last column represents whether it is rock or mine

[ ]:
```
sonar_data.describe()  #describe --> statistical measures of the data
```

[ ]:
```
                0           1           2           3           4           5  \
count  208.000000  208.000000  208.000000  208.000000  208.000000  208.000000
mean     0.029164    0.038437    0.043832    0.053892    0.075202    0.104570
std      0.022991    0.032960    0.038428    0.046528    0.055552    0.059105
min      0.001500    0.000600    0.001500    0.005800    0.006700    0.010200
25%      0.013350    0.016450    0.018950    0.024375    0.038050    0.067025
50%      0.022800    0.030800    0.034300    0.044050    0.062500    0.092150
75%      0.035550    0.047950    0.057950    0.064500    0.100275    0.134125
max      0.137100    0.233900    0.305900    0.426400    0.401000    0.382300

                6           7           8           9  …          50  \
count  208.000000  208.000000  208.000000  208.000000  …  208.000000
mean     0.121747    0.134799    0.178003    0.208259  …    0.016069
std      0.061788    0.085152    0.118387    0.134416  …    0.012008
min      0.003300    0.005500    0.007500    0.011300  …    0.000000
```

```
25%       0.080900    0.080425    0.097025    0.111275  …     0.008425
50%       0.106950    0.112100    0.152250    0.182400  …     0.013900
75%       0.154000    0.169600    0.233425    0.268700  …     0.020825
max       0.372900    0.459000    0.682800    0.710600  …     0.100400

                51          52          53          54          55          56  \
count  208.000000  208.000000  208.000000  208.000000  208.000000  208.000000
mean     0.013420    0.010709    0.010941    0.009290    0.008222    0.007820
std      0.009634    0.007060    0.007301    0.007088    0.005736    0.005785
min      0.000800    0.000500    0.001000    0.000600    0.000400    0.000300
25%      0.007275    0.005075    0.005375    0.004150    0.004400    0.003700
50%      0.011400    0.009550    0.009300    0.007500    0.006850    0.005950
75%      0.016725    0.014900    0.014500    0.012100    0.010575    0.010425
max      0.070900    0.039000    0.035200    0.044700    0.039400    0.035500

                57          58          59
count  208.000000  208.000000  208.000000
mean     0.007949    0.007941    0.006507
std      0.006470    0.006181    0.005031
min      0.000300    0.000100    0.000600
25%      0.003600    0.003675    0.003100
50%      0.005800    0.006400    0.005300
75%      0.010350    0.010325    0.008525
max      0.044000    0.036400    0.043900

[8 rows x 60 columns]
```

[ ]:
```
sonar_data[60].value_counts()
```

[ ]:
```
60
M    111
R     97
Name: count, dtype: int64
```

[ ]:
```
sonar_data.nunique()
```

[ ]:
```
0     177
1     182
2     190
3     181
4     193
     …
56    121
57    124
58    119
59    109
60      2
```

```
Length: 61, dtype: int64
```

[ ]:
```
duplicated_rows = sonar_data.duplicated()
```

[ ]:
```
duplicated_rows
```

[ ]:
```
0      False
1      False
2      False
3      False
4      False
       …
203    False
204    False
205    False
206    False
207    False
Length: 208, dtype: bool
```

M -> Mine

R -> Rock

[ ]:
```
sonar_data.groupby(60).mean()
```

[ ]:
```
          0         1         2         3         4         5         6   \
60
M   0.034989  0.045544  0.050720  0.064768  0.086715  0.111864  0.128359
R   0.022498  0.030303  0.035951  0.041447  0.062028  0.096224  0.114180

          7         8         9   …        50        51        52        53  \
60                                …
M   0.149832  0.213492  0.251022  …  0.019352  0.016014  0.011643  0.012185
R   0.117596  0.137392  0.159325  …  0.012311  0.010453  0.009640  0.009518

          54        55        56        57        58        59
60
M   0.009923  0.008914  0.007825  0.009060  0.008695  0.006930
R   0.008567  0.007430  0.007814  0.006677  0.007078  0.006024

[2 rows x 60 columns]
```

[ ]:
```
# separating data and Labels
X = sonar_data.drop(columns=60, axis=1)
Y = sonar_data[60]
```

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
correlation_matrix = X.corr()
```

[ ]:
```
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
```

[ ]:
```
correlation_matrix
```

[ ]:
```
          0         1         2         3         4         5         6   \
0   1.000000  0.735896  0.571537  0.491438  0.344797  0.238921  0.260815
1   0.735896  1.000000  0.779916  0.606684  0.419669  0.332329  0.279040
2   0.571537  0.779916  1.000000  0.781786  0.546141  0.346275  0.190434
3   0.491438  0.606684  0.781786  1.000000  0.726943  0.352805  0.246440
4   0.344797  0.419669  0.546141  0.726943  1.000000  0.597053  0.335422
5   0.238921  0.332329  0.346275  0.352805  0.597053  1.000000  0.702889
6   0.260815  0.279040  0.190434  0.246440  0.335422  0.702889  1.000000
7   0.355523  0.334615  0.237884  0.246742  0.204006  0.471683  0.675774
8   0.353420  0.316733  0.252691  0.247078  0.177906  0.327578  0.470580
9   0.318276  0.270782  0.219637  0.237769  0.183219  0.288621  0.425448
10  0.344058  0.297065  0.274610  0.271881  0.231684  0.333570  0.396588
11  0.210861  0.194102  0.214807  0.175381  0.211657  0.344451  0.274432
12  0.210722  0.249596  0.258767  0.215754  0.299086  0.411107  0.365391
13  0.256278  0.273170  0.291724  0.286708  0.359062  0.396233  0.409576
14  0.304878  0.307599  0.285663  0.278529  0.318059  0.367908  0.411692
15  0.239079  0.261844  0.237017  0.248245  0.328725  0.353783  0.363086
16  0.137845  0.152170  0.201093  0.223203  0.326477  0.293190  0.250024
17  0.041817  0.042870  0.120587  0.194992  0.299266  0.235778  0.208057
18  0.055227  0.040911  0.099303  0.189405  0.340543  0.226305  0.215495
19  0.156760  0.102428  0.103117  0.188317  0.285737  0.206841  0.196496
20  0.117663  0.075255  0.063990  0.142271  0.205088  0.174768  0.165827
21 -0.056973 -0.074157 -0.026815  0.036010  0.152897  0.123770  0.063773
22 -0.163426 -0.179365 -0.073400 -0.029749  0.073934  0.064081  0.009359
23 -0.218093 -0.196469 -0.085380 -0.102975 -0.000624  0.027026  0.011982
24 -0.295683 -0.295302 -0.214256 -0.206673 -0.067296 -0.043280 -0.057147
25 -0.342865 -0.365749 -0.291974 -0.291357 -0.125675 -0.100309 -0.126074
26 -0.341703 -0.337046 -0.263111 -0.294749 -0.169618 -0.129094 -0.179526
27 -0.224340 -0.234386 -0.256674 -0.256074 -0.214692 -0.118645 -0.116848
28 -0.199099 -0.228490 -0.290728 -0.300476 -0.283863 -0.156081 -0.129694
29 -0.077430 -0.115301 -0.197493 -0.236602 -0.273350 -0.151186 -0.068142
30 -0.048370 -0.055862 -0.106198 -0.190086 -0.214336 -0.054136 -0.096945
31 -0.030444 -0.049683 -0.109895 -0.169987 -0.173485 -0.051934 -0.115871
```

32	-0.031939	-0.108272	-0.170671	-0.164651	-0.200586	-0.144391	-0.127052
33	0.031319	-0.004247	-0.099409	-0.083965	-0.140559	-0.070337	-0.077662
34	0.098118	0.115824	0.017053	0.015200	-0.086529	-0.028815	-0.015531
35	0.080722	0.132611	0.053070	0.039282	-0.073481	-0.023621	0.002979
36	0.119565	0.169186	0.107530	0.063486	-0.064617	-0.064798	-0.001376
37	0.209873	0.217494	0.130276	0.089887	-0.008620	-0.048745	0.065900
38	0.208371	0.186828	0.110499	0.089346	0.063408	0.030599	0.080942
39	0.099993	0.098350	0.074137	0.045141	0.061616	0.081119	0.112673
40	0.127313	0.188226	0.189047	0.145241	0.098832	0.075797	0.041071
41	0.213592	0.261345	0.233442	0.144693	0.125181	0.048763	-0.028720
42	0.206057	0.186368	0.113920	0.050629	0.063706	0.034380	-0.025727
43	0.157949	0.133018	0.071946	-0.008407	0.031575	0.048870	0.061404
44	0.279968	0.285716	0.180734	0.087824	0.089202	0.085468	0.110813
45	0.319354	0.304247	0.173649	0.080012	0.081964	0.029524	0.076537
46	0.230343	0.255797	0.179528	0.046109	0.041419	0.016640	0.098925
47	0.203234	0.265279	0.234896	0.121065	0.084435	0.067196	0.155221
48	0.247560	0.313995	0.223074	0.133294	0.088128	0.080729	0.194720
49	0.269287	0.245868	0.081096	0.077925	0.066751	0.017300	0.166112
50	0.254450	0.320538	0.238110	0.174676	0.115936	0.171767	0.184152
51	0.355299	0.434548	0.394076	0.374651	0.266617	0.252288	0.144051
52	0.311729	0.346076	0.332914	0.364772	0.314985	0.162404	0.046403
53	0.322299	0.383960	0.367186	0.334211	0.205306	0.164073	0.163074
54	0.312067	0.380165	0.289731	0.284955	0.196472	0.133464	0.195541
55	0.220642	0.262263	0.287661	0.280938	0.199323	0.166758	0.174143
56	0.313725	0.280341	0.380819	0.340254	0.219395	0.161333	0.186324
57	0.368132	0.353042	0.334108	0.344865	0.238793	0.203986	0.242646
58	0.357116	0.352200	0.425047	0.420266	0.290982	0.220573	0.183578
59	0.347078	0.358761	0.373948	0.400626	0.253710	0.178158	0.222493

	7	8	9	…	50	51	52	53 \
0	0.355523	0.353420	0.318276	…	0.254450	0.355299	0.311729	0.322299
1	0.334615	0.316733	0.270782	…	0.320538	0.434548	0.346076	0.383960
2	0.237884	0.252691	0.219637	…	0.238110	0.394076	0.332914	0.367186
3	0.246742	0.247078	0.237769	…	0.174676	0.374651	0.364772	0.334211
4	0.204006	0.177906	0.183219	…	0.115936	0.266617	0.314985	0.205306
5	0.471683	0.327578	0.288621	…	0.171767	0.252288	0.162404	0.164073
6	0.675774	0.470580	0.425448	…	0.184152	0.144051	0.046403	0.163074
7	1.000000	0.778577	0.652525	…	0.260692	0.219038	0.102447	0.234008
8	0.778577	1.000000	0.877131	…	0.174873	0.207996	0.105352	0.202615
9	0.652525	0.877131	1.000000	…	0.167096	0.165537	0.097544	0.146725
10	0.584583	0.728063	0.853140	…	0.157615	0.165748	0.084801	0.142572
11	0.328329	0.363404	0.485392	…	0.113418	0.117699	0.042263	0.078457
12	0.322951	0.316899	0.405370	…	0.203347	0.147479	0.058599	0.160916
13	0.387114	0.329659	0.345684	…	0.180464	0.137443	0.133196	0.210925
14	0.391514	0.299575	0.294699	…	0.153162	0.135271	0.103444	0.218703
15	0.322237	0.241819	0.242869	…	0.099892	0.104039	0.096325	0.206922
16	0.140912	0.100146	0.121264	…	0.009036	0.020313	0.035635	0.129138

17	0.061333	0.027380	0.063745	…	-0.104656	-0.057236	-0.006627	0.072344
18	0.061825	0.067237	0.099632	…	-0.050988	0.011450	0.051367	0.120153
19	0.204950	0.266455	0.246924	…	-0.022960	0.028754	0.069692	0.171936
20	0.208785	0.264109	0.240862	…	-0.024222	-0.034845	-0.000270	0.167327
21	0.023786	0.019512	0.070381	…	-0.061054	-0.092204	-0.094203	0.045224
22	-0.092087	-0.154752	-0.094887	…	-0.108172	-0.108934	-0.152691	-0.055641
23	-0.124427	-0.189343	-0.178304	…	-0.143650	-0.175022	-0.225897	-0.125419
24	-0.196354	-0.198658	-0.179890	…	-0.200752	-0.250479	-0.238478	-0.219817
25	-0.203178	-0.137459	-0.109051	…	-0.191554	-0.256166	-0.248400	-0.277169
26	-0.233332	-0.119143	-0.095820	…	-0.137886	-0.191707	-0.259825	-0.269558
27	-0.120343	-0.028002	-0.052303	…	-0.027750	-0.064730	-0.148791	-0.213300
28	-0.139750	-0.093413	-0.137173	…	-0.000251	-0.054779	-0.130527	-0.235110
29	-0.017654	0.053398	-0.043998	…	0.038928	0.039053	-0.034937	-0.149564
30	-0.081072	-0.041649	-0.091193	…	0.048936	0.087360	0.026300	-0.146485
31	-0.108115	-0.028629	-0.058493	…	0.059594	0.090863	0.017997	-0.089302
32	-0.087246	-0.017885	-0.027245	…	-0.002591	0.003084	0.029192	-0.037753
33	-0.014578	0.013594	-0.021291	…	0.003386	0.008364	0.095110	0.064450
34	0.035733	0.015065	-0.035765	…	0.018382	0.052650	0.122798	0.138357
35	0.087187	0.036120	-0.004460	…	0.006165	0.023165	0.072182	0.136711
36	0.110739	0.111769	0.085072	…	-0.028291	0.002078	0.079799	0.130427
37	0.186609	0.223983	0.175717	…	0.094205	0.134015	0.171104	0.206931
38	0.206145	0.211897	0.233833	…	0.124038	0.108564	0.167599	0.200116
39	0.184411	0.122735	0.177357	…	0.066673	0.042677	0.128310	0.121381
40	0.097517	0.019589	-0.002523	…	0.277471	0.255774	0.254064	0.181579
41	0.076054	-0.005785	-0.018880	…	0.428751	0.359439	0.283622	0.214580
42	0.114721	0.052409	0.076138	…	0.397190	0.302861	0.253203	0.155339
43	0.135426	0.215710	0.216742	…	0.316501	0.217849	0.139544	0.095210
44	0.240176	0.320573	0.287459	…	0.416973	0.350208	0.181292	0.162879
45	0.169099	0.195447	0.138447	…	0.505304	0.429309	0.236971	0.187964
46	0.109744	0.084191	0.090662	…	0.570575	0.398600	0.206970	0.159920
47	0.222783	0.225667	0.268123	…	0.573572	0.365149	0.206376	0.209084
48	0.271422	0.222135	0.264885	…	0.526095	0.319286	0.150871	0.195826
49	0.191615	0.150527	0.162010	…	0.447926	0.341667	0.279681	0.280477
50	0.260692	0.174873	0.167096	…	1.000000	0.627038	0.330396	0.384052
51	0.219038	0.207996	0.165537	…	0.627038	1.000000	0.540414	0.343190
52	0.102447	0.105352	0.097544	…	0.330396	0.540414	1.000000	0.412337
53	0.234008	0.202615	0.146725	…	0.384052	0.343190	0.412337	1.000000
54	0.239551	0.179342	0.175254	…	0.278935	0.337581	0.315656	0.455059
55	0.276819	0.232764	0.151889	…	0.209752	0.203121	0.421588	0.397378
56	0.267212	0.193963	0.140327	…	0.191407	0.191264	0.308197	0.361443
57	0.287603	0.231745	0.212277	…	0.325665	0.309673	0.370764	0.404117
58	0.194400	0.097293	0.058273	…	0.317942	0.298711	0.346095	0.447118
59	0.146216	0.095243	0.097358	…	0.246764	0.195379	0.280780	0.283471

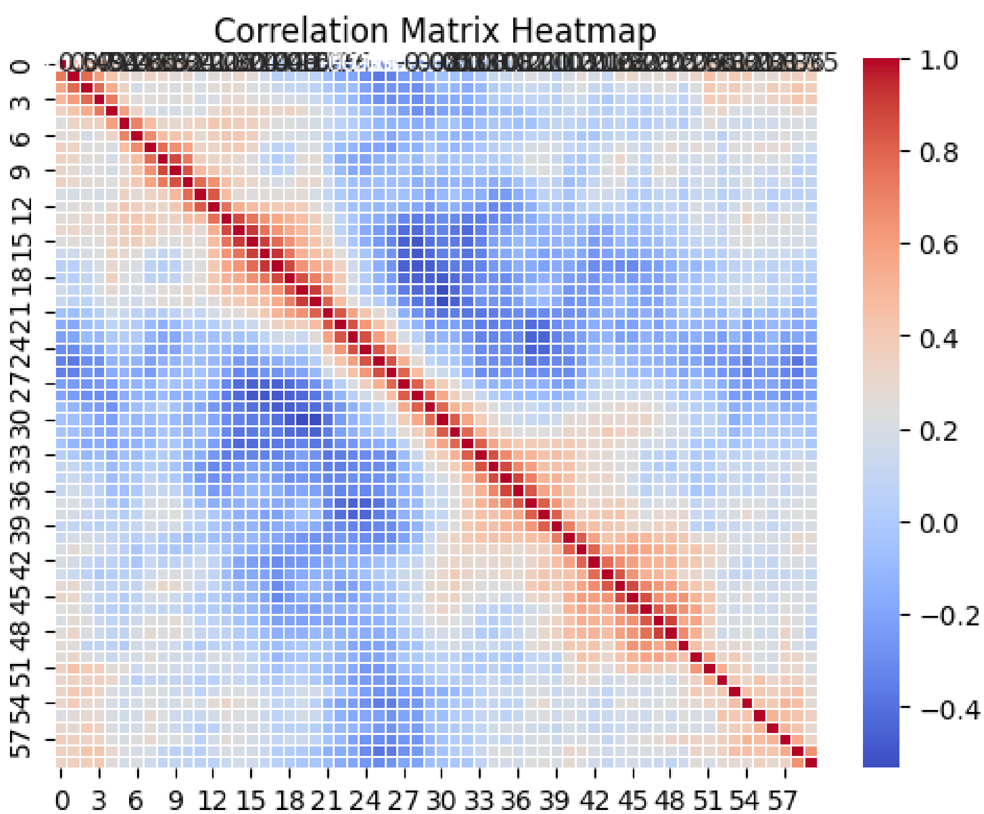
	54	55	56	57	58	59
0	0.312067	0.220642	0.313725	0.368132	0.357116	0.347078
1	0.380165	0.262263	0.280341	0.353042	0.352200	0.358761

2	0.289731	0.287661	0.380819	0.334108	0.425047	0.373948
3	0.284955	0.280938	0.340254	0.344865	0.420266	0.400626
4	0.196472	0.199323	0.219395	0.238793	0.290982	0.253710
5	0.133464	0.166758	0.161333	0.203986	0.220573	0.178158
6	0.195541	0.174143	0.186324	0.242646	0.183578	0.222493
7	0.239551	0.276819	0.267212	0.287603	0.194400	0.146216
8	0.179342	0.232764	0.193963	0.231745	0.097293	0.095243
9	0.175254	0.151889	0.140327	0.212277	0.058273	0.097358
10	0.228991	0.122332	0.103405	0.193358	0.067726	0.089695
11	0.164590	0.115658	0.030732	0.065273	0.044614	0.071364
12	0.272492	0.183743	0.057870	0.171140	0.151804	0.061411
13	0.326821	0.252166	0.190886	0.258675	0.209122	0.120966
14	0.261822	0.218395	0.202511	0.225545	0.193671	0.171089
15	0.240968	0.215478	0.191736	0.198019	0.182337	0.158438
16	0.168460	0.128968	0.145708	0.148563	0.121800	0.093992
17	0.093767	0.080812	0.056930	0.096022	0.028446	0.046617
18	0.099082	0.121331	0.045204	0.138365	0.023019	0.007468
19	0.157272	0.178498	0.066425	0.132453	-0.005364	-0.028540
20	0.059823	0.139089	0.030943	0.079818	-0.049413	-0.025201
21	-0.119720	-0.030877	-0.069909	-0.035829	-0.143209	-0.085696
22	-0.198577	-0.138900	-0.098291	-0.105235	-0.168149	-0.163696
23	-0.229297	-0.178320	-0.117429	-0.210556	-0.186527	-0.190877
24	-0.276419	-0.187789	-0.157967	-0.270222	-0.303155	-0.253233
25	-0.353657	-0.215894	-0.254240	-0.303427	-0.385725	-0.303949
26	-0.347931	-0.263403	-0.267069	-0.321868	-0.360340	-0.267596
27	-0.262620	-0.198093	-0.190854	-0.261443	-0.275442	-0.195130
28	-0.246181	-0.221273	-0.228155	-0.267938	-0.247318	-0.203776
29	-0.127523	-0.063403	-0.072976	-0.134302	-0.129402	-0.076100
30	-0.080546	-0.067373	-0.018733	-0.036092	-0.044197	-0.043015
31	-0.012792	0.017714	0.010611	0.018564	0.013499	-0.023863
32	0.000520	0.030027	0.045806	0.003712	0.054285	-0.015804
33	0.089024	0.109288	0.106959	0.083192	0.138214	0.075686
34	0.110776	0.131490	0.168361	0.143897	0.227783	0.191193
35	0.074314	0.069959	0.189471	0.106275	0.222683	0.176982
36	0.086914	0.116549	0.180789	0.110760	0.163162	0.166263
37	0.235457	0.217587	0.156320	0.169710	0.206001	0.233288
38	0.294578	0.223133	0.143131	0.218912	0.231150	0.222611
39	0.157435	0.150700	0.105603	0.143718	0.189058	0.202034
40	0.177851	0.220670	0.193532	0.196282	0.304521	0.281889
41	0.175505	0.157192	0.157646	0.201077	0.276762	0.220597
42	0.097671	0.123574	0.104120	0.210814	0.199334	0.161416
43	0.097255	0.133169	0.108185	0.109166	0.154547	0.108190
44	0.242757	0.170750	0.144281	0.167337	0.178402	0.157181
45	0.269119	0.178182	0.162125	0.237890	0.205291	0.180691
46	0.194223	0.146042	0.157815	0.240471	0.209045	0.139727
47	0.210950	0.219052	0.196814	0.270198	0.221425	0.123666
48	0.230033	0.155186	0.173098	0.328238	0.209152	0.088640

49	0.287612	0.235053	0.201609	0.342866	0.178118	0.139944
50	0.278935	0.209752	0.191407	0.325665	0.317942	0.246764
51	0.337581	0.203121	0.191264	0.309673	0.298711	0.195379
52	0.315656	0.421588	0.308197	0.370764	0.346095	0.280780
53	0.455059	0.397378	0.361443	0.404117	0.447118	0.283471
54	1.000000	0.429948	0.387204	0.503465	0.453658	0.264399
55	0.429948	1.000000	0.515154	0.463659	0.430804	0.349449
56	0.387204	0.515154	1.000000	0.509805	0.431295	0.287219
57	0.503465	0.463659	0.509805	1.000000	0.550235	0.329827
58	0.453658	0.430804	0.431295	0.550235	1.000000	0.642872
59	0.264399	0.349449	0.287219	0.329827	0.642872	1.000000

[60 rows x 60 columns]

[ ]:
```
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f",␣
    ↪linewidths=.5)
plt.title('Correlation Matrix Heatmap')
plt.show()
```



```
[ ]: missing_values = sonar_data.isnull().sum()
     for i in missing_values:
         if i>0:
             print(i)
```

0.3 Training and Test data

```
[ ]: X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.1,␣
      ↪stratify=Y, random_state=1)
```

X_Train, Y_Train are the training data , X_Test, Y_Test are the testing data - X_Test_predictions are the predicted values. Test data size will 0.1 times that is 10% of the original dataset while others will be training dataset. Stratify - separate based on Y - based on rock and mine, if random_state is 1 then throughout any system random state 1 will be same split of data

```
[ ]: print(X.shape, X_train.shape, X_test.shape)
```

```
(208, 60) (187, 60) (21, 60)
```

```
[ ]: print(X_train)
     print(Y_train)
```

```
         0       1       2       3       4       5       6       7       8   \
115  0.0414  0.0436  0.0447  0.0844  0.0419  0.1215  0.2002  0.1516  0.0818
38   0.0123  0.0022  0.0196  0.0206  0.0180  0.0492  0.0033  0.0398  0.0791
56   0.0152  0.0102  0.0113  0.0263  0.0097  0.0391  0.0857  0.0915  0.0949
123  0.0270  0.0163  0.0341  0.0247  0.0822  0.1256  0.1323  0.1584  0.2017
18   0.0270  0.0092  0.0145  0.0278  0.0412  0.0757  0.1026  0.1138  0.0794
..      …      …      …      …      …      …      …      …      …
140  0.0412  0.1135  0.0518  0.0232  0.0646  0.1124  0.1787  0.2407  0.2682
5    0.0286  0.0453  0.0277  0.0174  0.0384  0.0990  0.1201  0.1833  0.2105
154  0.0117  0.0069  0.0279  0.0583  0.0915  0.1267  0.1577  0.1927  0.2361
131  0.1150  0.1163  0.0866  0.0358  0.0232  0.1267  0.2417  0.2661  0.4346
203  0.0187  0.0346  0.0168  0.0177  0.0393  0.1630  0.2028  0.1694  0.2328

         9   …      50      51      52      53      54      55      56  \
115  0.1975  …  0.0222  0.0045  0.0136  0.0113  0.0053  0.0165  0.0141
38   0.0475  …  0.0149  0.0125  0.0134  0.0026  0.0038  0.0018  0.0113
56   0.1504  …  0.0048  0.0049  0.0041  0.0036  0.0013  0.0046  0.0037
123  0.2122  …  0.0197  0.0189  0.0204  0.0085  0.0043  0.0092  0.0138
18   0.1520  …  0.0045  0.0084  0.0010  0.0018  0.0068  0.0039  0.0120
..      …  …     …      …      …      …      …      …      …
140  0.2058  …  0.0798  0.0376  0.0143  0.0272  0.0127  0.0166  0.0095
5    0.3039  …  0.0104  0.0045  0.0014  0.0038  0.0013  0.0089  0.0057
154  0.2169  …  0.0039  0.0053  0.0029  0.0020  0.0013  0.0029  0.0020
131  0.5378  …  0.0228  0.0099  0.0065  0.0085  0.0166  0.0110  0.0190
203  0.2684  …  0.0203  0.0116  0.0098  0.0199  0.0033  0.0101  0.0065
```

```
         57      58      59
115  0.0077  0.0246  0.0198
38   0.0058  0.0047  0.0071
56   0.0011  0.0034  0.0033
123  0.0094  0.0105  0.0093
18   0.0132  0.0070  0.0088
..      …       …       …
140  0.0225  0.0098  0.0085
5    0.0027  0.0051  0.0062
154  0.0062  0.0026  0.0052
131  0.0141  0.0068  0.0086
203  0.0115  0.0193  0.0157

[187 rows x 60 columns]
115    M
38     R
56     R
123    M
18     R
      ..
140    M
5      R
154    M
131    M
203    M
Name: 60, Length: 187, dtype: object
```

0.4 Model Training –> Logistic Regression

[ ]:
```
model = LogisticRegression()
```

[ ]:
```
#training the Logistic Regression model with training data
model.fit(X_train, Y_train)
```

[ ]:
```
LogisticRegression()
```

0.5 Model Evaluation

[ ]:
```
#accuracy on training data
X_train_prediction = model.predict(X_train)
training_data_accuracy = accuracy_score(X_train_prediction, Y_train)
```

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

```
Accuracy on training data :  0.8342245989304813
```

[ ]:
```
#accuracy on test data
X_test_prediction = model.predict(X_test)
test_data_accuracy = accuracy_score(X_test_prediction, Y_test)
```

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

```
Accuracy on test data :  0.7619047619047619
```

[ ]:
```
# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, X_test_prediction))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, X_test_prediction))
```

```
Classification Report:
              precision    recall  f1-score   support

           M       0.75      0.82      0.78        11
           R       0.78      0.70      0.74        10

    accuracy                           0.76        21
   macro avg       0.76      0.76      0.76        21
weighted avg       0.76      0.76      0.76        21


Confusion Matrix:
[[9 2]
 [3 7]]
```

For Logistic Regression: We can see that the accuracy is decent - Confusion matrix shows better classification

0.6 Making a Predictive System

[ ]:
```
input_data = (0.0307,0.0523,0.0653,0.0521,0.0611,0.0577,0.0665,0.0664,0.1460,0.
 ↪2792,0.3877,0.4992,0.4981,0.4972,0.5607,0.7339,0.8230,0.9173,0.9975,0.9911,0.
 ↪8240,0.6498,0.5980,0.4862,0.3150,0.1543,0.0989,0.0284,0.1008,0.2636,0.2694,0.
 ↪2930,0.2925,0.3998,0.3660,0.3172,0.4609,0.4374,0.1820,0.3376,0.6202,0.4448,0.
 ↪1863,0.1420,0.0589,0.0576,0.0672,0.0269,0.0245,0.0190,0.0063,0.0321,0.0189,0.
 ↪0137,0.0277,0.0152,0.0052,0.0121,0.0124,0.0055)

# changing the input_data to a numpy array
input_data_as_numpy_array = np.asarray(input_data)

# reshape the np array as we are predicting for one instance
input_data_reshaped = input_data_as_numpy_array.reshape(1,-1)
```

```
prediction = model.predict(input_data_reshaped)
print(prediction)

if (prediction[0]=='R'):
  print('The object is a Rock')
else:
  print('The object is a mine')
```

```
['M']
The object is a mine
```

0.7 Feature Scaling

0.7.1 Logistic Regression without PCA

[ ]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

model = LogisticRegression(solver='lbfgs', random_state=1)
model.fit(X_train_scaled, Y_train)

y_pred = model.predict(X_test_scaled)

accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.76

Classification Report:
              precision    recall  f1-score   support

           M       0.75      0.82      0.78        11
           R       0.78      0.70      0.74        10
```

```
    accuracy                           0.76        21
   macro avg       0.76      0.76      0.76        21
weighted avg       0.76      0.76      0.76        21


Confusion Matrix:
[[9 2]
 [3 7]]
```

0.7.2 Logistic Regression with PCA

```python
model = LogisticRegression(solver='lbfgs', random_state=1)
model.fit(X_train_pca, Y_train)

# Make predictions
y_pred = model.predict(X_test_pca)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.71

Classification Report:
              precision    recall  f1-score   support

           M       0.69      0.82      0.75        11
           R       0.75      0.60      0.67        10

    accuracy                           0.71        21
   macro avg       0.72      0.71      0.71        21
weighted avg       0.72      0.71      0.71        21


Confusion Matrix:
[[9 2]
 [4 6]]
```

For Logistic Regression with PCA: We can see that the accuracy is not improved but reduced than the normal Logistic regression without PCA - Confusion matrix shows worser classification for negative examples and the positive examples remains the same

0.7.3 Random Forest Classifier with PCA

[ ]:
```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Dimensionality reduction using PCA
pca = PCA(n_components=10)  # You can adjust the number of components based on␣
↪your needs
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

# Create and train the Random Forest classifier
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train_pca, Y_train)

# Make predictions
y_pred = model.predict(X_test_pca)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.81

Classification Report:
              precision    recall  f1-score   support

           M       0.77      0.91      0.83        11
           R       0.88      0.70      0.78        10

    accuracy                           0.81        21
   macro avg       0.82      0.80      0.81        21
weighted avg       0.82      0.81      0.81        21


Confusion Matrix:
[[10  1]
 [ 3  7]]
```

For Random forest classifier with PCA: We can see that the accuracy is improved - Confusion matrix shows better classification of positive values but the negative values remains the same

0.7.4 Random forest classifier without PCA

[ ]:
```python
model.fit(X_train, Y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.76

Classification Report:
              precision    recall  f1-score   support

           M       0.75      0.82      0.78        11
           R       0.78      0.70      0.74        10

    accuracy                           0.76        21
   macro avg       0.76      0.76      0.76        21
weighted avg       0.76      0.76      0.76        21


Confusion Matrix:
[[9 2]
 [3 7]]
```

0.7.5 SVM with PCA

[ ]:
```python
from sklearn.svm import SVC

# Create and train the Support Vector Machine (SVM) classifier
model = SVC(kernel='linear', C=1.0, random_state=1)
model.fit(X_train_pca, Y_train)

# Make predictions
y_pred = model.predict(X_test_pca)
```

```
# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.76

Classification Report:
              precision    recall  f1-score   support

           M       0.71      0.91      0.80        11
           R       0.86      0.60      0.71        10

    accuracy                           0.76        21
   macro avg       0.79      0.75      0.75        21
weighted avg       0.78      0.76      0.76        21


Confusion Matrix:
[[10  1]
 [ 4  6]]
```

For Support Vector Machine with PCA: We can see that the accuracy is almost the same as logistic regression without PCA - Confusion matrix shows better classification of positive values but the negative values is only decent

0.7.6 SVM without PCA

[ ]:
```
model = SVC(kernel='linear', C=1.0, random_state=1)
model.fit(X_train, Y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))
```

```
print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.71

Classification Report:
              precision    recall  f1-score   support

           M       0.73      0.73      0.73        11
           R       0.70      0.70      0.70        10

    accuracy                           0.71        21
   macro avg       0.71      0.71      0.71        21
weighted avg       0.71      0.71      0.71        21


Confusion Matrix:
[[8 3]
 [3 7]]
```

For SVM without PCA: We can see that the accuracy is getting worser - The other methods have better accuracy than this

0.7.7 Decision Tree without PCA

[ ]:
```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(random_state=1)
model.fit(X_train, Y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.81

Classification Report:
              precision    recall  f1-score   support
```

```
           M       0.89      0.73      0.80        11
           R       0.75      0.90      0.82        10

    accuracy                           0.81        21
   macro avg       0.82      0.81      0.81        21
weighted avg       0.82      0.81      0.81        21


Confusion Matrix:
[[8 3]
 [1 9]]
```

0.7.8 Decision Tree with PCA

[ ]:
```python
model.fit(X_train_pca, Y_train)

# Make predictions
y_pred = model.predict(X_test_pca)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.86

Classification Report:
              precision    recall  f1-score   support

           M       0.83      0.91      0.87        11
           R       0.89      0.80      0.84        10

    accuracy                           0.86        21
   macro avg       0.86      0.85      0.86        21
weighted avg       0.86      0.86      0.86        21


Confusion Matrix:
[[10  1]
 [ 2  8]]
```

0.7.9 Neural network without PCA

[ ]:
```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(hidden_layer_sizes=(100,), max_iter=1000, random_state=1)

model.fit(X_train, Y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.86

Classification Report:
              precision    recall  f1-score   support

           M       0.83      0.91      0.87        11
           R       0.89      0.80      0.84        10

    accuracy                           0.86        21
   macro avg       0.86      0.85      0.86        21
weighted avg       0.86      0.86      0.86        21


Confusion Matrix:
[[10  1]
 [ 2  8]]
```

0.7.10 Neural Network with PCA

[ ]:
```
model.fit(X_train_pca, Y_train)

# Make predictions
y_pred = model.predict(X_test_pca)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

```python
# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.81

Classification Report:
              precision    recall  f1-score   support

           M       0.82      0.82      0.82        11
           R       0.80      0.80      0.80        10

    accuracy                           0.81        21
   macro avg       0.81      0.81      0.81        21
weighted avg       0.81      0.81      0.81        21


Confusion Matrix:
[[9 2]
 [2 8]]
```

0.7.11 KNN without PCA

[ ]:
```python
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=3)

model.fit(X_train.values, Y_train.values)

# Make predictions
y_pred = model.predict(X_test.values)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.90
```

```
Classification Report:
              precision    recall  f1-score   support

           M       0.85      1.00      0.92        11
           R       1.00      0.80      0.89        10

    accuracy                           0.90        21
   macro avg       0.92      0.90      0.90        21
weighted avg       0.92      0.90      0.90        21


Confusion Matrix:
[[11  0]
 [ 2  8]]
```

0.7.12 KNN with PCA

[ ]:
```python
model.fit(X_train_pca, Y_train)

# Make predictions
y_pred = model.predict(X_test_pca)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.95

Classification Report:
              precision    recall  f1-score   support

           M       0.92      1.00      0.96        11
           R       1.00      0.90      0.95        10

    accuracy                           0.95        21
   macro avg       0.96      0.95      0.95        21
weighted avg       0.96      0.95      0.95        21


Confusion Matrix:
[[11  0]
```

```
 [ 1  9]]
```

0.7.13 Native Bayes without PCA

[ ]:
```python
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X_train, Y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.62

Classification Report:
              precision    recall  f1-score   support

           M       0.67      0.55      0.60        11
           R       0.58      0.70      0.64        10

    accuracy                           0.62        21
   macro avg       0.62      0.62      0.62        21
weighted avg       0.63      0.62      0.62        21


Confusion Matrix:
[[6 5]
 [3 7]]
```

0.7.14 Native Bayes without PCA

[ ]:
```python
model.fit(X_train_pca, Y_train)

# Make predictions
y_pred = model.predict(X_test_pca)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
```

```
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.76

Classification Report:
              precision    recall  f1-score   support

           M       0.80      0.73      0.76        11
           R       0.73      0.80      0.76        10

    accuracy                           0.76        21
   macro avg       0.76      0.76      0.76        21
weighted avg       0.77      0.76      0.76        21


Confusion Matrix:
[[8 3]
 [2 8]]
```

0.7.15 Gradient Boosting Classifier without PCA

[ ]:
```
from sklearn.ensemble import GradientBoostingClassifier
model = GradientBoostingClassifier(random_state=1)
model.fit(X_train, Y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.76
```

```
Classification Report:
              precision    recall  f1-score   support

           M       0.75      0.82      0.78        11
           R       0.78      0.70      0.74        10

    accuracy                           0.76        21
   macro avg       0.76      0.76      0.76        21
weighted avg       0.76      0.76      0.76        21


Confusion Matrix:
[[9 2]
 [3 7]]
```

0.7.16 Gradient Boosting with PCA

[ ]:
```python
model.fit(X_train_pca, Y_train)

# Make predictions
y_pred = model.predict(X_test_pca)

# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display classification report and confusion matrix
print('\nClassification Report:')
print(classification_report(Y_test, y_pred))

print('\nConfusion Matrix:')
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy: 0.81

Classification Report:
              precision    recall  f1-score   support

           M       0.77      0.91      0.83        11
           R       0.88      0.70      0.78        10

    accuracy                           0.81        21
   macro avg       0.82      0.80      0.81        21
weighted avg       0.82      0.81      0.81        21


Confusion Matrix:
```

```
[[10  1]
 [ 3  7]]
```

```
[ ]:
```