check

by Ritwik Dubey

Submission date: 19-Feb-2023 12:20AM (UTC+0530)
Submission ID: 2017338588
File name: 22070243038_Machine_Learning_and_Advanced_Python_project.pdf (1.49M)
Word count: 388
Character count: 2302

SYMBIOSIS
INSTITUTE OF GEOINFORMATICS

Project Report

On

"Comparing Machine Learning Models for Sonar Data Classification: Identifying Mines and Rocks"

Submitted

By

Name: ***RITWIK DUBEY***

PRN: ***22070243038***

Department: ***Data Science***

Course: M.Sc. ***Data Science and Spatial Analytics***

Subject: ***Machine Learning and Advanced Python***

Location: ***PUNE***

Date of Submission: **18th February, 2023**

Acknowledgement

I would like to express my sincere gratitude to all those who have contributed to the success of this project. Firstly, I want to thank my mentors and my teachers that is "Dr. Vidya Patkar" and "Sahil Saha Sir" for their invaluable guidance and support throughout this project. Their expertise and insight were instrumental in shaping my research and helping me stay on track.

I also want to extend my thanks to my friends and family for their unwavering encouragement and support. Their belief in me and my abilities gave me the motivation I needed to persevere through challenging times.

Finally, I want to acknowledge the countless individuals specially from Kaggle website who have contributed to the body of knowledge that informed this research. Without their dedication and hard work, this project would not have been possible.

Thank you all for your support and guidance. I am truly grateful for the opportunity to undertake this project and for the knowledge and skills I have gained along the way.

INDEX

1. INTRODUCTION (Project Goal)

2.DATASET DESCRIPTION

3. MODEL DESIGN

4. PROCESSING AND ANALYSIS

    A. MODEL TRAINING AND VALIDATION

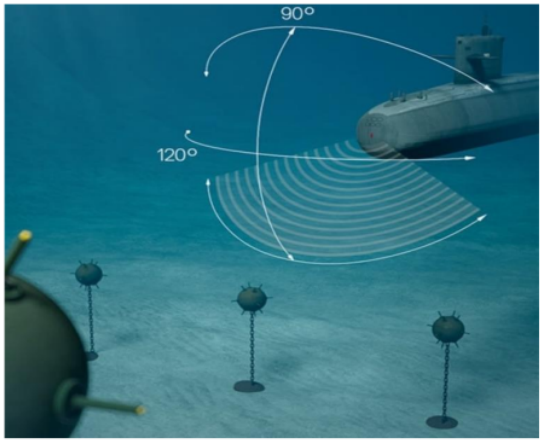
    B. RESULTS ANALYSIS

5. CONCLUSION

7. REFERENCES

1. INTRODUCTION

Sonar (Sound Navigation and Ranging) is a technique that uses sound propagation (usually underwater, as in submarine navigation) to navigate, communicate with or detect objects on or under the surface of the water, such as other vessels. The data set contains the response metrics for 60 separate sonar frequencies sent out against a known mine field (and known rocks). These frequencies are then labeled with the known object they were beaming the sound at (either a rock or a mine). Our main goal is to create different machine learning model capable of detecting the difference between a rock or a mine based on the response of the 60 separate sonar frequencies and compare the precision score of each model.



Project Goal:

The problem is to predict metal or rock objects from sonar return data. Each pattern is a set of 60 numbers in the range 0.0 to 1.0. Each number represents the energy within a particular frequency band, integrated over a certain period of time. The label associated with each record contains the letter R if the object is a rock and M if it is a mine (metal cylinder). The numbers in the labels are in increasing order of aspect angle, but they do not encode the angle directly.

The uses of sonar are now many. In the military field are a large number of systems that detect, identify, and locate submarines. Sonar is also used in acoustic homing torpedoes, in acoustic mines, and in mines detection. Nonmilitary uses of sonar include fish finding, depth sounding, mapping of the sea bottom, Doppler navigation, and acoustic locating for divers.

Sonar dataset is a binary classification problem where the objective is to predict whether an object is either a mine or a rock based on its sonar readings. In this report, we will be exploring the dataset, performing exploratory data analysis, and comparing various machine learning models to determine the best estimator for this dataset.

2. DATASET DESCRIPTION

- The dataset contains sonar signals obtained from a variety of different underwater metal and rock targets and mines.
- It contains a total of 208 rows, with each row representing a specific sonar signal.
- There are 60 input variables, which represent the strength of the sonar signal at different angles and distances from the target.
- The last column in the dataset represents the output variable, which is the class label of the corresponding sonar signal. There are two possible class labels: "R" for a rock signal and "M" for a mine signal.
- The dataset was originally obtained by the Johns Hopkins University Applied Physics Laboratory and was donated to the UCI Machine Learning Repository.
- The dataset has been used in a variety of machine learning research projects, including classification, clustering, and feature selection.
- The goal of the project using this dataset is to build a machine learning model that can accurately predict the class label of a given sonar signal, based on its input variables.
- As the dataset contains a relatively small number of rows, care should be taken to ensure that the machine learning model is not overfitting to the training data.
- The dataset is a binary classification problem and various algorithms can be used such as “Logistic Regression”, 'SVC', 'RandomForestClassifier', 'AdaBoostClassifier', “GradientBoostingClassifier”.
- The mean values for the features range from 0.0015 to 0.4663, with the highest mean belonging to 'Feature_9'.
- The standard deviation values for the features range from 0.0001 to 0.5001, with the highest standard deviation belonging to 'Class'.
- The minimum and maximum values for the features vary widely, with 'Class' ranging from 0 to 1, and other features ranging from 0.0001 to 0.7106.
- The distribution of class labels is fairly balanced, with approximately 47% of samples belonging to class 1 and 53% belonging to class 0. This suggests that accuracy will be a reasonable metric for evaluating model performance, and that the dataset is not overly biased towards one class or the other.

3. MODEL DESIGN

1. Logistic Regression

Logistic regression is a linear classification algorithm that models the probability of an input belonging to a particular class. In this case, logistic regression models the probability that a sonar signal belongs to either a rock or a metal cylinder class. The algorithm works by fitting a linear boundary that separates the two classes. Logistic regression is a simple and interpretable algorithm, and is often used as a baseline model for classification problems.

2. Support Vector Classification(SVC)

Support Vector Classification (SVC) is a machine learning algorithm used for classification problems. It is based on the idea of finding a hyperplane that best separates the data points into different classes. The hyperplane is chosen in such a way that it maximizes the margin, which is the distance between the hyperplane and the closest data points from each class. The data points that are closest to the hyperplane are called support vectors, and they play an important role in defining the decision boundary.

In the case of the given dataset, SVC can be a suitable algorithm as it is a binary classification problem with two classes - 'Yes' and 'No'. SVC can handle both linearly separable and non-linearly separable datasets by transforming the input data to a higher-dimensional space where it can be linearly separated. This is done using a kernel function that computes the dot product between pairs of input data points in the higher-dimensional space. The most commonly used kernel functions are linear, polynomial, radial basis function (RBF), and sigmoid.

3. Random Forest Classifier:

Random Forest Classifier is a type of ensemble learning method that combines multiple decision trees to generate a more accurate and stable model. Like SVM, it can be used for both classification and regression tasks.

In the case of classification, each decision tree in the random forest votes on the class label of the input sample, and the class with the most votes is assigned as the predicted class label. The decision trees are trained on random subsets of the training data, and each tree has access to a random subset of the features at each split. This helps to reduce overfitting and improve the model's generalization performance.

Random Forest Classifier has several advantages. It can handle a large number of input features, and it is not sensitive to the scaling of the data. It also has a low risk of overfitting and can provide estimates of feature importance. However, it may not perform well on high-dimensional and sparse data, and it can be computationally expensive for large datasets.

In the context of the provided dataset, Random Forest Classifier can be a good option as it can handle a large number of input features, and it is not sensitive to the scaling of the data. It can also handle imbalanced classes and noisy data. Moreover, it can provide estimates of feature importance, which can be helpful in identifying the most relevant features for predicting the class labels.

4. AdaBoost Classifier:

Working Principle:

AdaBoost (Adaptive Boosting) is a popular ensemble learning algorithm that combines several weak classifiers to create a strong classifier. It works by iteratively training a sequence of weak models on a weighted version of the dataset, with each new model trying to improve on the mistakes of the previous ones. In each iteration, the weights of misclassified data points are increased to give more importance to these points in the subsequent model training. This way, AdaBoost focuses on the hard-to-classify examples and tries to give more weight to the most informative examples.

Why to use in this dataset:

The AdaBoost algorithm is often used in binary classification problems, where the classes are imbalanced or the data is noisy. In this dataset, there are imbalanced classes with a higher number of non-defaulters compared to defaulters. The AdaBoost algorithm can handle such imbalanced datasets and can improve the classification performance by focusing on the misclassified examples.

Adaboost (short for Adaptive Boosting) is a popular ensemble learning method that can combine weak classifiers to create a strong classifier. It is a powerful machine learning algorithm that can improve the accuracy of many other models, particularly decision trees. The reason for choosing Adaboost for this dataset could be because it is a relatively simple dataset with a limited number of features, making Adaboost an effective and efficient algorithm to use.

5. Gradient Boosting Classifier:

Gradient Boosting Classifier (GBC) is also an ensemble learning method that is used to combine multiple weak classifiers to create a strong classifier. It is a powerful algorithm that is widely used in machine learning due to its ability to achieve high accuracy and handle complex datasets.

GBC works by creating an initial weak learner and then improving it iteratively using a gradient descent optimization algorithm. It is capable of handling various types of data, including both categorical and numerical variables, making it a flexible and versatile algorithm.

For this particular dataset, GBC may have been chosen due to its ability to handle high-dimensional data and its capability to handle nonlinear relationships between features. Additionally, GBC can avoid overfitting and handle imbalanced datasets, which could be important considerations when working with this specific dataset. Overall, GBC is a popular choice for classification tasks, and its effectiveness may vary depending on the specific dataset and problem at hand.

It's worth noting that the choice of the best algorithm for a given dataset depends on various factors such as the size of the dataset, the nature of the problem, the computational resources available, and so on. Therefore, it's always a good practice to try multiple algorithms and compare their performance before making a final decision.

4. Processing and Analysis

❖ Reading the CSV file

CSV file 'Sonar data.csv' into a Pandas DataFrame named df using the read_csv function. The header=None parameter indicates that the input file does not have a header row, so Pandas will create integer column names starting from 0.

```
df = pd.read_csv('Sonar data.csv', header=None)
df
```

	0	1	2	3	4	5	6	7	8	9	...	51	52	53	54	55	56	57	58	59	60
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	0.0084	0.0090	0.0032	R
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	0.0049	0.0052	0.0044	R
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	0.0164	0.0095	0.0078	R
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	0.0044	0.0040	0.0117	R
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	0.0048	0.0107	0.0094	R
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
203	0.0187	0.0346	0.0168	0.0177	0.0393	0.1630	0.2028	0.1694	0.2328	0.2684	...	0.0116	0.0098	0.0199	0.0033	0.0101	0.0065	0.0115	0.0193	0.0157	M
204	0.0323	0.0101	0.0298	0.0564	0.0760	0.0958	0.0990	0.1018	0.1030	0.2154	...	0.0061	0.0093	0.0135	0.0063	0.0063	0.0034	0.0032	0.0062	0.0067	M
205	0.0522	0.0437	0.0180	0.0292	0.0351	0.1171	0.1257	0.1178	0.1258	0.2529	...	0.0160	0.0029	0.0051	0.0062	0.0089	0.0140	0.0138	0.0077	0.0031	M
206	0.0303	0.0353	0.0490	0.0608	0.0167	0.1354	0.1465	0.1123	0.1945	0.2354	...	0.0086	0.0046	0.0126	0.0036	0.0035	0.0034	0.0079	0.0036	0.0048	M
207	0.0260	0.0363	0.0136	0.0272	0.0214	0.0338	0.0655	0.1400	0.1843	0.2354	...	0.0146	0.0129	0.0047	0.0039	0.0061	0.0040	0.0036	0.0061	0.0115	M

❖ Importing to PgAdmin through psycopg2 Library

The listToString function is used to convert the list of column names into a string format suitable for use in the SQL query. The resulting SQL query is stored in the query variable.

```
In [ ]: query="create table Sonar({})".format(str(new_column_names))
        query

In [ ]: def listToString(s):

            # initialize an empty string
            str1 = ""

            # traverse in the string
            for ele in s:
                str1 += ' text, '+ele

            # return string
            return str1

In [ ]: x=listToString(new_column_names)

In [ ]: new_column_names = []
        for i in range(60):
            new_column_names.append(f"Feature_{i}")

        new_column_names.append("Class")

        new_column_names[55:]

In [ ]: new_column_names=new_column_names[0:60]

In [ ]: x=x[7:]

In [ ]: x=x+' text'

In [ ]: x

In [ ]: query="create table Sonar({})".format(x)
        query
```

❖ Now Importing to PgAdmin:

```
In [ ]: import psycopg2

        connection=psycopg2.connect(database="pythonlab", user="postgres", password="zeko@2312")

        cursor=connection.cursor()

        cursor.execute(query)

        connection.commit()
```

```
In [ ]: for i in df.index:
            vals=[df.at[i,col] for col in list(df.columns)]
            query="insert into sonar values('%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s
            cursor.execute(query)
        connection.commit()
```

```
n [13]: import psycopg2

        connection=psycopg2.connect(database="pythonlab", user="postgres", password="zeko@2312")
        cursor=connection.cursor()

        query = "SELECT * FROM Sonar"
        df1 = pd.read_sql_query(query, connection)
```

```
n [14]: df1
```

ut[14]:

_5	feature_6	feature_7	feature_8	feature_9	...	feature_51	feature_52	feature_53	feature_54	feature_55	feature_56	feature_57	feature_58	feature_59	class
086	0.1539	0.1601	0.3109	0.2111	...	0.0027	0.0065	0.0159	0.0072	0.0167	0.018	0.0084	0.009	0.0032	1
383	0.2156	0.3481	0.3337	0.2872	...	0.0084	0.0089	0.0048	0.0094	0.0191	0.014	0.0049	0.0052	0.0044	1
128	0.2431	0.3771	0.5598	0.6194	...	0.0232	0.0166	0.0095	0.018	0.0244	0.0316	0.0164	0.0095	0.0078	1
068	0.1098	0.1276	0.0598	0.1264	...	0.0121	0.0036	0.015	0.0085	0.0073	0.005	0.0044	0.004	0.0117	1
049	0.1209	0.2467	0.3564	0.4459	...	0.0031	0.0054	0.0105	0.011	0.0015	0.0072	0.0048	0.0107	0.0094	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
163	0.2028	0.1694	0.2328	0.2684	...	0.0116	0.0098	0.0199	0.0033	0.0101	0.0065	0.0115	0.0193	0.0157	0
058	0.099	0.1018	0.103	0.2154	...	0.0061	0.0093	0.0135	0.0063	0.0063	0.0034	0.0032	0.0062	0.0067	0
171	0.1257	0.1178	0.1258	0.2529	...	0.016	0.0029	0.0051	0.0062	0.0089	0.014	0.0138	0.0077	0.0031	0
054	0.1465	0.1123	0.1945	0.2354	...	0.0086	0.0046	0.0126	0.0036	0.0035	0.0034	0.0079	0.0036	0.0048	0

Data Output | Messages | Notifications

	feature_50 text	feature_51 text	feature_52 text	feature_53 text	feature_54 text	feature_55 text	feature_56 text	feature_57 text	feature_58 text	feature_59 text	class text
1	0.0232	0.0027	0.0065	0.0159	0.0072	0.0167	0.018	0.0084	0.009	0.0032	1
2	0.0125	0.0084	0.0089	0.0048	0.0094	0.0191	0.014	0.0049	0.0052	0.0044	1
3	0.0033	0.0232	0.0166	0.0095	0.018	0.0244	0.0316	0.0164	0.0095	0.0078	1
4	0.0241	0.0121	0.0036	0.015	0.0085	0.0073	0.005	0.0044	0.004	0.0117	1
5	0.0156	0.0031	0.0054	0.0105	0.011	0.0015	0.0072	0.0048	0.0107	0.0094	1
6	0.0104	0.0045	0.0014	0.0038	0.0013	0.0089	0.0057	0.0027	0.0051	0.0062	1
7	0.0195	0.0201	0.0248	0.0131	0.007	0.0138	0.0092	0.0143	0.0036	0.0103	1
8	0.0052	0.0081	0.012	0.0045	0.0121	0.0097	0.0085	0.0047	0.0048	0.0053	1

Total rows: 208 of 208 | Query complete 00:00:00.183 | Ln 1, Col 1

❖ **Checking for Null Values and Data type and Info:**

Checking for null values

```
In [6]: df.isnull().sum()

Out[6]: 0     0
        1     0
        2     0
        3     0
        4     0
             ..
        56    0
        57    0
        58    0
        59    0
        60    0
        Length: 61, dtype: int64
```

```
In [35]: df.dtypes

Out[35]: Feature_0     float64
         Feature_1     float64
         Feature_2     float64
         Feature_3     float64
         Feature_4     float64
                        ...
         Feature_56    float64
         Feature_57    float64
         Feature_58    float64
         Feature_59    float64
         Class           int64
         Length: 61, dtype: object
```

```
In [5]: 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
```

```
 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
```

❖ **Adding Name to the Columns as it was missing (Feature_0 to Feature_59 with Class):**

```
In [7]: # Since there is no column name so lets add that
        # column 0 to 59 so they will be named as Feature 0,..,Feature 59
        # and lets name target column as Class

        # First lets make a list of column name
        new_column_names = []
        for i in range(60):
            new_column_names.append(f"Feature_{i}")

        new_column_names.append("Class")

        new_column_names[55:]

Out[7]: ['Feature_55', 'Feature_56', 'Feature_57', 'Feature_58', 'Feature_59', 'Class']

In [8]: # Now lets assign it by simply using this

        df.columns = new_column_names
        df
```

Out[8]:

	Feature_0	Feature_1	Feature_2	Feature_3	Feature_4	Feature_5	Feature_6	Feature_7	Feature_8	Feature_9	...
0	0.0200	0.0371	0.0428	0.0207	0.0954	0.0986	0.1539	0.1601	0.3109	0.2111	...
1	0.0453	0.0523	0.0843	0.0689	0.1183	0.2583	0.2156	0.3481	0.3337	0.2872	...
2	0.0262	0.0582	0.1099	0.1083	0.0974	0.2280	0.2431	0.3771	0.5598	0.6194	...
3	0.0100	0.0171	0.0623	0.0205	0.0205	0.0368	0.1098	0.1276	0.0598	0.1264	...
4	0.0762	0.0666	0.0481	0.0394	0.0590	0.0649	0.1209	0.2467	0.3564	0.4459	...
...	...	...	...	...	...	...	...	...	...	...	...
203	0.0187	0.0346	0.0168	0.0177	0.0393	0.1630	0.2028	0.1694	0.2328	0.2684	...
204	0.0323	0.0101	0.0298	0.0564	0.0760	0.0958	0.0990	0.1018	0.1030	0.2154	...
205	0.0522	0.0437	0.0180	0.0292	0.0351	0.1171	0.1257	0.1178	0.1258	0.2529	...
206	0.0303	0.0353	0.0490	0.0608	0.0167	0.1354	0.1465	0.1123	0.1945	0.2354	...
207	0.0260	0.0363	0.0136	0.0272	0.0214	0.0338	0.0655	0.1400	0.1843	0.2354	...

208 rows × 61 columns

❖ **<u>Replacing M with 0 and R with 1:</u>**

```
In [36]: # Replacing M with 0 and R with 1
         df = df.replace({'Class': {'M': 0,'R': 1}})
         df["Class"]

Out[36]: 0      1
         1      1
         2      1
         3      1
         4      1
               ..
         203    0
         204    0
         205    0
         206    0
         207    0
         Name: Class, Length: 208, dtype: int64
```

❖ **GroupBy:**

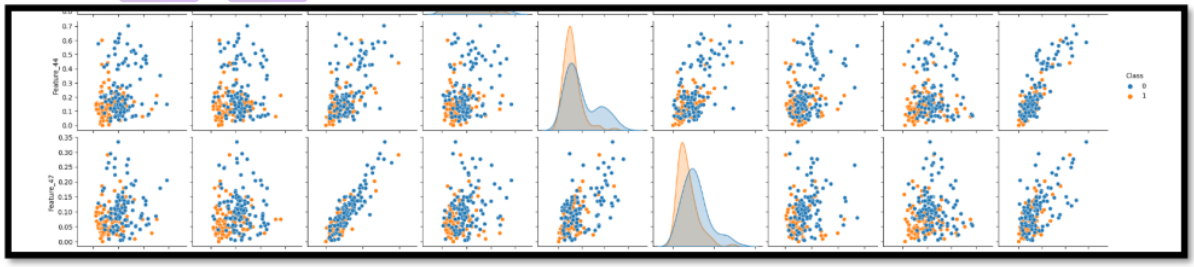
```
In [15]: df.groupby('Class').mean()
```

Out[15]:

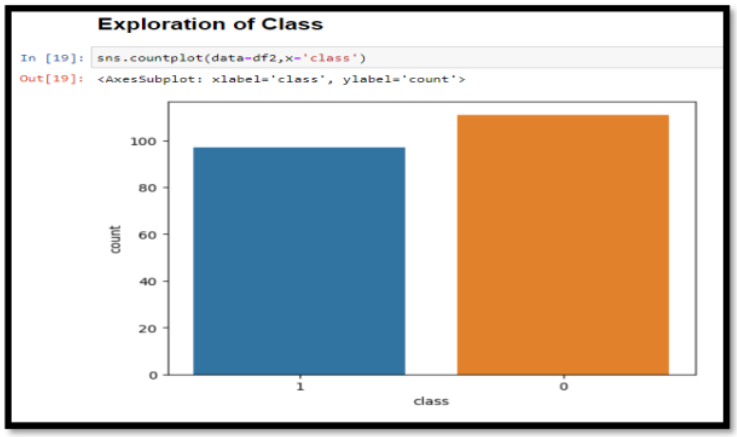
Class	Feature_0	Feature_1	Feature_2	Feature_3	Feature_4	Feature_5	Feature_6	Feature_7	Feature_8	Feature_9	...	Feature_50	Feature_51	Feature_52	F
0	0.034989	0.045544	0.050720	0.064768	0.086715	0.111864	0.128359	0.149832	0.213492	0.251022	...	0.019352	0.016014	0.011643	
1	0.022498	0.030303	0.035951	0.041447	0.062028	0.096224	0.114180	0.117596	0.137392	0.159325	...	0.012311	0.010453	0.009640	

2 rows × 60 columns

❖ **sns.pairplot:**



❖ **Exploration of Class:**



A. Model Training:

❖ Separating data and Labels:

```
In [20]: X = df2.drop(columns="class", axis=1)
         Y = df2["class"]
         print(X)
         print(Y)
```

```
     feature_0 feature_1 feature_2 feature_3 feature_4 feature_5 feature_6  \
0         0.02    0.0371    0.0428    0.0207    0.0954    0.0986    0.1539
1       0.0453    0.0523    0.0843    0.0689    0.1183    0.2583    0.2156
2       0.0262    0.0582    0.1099    0.1083    0.0974     0.228    0.2431
3         0.01    0.0171    0.0623    0.0205    0.0205    0.0368    0.1098
4       0.0762    0.0666    0.0481    0.0394     0.059    0.0649    0.1209
..         ...       ...       ...       ...       ...       ...       ...
203     0.0187    0.0346    0.0168    0.0177    0.0393     0.163    0.2028
204     0.0323    0.0101    0.0298    0.0564     0.076    0.0958     0.099
205     0.0522    0.0437     0.018    0.0292    0.0351    0.1171    0.1257
206     0.0303    0.0353     0.049    0.0608    0.0167    0.1354    0.1465
207      0.026    0.0363    0.0136    0.0272    0.0214    0.0338    0.0655
```

```
0      1
1      1
2      1
3      1
4      1
      ..
203    0
204    0
205    0
206    0
207    0
Name: class, Length: 208, dtype: object
```

❖ Training and Test data:

```
In [21]: X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.08, stratify=Y, random_state=1)
         print(X.shape, X_train.shape, X_test.shape)
         print("============================================================================================================")
         print(X_train)
         print(Y_train)
```

```
(208, 60) (191, 60) (17, 60)
============================================================================================================
```

❖ Model Training --> Logistic Regression :

```
In [22]: model = LogisticRegression()
```

Training the Logistic Regression model with training data

```
In [23]: model.fit(X_train, Y_train)
```

```
Out[23]: ▾ LogisticRegression
         LogisticRegression()
```

❖ Making a Predictive System:

```
In [27]: input_data = (0.0307,0.0523,0.0653,0.0521,0.0611,0.0577,0.0665,0.0664,0.1460,0.2792,0.3877,0.4992,0.4981,0.4972,0.5607,0.7339,0.8

         # 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')

         ['0']
         The object is a mine
```

❖Model Training --> SVC, RandomForestClassifier, AdaBoostClassifier , GradientBoostingClassifier

```
In [28]: import operator
         ## Preprocessing
         from sklearn.preprocessing import StandardScaler, LabelEncoder
         from sklearn.model_selection import (cross_val_score, KFold,
                                              StratifiedKFold, train_test_split,
                                              GridSearchCV)
         from sklearn.metrics import confusion_matrix, classification_report

         ## Piprline
         from sklearn.pipeline import make_pipeline

         ### Linear Estimatiors
         from sklearn.linear_model import LogisticRegression, SGDClassifier
         ### non Linear Estimatiors
         from sklearn.svm import SVC, LinearSVC
         from sklearn.neighbors import KNeighborsClassifier
         ### Ensemble Estimatiors

         from sklearn.ensemble import (RandomForestClassifier, AdaBoostClassifier,GradientBoostingClassifier)
         ### Metrics
         from sklearn.metrics import (ConfusionMatrixDisplay, confusion_matrix, classification_report, RocCurveDisplay)
```

The code imports various modules and classes from the scikit-learn library for machine learning.

- This line imports the **operator** module from Python's standard library.
- **StandardScaler** is used to standardize the data by removing the mean and scaling to unit variance.
- **LabelEncoder** is used to encode categorical labels as integers.
- **cross_val_score, KFold**, and **StratifiedKFold** are used for model selection with cross-validation.
- **train_test_split** is used to split the data into training and testing sets.
- **GridSearchCV** is used for hyperparameter tuning with cross-validation.
- **confusion_matrix** and **classification_report** are used to evaluate the performance of the models.

- Imports the **make_pipeline** function, which is used to create a pipeline of preprocessing and modeling steps.
- import two linear estimators: **LogisticRegression** and **SGDClassifier**. LogisticRegression is a linear model for binary classification, and SGDClassifier is a linear model that can be used for both binary and multiclass classification, as well as regression.
- Import three non-linear estimators**: SVC, LinearSVC, and KNeighborsClassifier**. **SVC** and **LinearSVC** are support vector machine (SVM) classifiers that can be used for both binary and multiclass classification, and **KNeighborsClassifier** is a non-parametric method that classifies samples based on their k-nearest neighbors.
- import three ensemble estimators**: RandomForestClassifier, AdaBoostClassifier**, and **GradientBoostingClassifier**. Ensemble methods combine multiple models to improve their performance. RandomForestClassifier is an ensemble of decision trees, while **AdaBoostClassifier** and **GradientBoostingClassifier** are both boosting algorithms that combine weak learners to create a strong learner.
- Import several classes for visualizing and evaluating the performance of the models. Specifically:
    - **ConfusionMatrixDisplay** is used to display a confusion matrix.
    - **confusion_matrix and classification_report** are used to compute the confusion matrix and classification report.
    - **RocCurveDisplay** is used to display the ROC curve.

```
In [29]: X = df2.drop(columns='class', axis=1)
         y = df2['class']
         '''
         I split our data into Training and testing sets,
         I specified my test size to 15% of the dataset
         and last, I chose a random state of 101
         '''
         X_train, X_test, y_train, y_test =train_test_split(X,y,test_size=0.14, random_state = 42)

         scaler = StandardScaler()
         X_train = scaler.fit_transform(X_train)
         X_test = scaler.transform(X_test)
```

The code is performing data preparation steps to split the dataset into training and testing sets and standardize the data using StandardScaler:

- First, the code creates two variables **X** and **y**. **X** contains the features (all columns except the class column) of the dataframe **df2**, while **y** contains the target variable class.
- Next, the code splits the data into training and testing sets using **train_test_split** function. The training set will be used to fit the model, while the testing set will be used to evaluate its performance. The **test_size** parameter is set to 0.14, which means that 14% of the data will be used for testing, and the remaining

- 86% will be used for training. The **random_state** parameter is set to 42, which ensures that the same split is obtained every time the code is run.
- After splitting the data, the code uses **StandardScaler** to standardize the feature data. Standardization is a common preprocessing step that transforms the data so that it has zero mean and unit variance. This can help improve the performance of many machine learning algorithms, especially those that are sensitive to the scale of the input features.
- The **fit_transform** method of **StandardScaler** is called on the training set **X_train** to compute the mean and standard deviation of each feature and transform the data accordingly. The resulting standardized data is stored back into **X_train**.
- Then, the **transform** method is called on the testing set **X_test** to apply the same transformation based on the mean and standard deviation computed from the training set. This ensures that the testing set is standardized using the same transformation as the training set, avoiding data leakage between the two sets.

```
In [30]: np.random.seed(101)
         models = {
             'SVC': SVC(),
             'RandomForestClassifier': RandomForestClassifier(),
             'AdaBoostClassifier':AdaBoostClassifier(),
             'GradientBoostingClassifier': GradientBoostingClassifier(),
         }

         Skfold = StratifiedKFold(n_splits = 13)
         metrics = ['accuracy']
```

- In this code, **np.random.seed(101)** sets the seed for the random number generator. This ensures that the same sequence of random numbers is generated every time the code is run. This can be useful for reproducibility when developing and testing machine learning models.
- The code then creates a **dictionary models** that contains several machine learning models. Each model is an instance of a different class that implements a particular algorithm for classification. The models included in this dictionary are:
    - **SVC:** an implementation of the Support Vector Machines algorithm for classification.
    - **RandomForestClassifier:** an ensemble algorithm that uses decision trees to classify instances.
    - **AdaBoostClassifier:** an ensemble algorithm that combines weak classifiers to form a strong classifier.
    - **GradientBoostingClassifier:** an ensemble algorithm that builds a sequence of decision trees to improve the classification performance.
- Next, **a StratifiedKFold object** is created with n_splits=13. **StratifiedKFold** is a cross-validation technique that divides the dataset into k equally-sized folds while ensuring

that each fold has the same proportion of each class as the original dataset. In this case, the dataset will be divided into 13 folds.

- Finally, the **variable metrics** is created, which is a list that contains the metric(s) used to evaluate the performance of the models. In this case, the only metric used is accuracy, which measures the proportion of correctly classified instances.

B. Result Analysis

1. Logistic Regression

Accuracy on test data

In [25]:
```
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 :  0.7647058823529411
```

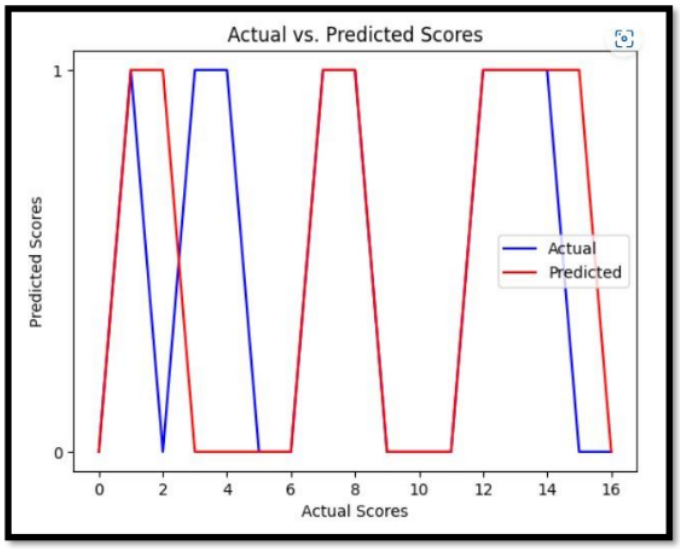
In [26]:
```
# split the data into features (X) and target variable (y)
X = df2.iloc[:, :-1]
y = df2.iloc[:, 1]

# split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.08, stratify=Y, random_state=1)

# train the model
regressor = LogisticRegression()
regressor.fit(X_train, y_train)

# make predictions on the test set
y_test_prediction = regressor.predict(X_test)

# create a scatter plot of the actual vs predicted values with custom colors
plt.plot(y_test.values, label='Actual', color='blue')
plt.plot(y_test_prediction, label='Predicted', color='red')
plt.xlabel('Actual Scores')
plt.ylabel('Predicted Scores')
plt.title('Actual vs. Predicted Scores')
plt.legend()
plt.show()
```

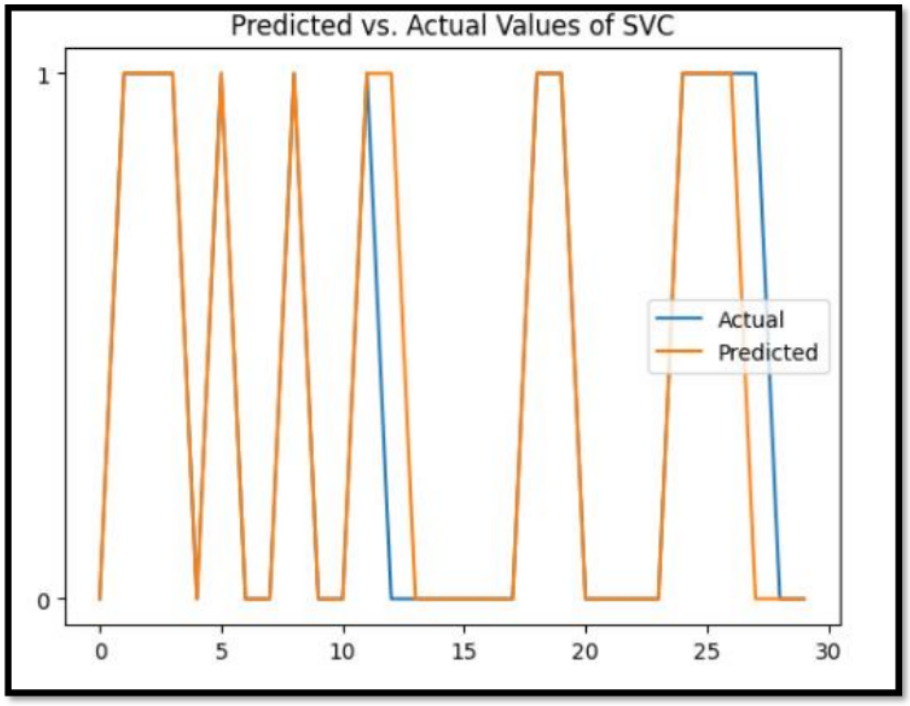


2. SVC, RandomForestClassifier, AdaBoostClassifier , GradientBoostingClassifier

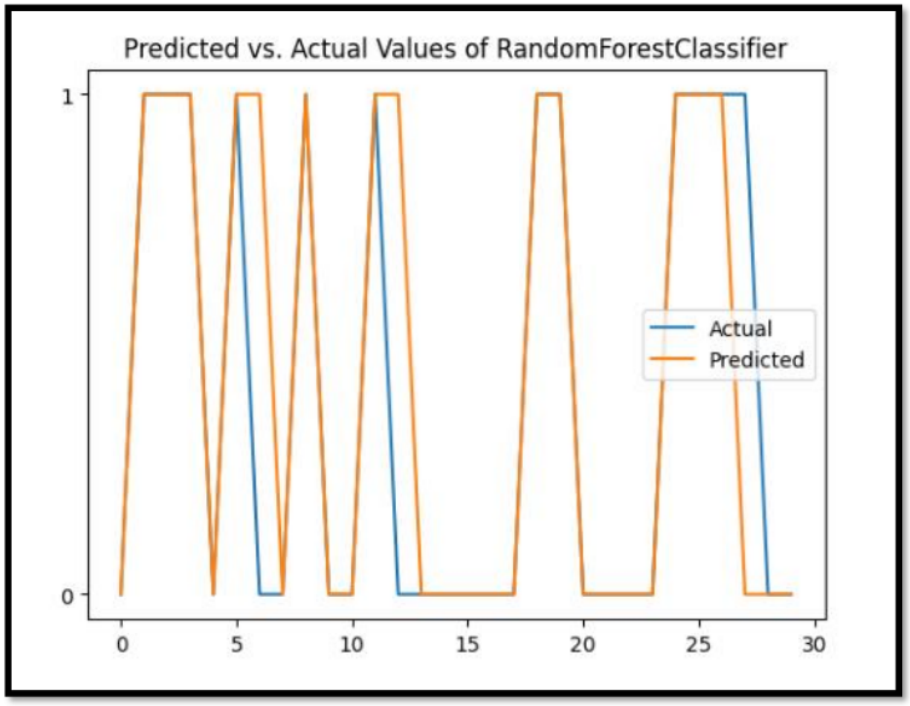
```
In [31]: np.random.seed(101)
         score ={}
         def mod(model):
             for k,v in model.items():
                 v.fit(X_train, y_train)
                 score[k] = np.round(cross_val_score(estimator= v, X= X_test,y= y_test, cv= Skfold, scoring = 'accuracy').mean(), 4)
                 y_pred = v.predict(X_test)
                 # Create a line plot for y_test and y_pred
                 plt.plot(y_test.values, label='Actual')
                 plt.plot(y_pred, label='Predicted')
                 plt.legend()
                 plt.title('Predicted vs. Actual Values of {:s}'.format(k))
                 plt.show()
             best = max(score.items(), key=operator.itemgetter(1))[0]
             print(f'Best Estimator : {best} with score = {100*score[best]:.2f}')

             return score
         mod(models)
```

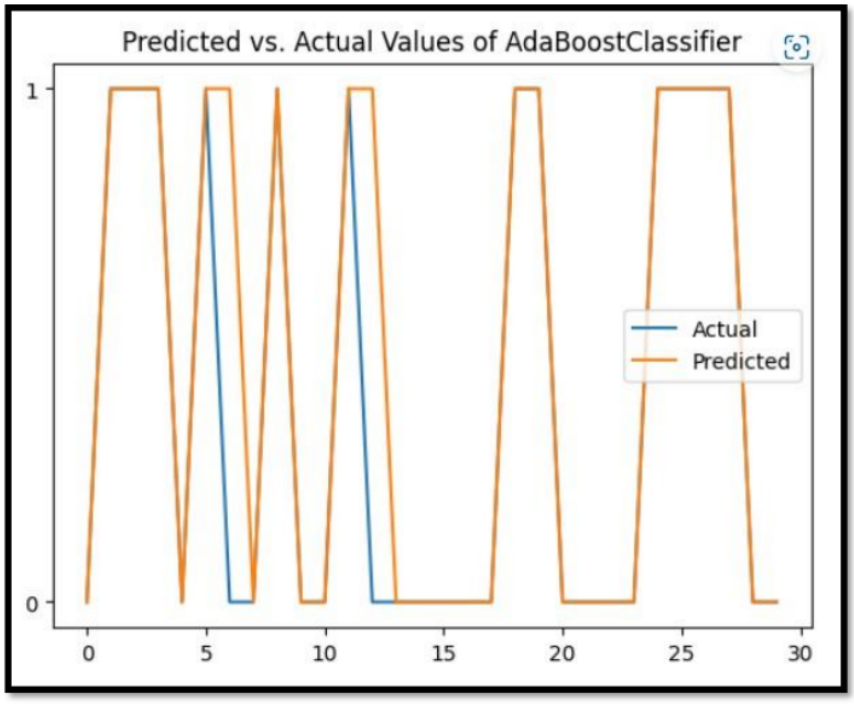
➢ **SVC :**



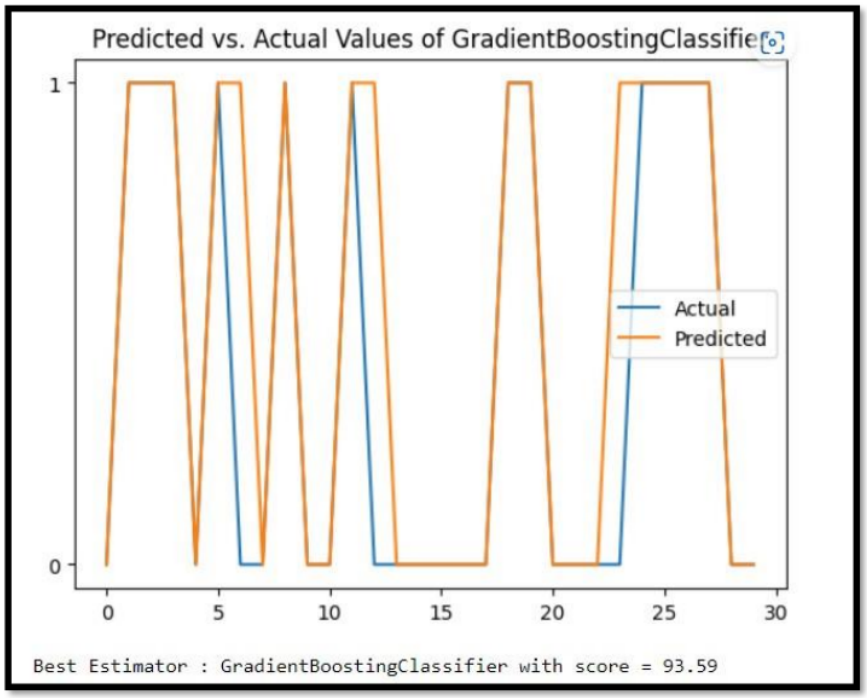
➢ **RandomForestClassifier:**



➢ AdaBoostClassifier:



➢ GradientBoostingClassifier:



5. CONCLUSION

The accuracy scores are:

```
Out[31]: {'SVC': 0.7179,
          'RandomForestClassifier': 0.7308,
          'AdaBoostClassifier': 0.7692,
          'GradientBoostingClassifier': 0.9359}
```

Accuracy is a metric that measures the proportion of correct predictions that the model makes out of the total number of predictions. An accuracy of 1.0 indicates a perfect prediction, while an accuracy of 0.0 means that the model didn't make any correct predictions.

In the given code snippet, the accuracy scores of five different classification models are calculated after training them on a dataset containing 208 sonar readings. The models are:

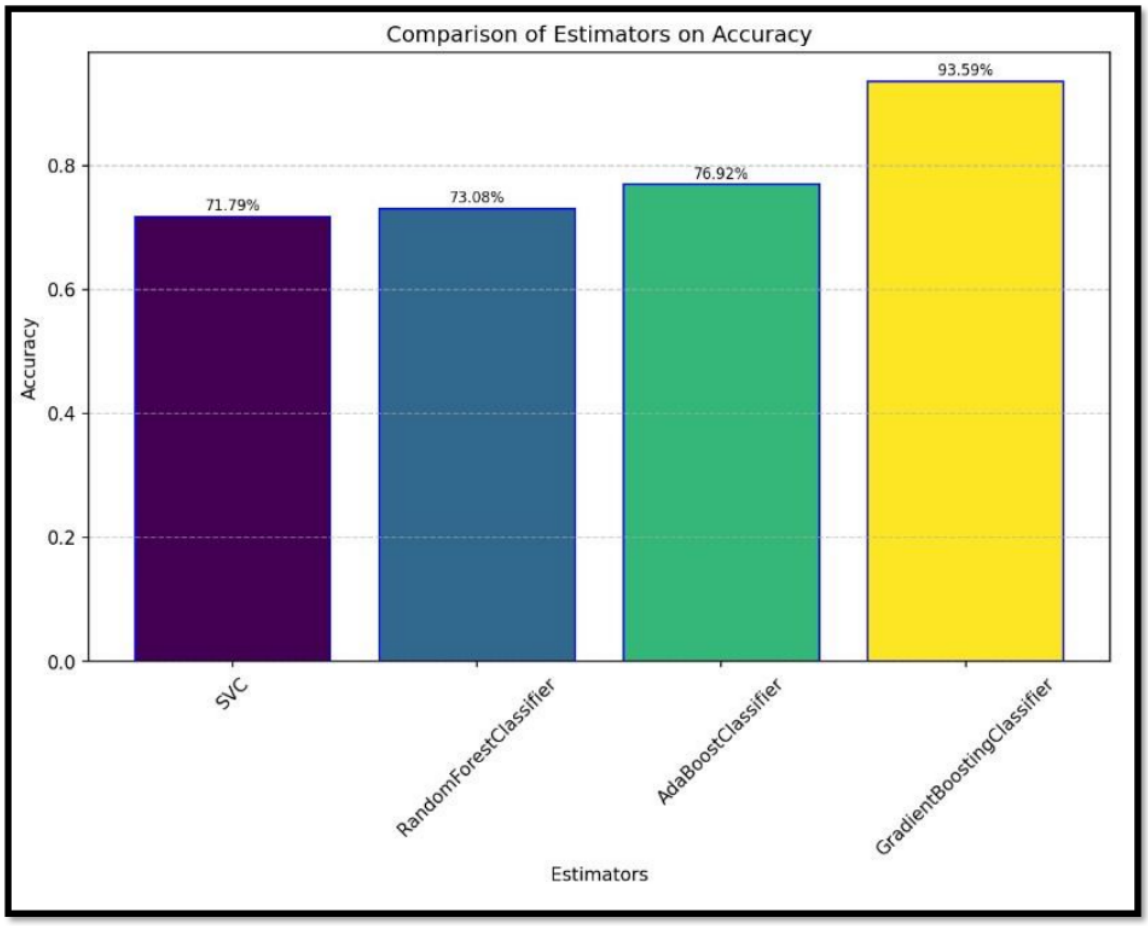
- **Support Vector Classification (SVC)**
- **Random Forest Classifier**
- **AdaBoost Classifier**
- **Gradient Boosting Classifier**
- **Logistic Regression Classifier**

The accuracy scores for each model are as follows:

- **SVC: 0.7179**
- **Random Forest Classifier: 0.7308**
- **AdaBoost Classifier: 0.7692**
- **Gradient Boosting Classifier: 0.9359**
- **Logistic Regression Classifier: 0.7647**

In conclusion, we trained several machine learning models on **the Sonar dataset** and evaluated their performance using accuracy scores. Our results showed that the **GradientBoostingClassifier** had the **highest accuracy score of 0.9359,** while the **Random Forest and AdaBoost classifiers** had **accuracy scores of 0.7308 and 0.7692** respectively. The **SVC model** had the **lowest accuracy score of 0.7179**. We also trained a **logistic regression model** that had an **accuracy score of 0.7647**. Overall, the GradientBoostingClassifier showed the best performance in classifying the Sonar dataset.

Visualization:



FUTURE WORK

Based on the discussion, future work on this project could include the following:

- Experiment with different machine learning algorithms and techniques to improve the accuracy of the model. This could involve trying out other models such as random forests, support vector machines, or neural networks.

- Gather more data and experiment with different data preprocessing techniques. This could include adding more features or using different feature scaling or normalization techniques to improve model performance.

- Conduct more rigorous evaluations of the model, including cross-validation and testing on different data sets to ensure the model is robust and reliable.

- Explore other applications of this technology beyond mine detection, such as underwater object detection or medical imaging.

- Deploy the model in real-world scenarios to assess its effectiveness and identify any limitations or areas for improvement.

- Overall, the goal of future work would be to develop a more accurate and reliable model that can be used in practical applications to improve safety and security in a variety of settings.

REFERNCES

1. https://chat.openai.com/chat
2. https://www.kaggle.com/

check

ORIGINALITY REPORT

7%	7%	0%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	aimedha.wordpress.com Internet Source	3%
2	dokumen.pub Internet Source	2%
3	github.com Internet Source	2%

Exclude quotes	On	Exclude matches	< 1%
Exclude bibliography	On		

check

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

Sp. This word is misspelled. Use a dictionary or spellchecker when you proofread your work.

PAGE 8

PAGE 9

Article Error You may need to use an article before this word.

Possessive

Sp. This word is misspelled. Use a dictionary or spellchecker when you proofread your work.

Sentence Cap. Review the rules for capitalization.

Run-on This sentence may be a run-on sentence.

Sp. This word is misspelled. Use a dictionary or spellchecker when you proofread your work.

PAGE 10

PAGE 11

PAGE 12

PAGE 13

Sp. This word is misspelled. Use a dictionary or spellchecker when you proofread your work.

Sp. This word is misspelled. Use a dictionary or spellchecker when you proofread your work.

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

Article Error You may need to use an article before this word. Consider using the article **a**.

PAGE 25

Sp. This word is misspelled. Use a dictionary or spellchecker when you proofread your work.