Project Title: Predictive Modeling for Early Detection of Liver Disease in Patients

Phase 2 : Predictive modeling

Name: Ratnak Saha

Introduction

The first phase of our project involved analyzing and preprocessing the Indian Liver Patient Dataset (ILPD) from the UCI Machine Learning Repository. There are 583 patient records in the dataset, each of which contains 10 features related to liver health, such as age, gender, and various biochemical markers. To begin with, we cleaned the data to remove missing values, outliers, and inconsistencies. The "Albumin and Globulin Ratio" was imputed with the mean to ensure data integrity.

The second phase of the report required to create predictive models for liver disease. A number of classification algorithms were explored, including Logistic Regression, K-Nearest Neighbors (KNN), Decision Trees, Random Forests, XGBoost, LightGBM and Neural Networks. Model performance was measured by accuracy, precision, recall, F1-score, and AUC (Area Under the Curve). In addition to Logistic Regression, ensemble methods like the Voting Classifier outperformed individual models by combining their strengths. Among the models we evaluated, the Voting Classifier achieved the highest test accuracy (0.771) and AUC (0.777). To mitigate overfitting risk, cross-validation was used to ensure robust model evaluation. This required dividing the dataset into many folds and training/testing the models on diverse subsets, resulting in a more accurate evaluation of their performance.

The ideas and discoveries from Phase 1, particularly the identification of essential biochemical markers and the performance of multiple models via cross-validation, influenced the direction of Phase 2. In Phase 2, we worked on refining and optimizing the models, resolving class imbalance to an extent, and improving interpretability.

In addition to regular model training, Grid Search was used to fine-tune hyperparameters for various models, most notably the KNN, Random Forest and LightGBM classifier. Grid Search methodically evaluated a variety of hyperparameter combinations to determine the optimal settings for improving model performance. For example, refined parameters was used such as the number of estimators, maximum depth, and minimum sample split, resulting in improved model accuracy and robustness.

Report Overview:

In Phase 2 of the research, emphasis were focused on developing predictive models for the early diagnosis of liver illness, expanding on the foundation created in Phase 1. This phase consisted of numerous critical components, including improved data preprocessing and feature engineering techniques to improve model performance. Machine learning algorithms such as KNN, logistic regression, decision trees, random forests, XGBoost, and LightGBM were carefully selected and fine-tuned, with techniques such as grid search and cross-validation used to optimize hyperparameters and assess generalization capabilities.

Ensemble techniques, specifically the Voting Classifier, were used to aggregate predictions from multiple models in order to improve overall performance and reduce overfitting. A comprehensive model evaluation was performed, which included an examination of performance metrics such as accuracy, precision, recall, and F1-score, as well as the interpretation of confusion matrices, ROC curves, and classification reports. The insights generated from model interpretations were thoroughly examined, yielding substantial recommendations for using predictive models in clinical contexts and directing future research.

Overview of Methodology:

The predictive modeling methodology used in this project consists of multiple consecutive processes aimed at generating accurate models for the early detection of liver illness:

Data Preprocessing: The initial step is to analyze and preprocess the Indian Liver Patient Dataset (ILPD) received from the UCI Machine Learning Repository. This includes dealing with missing values, outliers, and irregularities in the dataset to maintain data integrity. Furthermore, feature engineering techniques may be used to create new features or change existing ones in order to improve the predictive power of the models.

Feature Selection: Following data preprocessing, feature selection techniques are used to discover the most important features for modeling. To pick the subset of characteristics with the highest predictive power, employ techniques such as Select K Best with F-score or correlation analysis.

Model Selection: Following feature selection, a number of classification algorithms are used to construct predictive models. Common techniques include logistic regression, K-nearest neighbors (KNN), decision trees, random forests, XGBoost, and lightGBM. The algorithms used are determined by the nature of the problem and the dataset's properties.

Hyperparameter Tuning: Each model is trained with a set of hyperparameters, and approaches like Grid Search or Random Search are used to determine the best combination of hyperparameters to maximize model performance. This procedure helps to fine-tune the models and increase their predicted accuracy.

Model Evaluation: Each model's performance is evaluated using a variety of metrics, including accuracy, precision, recall, F1-score, and Area Under the ROC Curve (AUC). Cross-validation approaches such as K-Fold and Stratified K-Fold cross-validation are used to ensure robust evaluation and reduce overfitting.

Ensemble approaches, such as the Voting Classifier, can be used to aggregate predictions from many base models, boosting overall performance while reducing overfitting. This involves combining the predictions of individual models to generate a final prediction.

Finally, the models are interpreted in order to get insights into their decision-making procedures. To further comprehend the elements influencing the model's predictions, interpretability approaches like as feature importances, confusion matrices, and ROC curves are used. These findings are examined and utilized to make recommendations for employing predictive models in clinical settings and for future study.

In [1]:
```
from sklearn.model_selection import RandomizedSearchCV,GridSearchCV, train_test_
from sklearn.pipeline import Pipeline
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tabulate import tabulate
from IPython.display import display, HTML
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder
import numpy as np
from sklearn.metrics import accuracy_score, confusion_matrix, classification_rep
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.ensemble import VotingClassifier, RandomForestClassifier, StackingC
from sklearn.linear_model import LogisticRegression
import warnings
warnings.filterwarnings("ignore")
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn import preprocessing
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn import feature_selection as fs
from sklearn_pandas import DataFrameMapper
from sklearn.impute import SimpleImputer
import lightgbm as lgbm
from lightgbm import LGBMClassifier
import xgboost as xgb
from imblearn.under_sampling import RandomUnderSampler
from keras.models import Sequential
from keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.callbacks import EarlyStopping
import seaborn as sns
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

In [2]:
```
ILPD = pd.read_csv("Indian_Liver_Patient_Dataset.csv")
print(ILPD.head(n=10))
print(ILPD.info())
```

```
   Age  Gender  Total Bilirubin  Direct Bilirubin  Alkaline Phosphotase  \
0   65  Female              0.7               0.1                   187   
1   62    Male             10.9               5.5                   699   
2   62    Male              7.3               4.1                   490   
3   58    Male              1.0               0.4                   182   
4   72    Male              3.9               2.0                   195   
5   46    Male              1.8               0.7                   208   
6   26  Female              0.9               0.2                   154   
7   29  Female              0.9               0.3                   202   
8   17    Male              0.9               0.3                   202   
9   55    Male              0.7               0.2                   290   

   Alamine Aminotransferase  Aspartate Aminotransferase  Total Proteins  \
0                        16                          18             6.8   
1                        64                         100             7.5   
2                        60                          68             7.0   
3                        14                          20             6.8   
4                        27                          59             7.3   
5                        19                          14             7.6   
6                        16                          12             7.0   
7                        14                          11             6.7   
8                        22                          19             7.4   
9                        53                          58             6.8   

   Albumin  Albumin and Globulin Ratio  Selector  
0      3.3                        0.90         1  
1      3.2                        0.74         1  
2      3.3                        0.89         1  
3      3.4                        1.00         1  
4      2.4                        0.40         1  
5      4.4                        1.30         1  
6      3.5                        1.00         1  
7      3.6                        1.10         1  
8      4.1                        1.20         2  
9      3.4                        1.00         1  
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 583 entries, 0 to 582
Data columns (total 11 columns):
 #   Column                      Non-Null Count  Dtype  
---  ------                      --------------  -----  
 0   Age                         583 non-null    int64  
 1   Gender                      583 non-null    object 
 2   Total Bilirubin             583 non-null    float64
 3   Direct Bilirubin            583 non-null    float64
 4   Alkaline Phosphotase        583 non-null    int64  
 5   Alamine Aminotransferase    583 non-null    int64  
 6   Aspartate Aminotransferase  583 non-null    int64  
 7   Total Proteins              583 non-null    float64
 8   Albumin                     583 non-null    float64
 9   Albumin and Globulin Ratio  579 non-null    float64
 10  Selector                    583 non-null    int64  
dtypes: float64(5), int64(5), object(1)
memory usage: 50.2+ KB
None
```

In [3]:

```
# making columns lower cases
ILPD.columns = ILPD.columns.str.lower().str.strip()
column_names = {
'age':'Age',
'gender': 'Gender',
```

```
'total bilirubin': 'TB',
'direct bilirubin' :'DB',
'alkaline phosphotase': 'Alkphos',
'alamine aminotransferase': "Sgpt" ,
'aspartate aminotransferase':"Sgot",
'total proteins':"TP",
'albumin': "ALB",
'albumin and globulin ratio': "A/G Ratio",
'selector': "Selector"
}

ILPD = ILPD.rename(columns = column_names)
print(ILPD.sample(5, random_state=1234))
```

```
     Age  Gender   TB   DB  Alkphos  Sgpt  Sgot   TP  ALB  A/G Ratio  Selector
380   50    Male  1.7  0.8      331    36    53  7.3  3.4        0.9         1
113   74    Male  0.6  0.1      272    24    98  5.0  2.0        0.6         1
301   51  Female  0.9  0.2      280    21    30  6.7  3.2        0.8         1
532   62    Male  0.7  0.2      162    12    17  8.2  3.2        0.6         2
73    52    Male  0.6  0.1      171    22    16  6.6  3.6        1.2         1
```

In [4]:
```
# Checking for data types
print(f"Shape of the dataset = {ILPD.shape} \n")
print(f"Data types are below where 'object' indicates a string type: ")
print(ILPD.dtypes)
```

```
Shape of the dataset = (583, 11)

Data types are below where 'object' indicates a string type:
Age            int64
Gender        object
TB           float64
DB           float64
Alkphos        int64
Sgpt           int64
Sgot           int64
TP           float64
ALB          float64
A/G Ratio    float64
Selector       int64
dtype: object
```

In [5]:
```
for column in ILPD.columns:
    unique_values = ILPD[column].unique()
    print(f"Unique values for column '{column}':{unique_values}")
```

```
Unique values for column 'Age':[65 62 58 72 46 26 29 17 55 57 64 74 61 25 38 33 4
0 51 63 34 20 84 52 30
 48 47 45 42 50 85 35 21 32 31 54 37 66 60 19 75 68 70 49 14 13 18 39 27
 36 24 28 53 15 56 44 41  7 22  8  6  4 43 23 12 69 16 78 11 73 67 10 90]
Unique values for column 'Gender':['Female' 'Male']
Unique values for column 'TB':[ 0.7 10.9  7.3  1.   3.9  1.8  0.9  0.6  2.7  1.1
1.6  2.2  2.9  6.8
  1.9  4.1  6.2  4.   2.6  1.3 14.2  1.4  2.4 18.4  3.1  8.9  0.8  2.8
  2.   5.7  8.6  5.8  5.2  3.8  6.6  0.5  5.3  3.2  1.2 12.7 15.9 18.
 23.  22.7  1.7  3.  11.3  4.7  4.2  3.5  5.9  8.7 11.  11.5  4.5 75.
 22.8 14.1 14.8 10.6  8.   1.5  2.1  6.3  2.3 27.2  2.5  3.6 30.5 16.4
 14.5 18.5 23.2  3.7  3.3  7.1  6.7 22.6  7.5  5.   4.9  8.2  0.4  7.4
 23.3  7.9  3.4 19.8 32.6 17.7 20.  26.3  4.4  9.4 30.8 19.6 15.8  5.5
 20.2 27.7 11.1 10.2 42.8 15.2 16.6 17.3 22.5 16.7  7.7 15.6 12.1 25.
 15. ]
Unique values for column 'DB':[ 0.1  5.5  4.1  0.4  2.   0.7  0.2  0.3  1.3  0.8
0.5  1.   3.   1.9
  1.2  7.8  0.6  1.1  3.2  1.8  8.8  1.6  4.5  2.8  4.   2.7  2.4  1.5
  2.3  3.6  6.2  7.   8.2 11.3 10.2  2.5  1.4  1.7  5.6  2.2  2.1  4.9
  5.   0.9 12.6  7.6  9.   4.6 11.8 14.2  8.9  6.4  9.5  3.3 11.4  4.3
  3.7  2.6  3.9  5.1 12.8 10.4 17.1 14.1  8.5 10.  12.1  2.9  5.2 18.3
  7.2 11.7 10.8  6.1  4.2 19.7  7.7  8.4  6.  13.7]
Unique values for column 'Alkphos':[ 187  699  490  182  195  208  154  202  290
210  260  310  214  145
  183  342  165  293  610  482  542  231  194  289  240  128  188  190
  156  410  374  263  275  168  160  630  415  150  230  176  206  170
  161  253  198  272  175  367  158  259  470  215  239  186  205  171
  162  518 1620  146  670  915   75  148  258  237  269  320  298  538
  238  308  204  282  265  312  243  224  225  486  257  179  661 1580
 1630  280  300  178  177  201  802  248 1896  512  199 1110  380  159
  332  189  392  286  180  218  462  196  750 1050  599  292  962  950
  200 1020  562  386  250  191  614  314  209 1124  664  142  169 1420
  135  163  285  350  220  219  401  100  116  125  147  192  400  120
  173  157 2110  360  316  498  480  680  152  859  901  335  245  505
  228  185  247  348  140  358  110  235  460  262  144  123  575  155
  315  174  340  234  430  588  527  574  106  216   63  302  211  458
  375  405  650  115  621  256  418  271  130  558  326  331  172  105
  102  149  580   92  719  554  555  509  690  862  592  450 1350  246
  166 1750  236  212  279  181 1550 1100  686  309  164  270  137   90
  167  197  226  352  103  850  276  193  805  151  349  365  305  127
  254  108  268  138  466  227  395   97  406  114  153  768  232  390
  356  388  143  251  134  612  515  560  500   98  184]
Unique values for column 'Sgpt':[  16   64   60   14   27   19   22   53   51   3
1   61   91  168   15
  232   17  116   52  875 1680   20   13   45   35   59  102   18   38
  123   33   42   25  407   48   36 1630   39   21   80   86   26   24
   37   40   62   55  166  189   95   12  194   58   28  119  412  404
  220  126  190   97  308   32   29   11   63  181   88   74 2000 1350
 1250  482  322  133   46   57   50   34   72   84   30   70  140   99
   43  378  112   71   23   79  114  118  107  790  950   82   41   56
   85  149  230   69   90   89  148   65  205   96  152  390   10  120
   78  178  179   47  160   54  198   44  349  110  115   94  142  137
  155  157  141  284  440   93   76   49  425  159  622  779  132  154
  196   68  509   67  139  382   75  321  233  173  213  131]
Unique values for column 'Sgot':[  18  100   68   20   59   14   12   11   19   5
8   56   30   41   53
  441   23  245   28   34   66   55   45  731  850   21  111   44   57
   80   36   77   73   50  110   47  576   15  178   27  960  406  150
   61   54   24   16   43   97   86   88   95   26   17  397   29   22
  127   79  142  152   31  350  794  400  202  630  950  161  405   92
```

```
   39   10  116   98  285   64  149 2946 1600 1050  275  113   84   25
   40   83   65 4929   90  140  139   87   38   42  233  138   82   35
   32  187   62   74   67   37  602   63   99  103  145  247  114  104
   51   60 1500   33  180  148   46   13   85  231  156   89  298   48
  130   75  500  105  250  232  143  176   70   52   91  236  108  190
   71  126  141  102   81  511   72  135  497  844  368  188  248  401
   76  221  235  185  230  540  181  155  200  186  623  220   78  348
  125  330  562  384  367  101  168  134   49]
Unique values for column 'TP':[6.8 7.5 7.  7.3 7.6 6.7 7.4 5.9 8.1 5.8 5.5 6.4 4.
3 6.  5.  7.2 3.9 5.2
 4.9 5.6 6.9 6.2 5.1 6.1 6.5 5.7 6.6 6.3 8.  4.4 5.3 4.6 4.7 5.4 7.1 4.
 3.7 2.7 3.  3.8 7.8 4.5 4.1 4.8 7.9 8.5 7.7 8.2 2.8 9.5 9.6 8.3 8.6 8.4
 8.9 8.7 3.6 9.2]
Unique values for column 'ALB':[3.3 3.2 3.4 2.4 4.4 3.5 3.6 4.1 2.7 3.  2.3 3.1
2.6 1.6 3.9 4.  1.9 1.5
 2.9 2.  2.2 2.8 1.8 2.5 2.1 3.7 3.8 4.3 1.7 4.2 4.5 0.9 1.4 4.7 5.5 4.9
 4.6 4.8 5.  1. ]
Unique values for column 'A/G Ratio':[0.9  0.74 0.89 1.   0.4  1.3  1.1  1.2  0.8
0.6  0.87 0.7  0.92 0.55
 0.5  1.85 0.95 1.4  1.18 0.61 1.34 1.39 1.6  1.58 1.25 0.78 0.76 1.55
 0.71 0.62 0.67 0.75 1.16 1.5  1.66 0.96 1.38 0.52 0.47 0.93 0.48 0.58
 0.69 1.27 1.12 1.06 0.53 1.03 0.68  nan 1.9  1.7  1.8  0.3  0.97 0.35
 1.51 0.64 0.45 1.36 0.88 1.09 1.11 1.72 2.8  0.46 0.39 1.02 2.5  0.37]
Unique values for column 'Selector':[1 2]
```

In [6]:
```
ILPD_new = ILPD.drop(columns =["Selector"])
display(HTML('<b> Table 2: Summary of numerical features <b>'))
display(ILPD_new.describe(include=['int64','float64']).T)
```

Table 2: Summary of numerical features

	count	mean	std	min	25%	50%	75%	max
Age	583.0	44.746141	16.189833	4.0	33.0	45.00	58.0	90.0
TB	583.0	3.298799	6.209522	0.4	0.8	1.00	2.6	75.0
DB	583.0	1.486106	2.808498	0.1	0.2	0.30	1.3	19.7
Alkphos	583.0	290.576329	242.937989	63.0	175.5	208.00	298.0	2110.0
Sgpt	583.0	80.713551	182.620356	10.0	23.0	35.00	60.5	2000.0
Sgot	583.0	109.910806	288.918529	10.0	25.0	42.00	87.0	4929.0
TP	583.0	6.483190	1.085451	2.7	5.8	6.60	7.2	9.6
ALB	583.0	3.141852	0.795519	0.9	2.6	3.10	3.8	5.5
A/G Ratio	579.0	0.947064	0.319592	0.3	0.7	0.93	1.1	2.8

In [7]:
```
print(ILPD.isna().sum())
```

```
Age           0
Gender        0
TB            0
DB            0
Alkphos       0
Sgpt          0
Sgot          0
TP            0
ALB           0
A/G Ratio     4
Selector      0
dtype: int64
```

In [8]:
```
mean_AGR = ILPD["A/G Ratio"].mean()
ILPD["A/G Ratio"] = ILPD["A/G Ratio"].fillna(mean_AGR)
ILPD[ILPD["A/G Ratio"].isna()]
```

Out[8]:

Age	Gender	TB	DB	Alkphos	Sgpt	Sgot	TP	ALB	A/G Ratio	Selector

In [9]:
```
ILPD_1 = pd.get_dummies(ILPD, columns = ["Gender"])
print(ILPD_1.head())
```

```
   Age    TB   DB  Alkphos  Sgpt  Sgot   TP  ALB  A/G Ratio  Selector  \
0   65   0.7  0.1      187    16    18  6.8  3.3       0.90         1
1   62  10.9  5.5      699    64   100  7.5  3.2       0.74         1
2   62   7.3  4.1      490    60    68  7.0  3.3       0.89         1
3   58   1.0  0.4      182    14    20  6.8  3.4       1.00         1
4   72   3.9  2.0      195    27    59  7.3  2.4       0.40         1

   Gender_Female  Gender_Male
0              1            0
1              0            1
2              0            1
3              0            1
4              0            1
```

In [10]:
```
ILPD_1["Selector"] = ILPD_1["Selector"].replace(2,0).values
print(ILPD_1["Selector"])
```

```
0      1
1      1
2      1
3      1
4      1
      ..
578    0
579    1
580    1
581    1
582    0
Name: Selector, Length: 583, dtype: int64
```

In [11]:
```
Data = ILPD_1.drop(columns = "Selector").values
target = ILPD_1["Selector"].values
X = preprocessing.StandardScaler().fit_transform(Data)
Y = target
print(X)
print(Y)
```

```
[[ 1.25209764 -0.41887783 -0.49396398 ... -0.14789798  1.76228085
  -1.76228085]
 [ 1.06663704  1.22517135  1.43042334 ... -0.65069686 -0.56744644
   0.56744644]
 [ 1.06663704  0.6449187   0.93150811 ... -0.17932291 -0.56744644
   0.56744644]
 ...
 [ 0.44843504 -0.4027597  -0.45832717 ...  0.16635131 -0.56744644
   0.56744644]
 [-0.84978917 -0.32216906 -0.35141677 ...  0.16635131 -0.56744644
   0.56744644]
 [-0.41704777 -0.37052344 -0.42269037 ...  1.73759779 -0.56744644
   0.56744644]]
[1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 0 0 1 1 0 0 0 1 0
 1 1 1 1 0 0 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1
 1 0 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 1 0 0 0 0 0 0
 1 0 1 0 0 1 1 1 1 1 1 0 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1
 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1
 0 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 0 1 1 1 1 1 1 1 0 1 1 0 0 1 0 1 1 1
 1 0 1 1 1 1 0 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0
 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1
 1 0 1 0 0 1 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 1 1 0 1 1 1 0 1 0 0 0 0 0 1 1 1
 0 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 0 0 0 0 0 0 0 1 1
 1 0 1 0 0 1 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 0
 1 1 1 1 0 1 0 0 1 1 0 1 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 0 0 1 1 1 1 1 1 1 1
 0 0 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 0
 1 1 0 1 1 1 1 0 0 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1
 1 1 1 1 1 1 0 0 1 1 1 1 0 1 0 1 0 1 1 1 1 0 0 0 0 1 1 0 1 1 1 1 1 0 1 1 1
 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0]
```

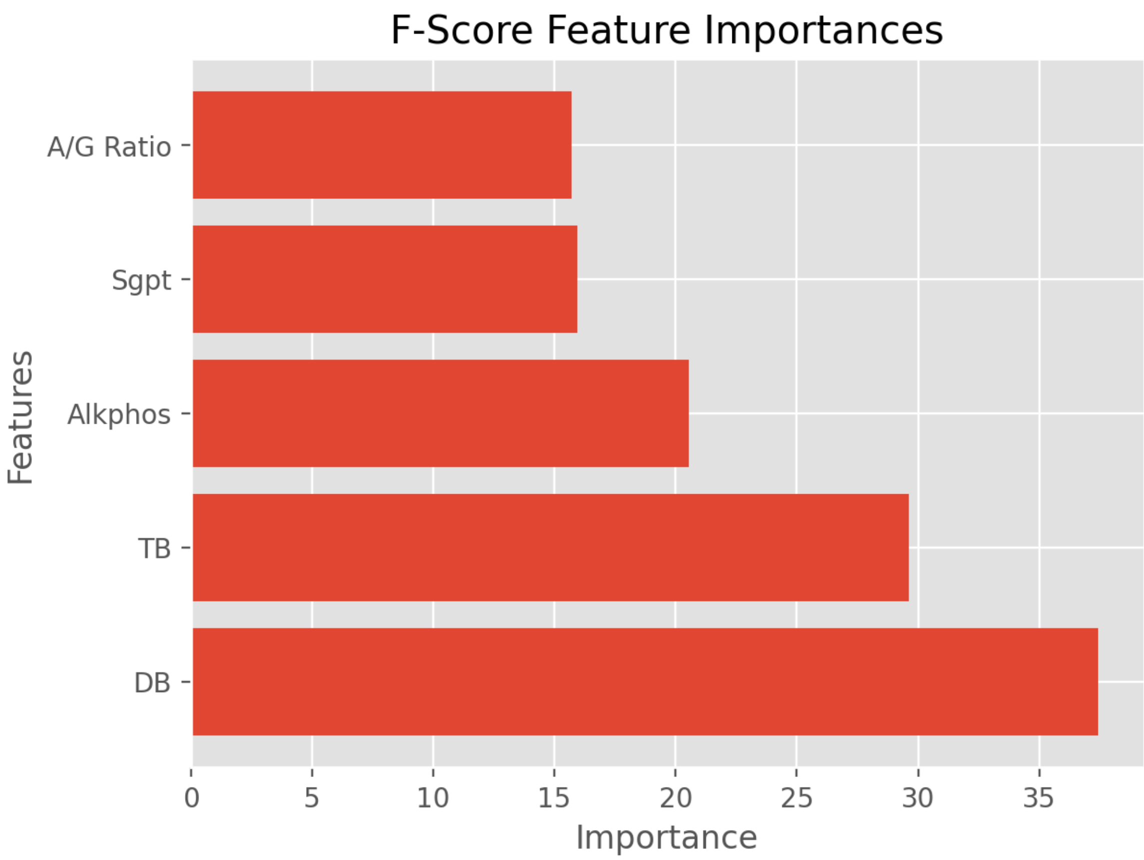
Feature Selection using F-score

In [12]:
```
num_features = 5

fs_fit_fscore = fs.SelectKBest(fs.f_classif, k=num_features)
fs_fit_fscore.fit_transform(X,Y)
fs_indices_fscore = np.argsort(np.nan_to_num(fs_fit_fscore.scores_))[::-1][0:num
print(fs_indices_fscore)


feature_importances_fscore = fs_fit_fscore.scores_[fs_indices_fscore]
print(feature_importances_fscore)

best_features_fscore = ILPD_1.columns[fs_indices_fscore].values
print(best_features_fscore)
```

```
[2 1 3 4 8]
[37.43959214 29.60928154 20.55843531 15.94121994 15.72213621]
['DB' 'TB' 'Alkphos' 'Sgpt' 'A/G Ratio']
```

In order to identify the most influential features from the dataset, we used the SelectKBest method with an F-score criterion. Based on F-scores, the top five features were selected according to their statistical significance in relation to the target variable. It was determined that Direct Bilirubin (DB), Total Bilirubin (TB), Alkaline Phosphatase (Alkphos), Serum Glutamic Pyruvic Transaminase (Sgpt), and Albumin/Globulin Ratio (A/G Ratio) exhibited the highest F-scores of 37.44, 29.61, 20.56, 15.94, and 15.72, respectively. A more focused analysis and supervised modeling can be conducted based on the findings of this study, which provide valuable insights into the key drivers influencing the target variable.

In [13]:
```
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
plt.style.use("ggplot")

def plot_imp(best_features, scores, method_name):
    plt.barh(best_features, scores)
    plt.title(method_name + ' Feature Importances')
    plt.xlabel("Importance")
    plt.ylabel("Features")
    plt.show()
plot_imp(best_features_fscore, feature_importances_fscore, 'F-Score')
```



In [15]:
```
X_selected = ILPD_1[['DB','TB',"Alkphos","Sgpt","A/G Ratio"]]
X_selected = preprocessing.StandardScaler().fit_transform(X_selected)
print(X_selected)
```

```
[[-0.49396398 -0.41887783 -0.42671496 -0.35466541 -0.14789798]
 [ 1.43042334  1.22517135  1.68262856 -0.09159933 -0.65069686]
 [ 0.93150811  0.6449187   0.82158795 -0.11352151 -0.17932291]
 ...
 [-0.45832717 -0.4027597  -0.18776589 -0.17928803  0.16635131]
 [-0.35141677 -0.32216906 -0.43907439 -0.28341834  0.16635131]
 [-0.42269037 -0.37052344 -0.30724042 -0.32726269  1.73759779]]
```

K-Nearest Neighbors (KNN)

In [18]:
```
steps = [("KNN",KNeighborsClassifier())]
pipeline = Pipeline(steps)

# Creating the parameter space
parameters = {"KNN__n_neighbors":[19,20],
             "KNN__p":[2,3],
             "KNN__metric":['chebyshev'],
             "KNN__weights": ['uniform'],
             "KNN__algorithm":["auto"],
             "KNN__leaf_size":[50,70],
             "KNN__n_jobs":[-1,1]}
```

The KNN classifier is configured as a pipeline with a single step, and the parameter space is defined so that the hyperparameters can be tuned. Using this pipeline, that can train and predict models efficiently using the entire machine learning workflow. Tuning occurs with respect to the parameters defined in the parameter space, which include the number of neighbors, the distance metric, the weight function, and the algorithm configuration. KNN classifiers are optimized in terms of predictive accuracy and generalization ability by systematically tuning these hyperparameters.

In [19]:
```
# Define the cross-cross validation strategy
kf = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 123)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_selected,Y, test_size =0.3
```

Model evaluation and performance assessment are conducted using cross-validation and train-test split methodologies. With StratifiedKFold and five splits, the cross-validation strategy ensures that complementary subsets of the data are partitioned while maintaining the class distribution, so that the model can be evaluated robustly. In addition, the train-test split operation divides the dataset into separate training and testing subsets using a 30% test size and 123 random states for reproducibility. This facilitates unbiased estimation of the model's predictive performance on unseen data by training it on one part of the data and evaluating it on the other.

In [20]:
```
# Performing grid search and Cross validation
grid_search = GridSearchCV(pipeline, parameters, cv = kf, scoring = "accuracy")

grid_search.fit(X_train, y_train)

# Get the best scoer and best parameters
print("Best score:",grid_search.best_score_)
print("Best parameters:",grid_search.best_params_)
```

```
Best score: 0.6541403191809696
Best parameters: {'KNN__algorithm': 'auto', 'KNN__leaf_size': 50, 'KNN__metric':
'chebyshev', 'KNN__n_jobs': -1, 'KNN__n_neighbors': 19, 'KNN__p': 2, 'KNN__weight
s': 'uniform'}
```

To identify the optimal configuration for the KNN classifier, a grid search and cross-validation were used in the process of tuning and evaluating the model. To maximize the model's accuracy, GridSearchCV's pipeline, parameter grid, and cross-validation strategy were used to test various combinations of hyperparameters. Models with the following optimal hyperparameters achieved approximately 65.41% accuracy: algorithm='auto', leaf_size=50, metric='chebyshev', n_neighbors=19, p=2, weights='uniform'. According to these results, the optimal configuration of the KNN classifier for the given dataset maximizes its prediction accuracy.

In [21]:
```
# Evaluating the best model on the test set
best_model = grid_search.best_estimator_

test_accuracy = best_model.score(X_test,y_test)
print("Test set accuracy of the best model", test_accuracy)
```

```
Test set accuracy of the best model 0.7771428571428571
```

Following model selection and parameter tweaking with grid search and cross-validation, the best-performing model, a k-nearest neighbors (KNN) classifier, was tested on a test set to assess its ability to predict liver disease in patients. On the test set, the model had an accuracy of roughly 77.71%, indicating that it properly classified about 77.71% of the cases, which is critical for effectively identifying patients with liver illness. This statistic is an important indicator of the model's capacity to generalize effectively to previously encountered patient data, demonstrating its potential effectiveness in real-world circumstances. The enhanced KNN classifier's high test set accuracy shows that it has moderately potential predictive capabilities, providing vital support in identifying liver illness and assisting healthcare practitioners in making educated clinical decisions.

In [22]:
```
y_pred = best_model.predict(X_test)
print(y_pred)

# Assesing the accuracy of the prediction
accuracy = accuracy_score(y_test,y_pred)
print("Accuracy on the test data", accuracy)
```

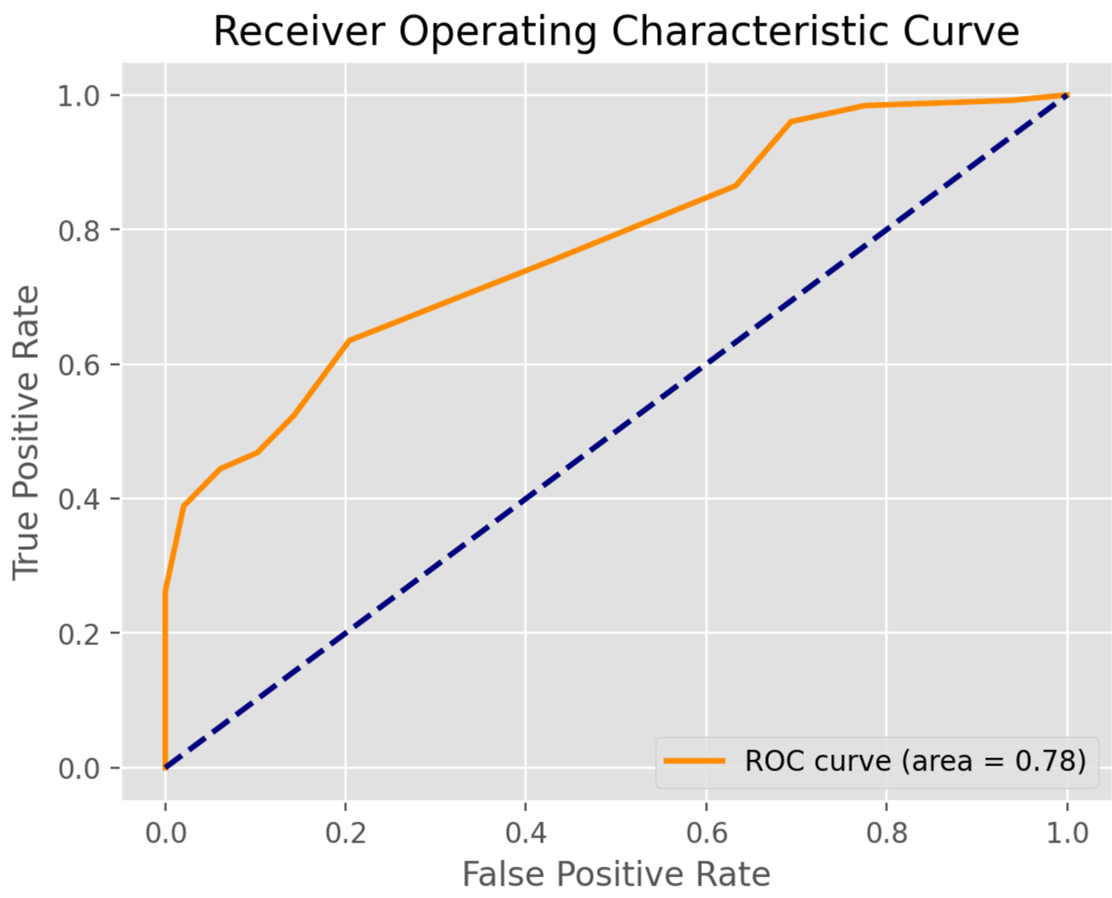
```
[1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1
 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 0 1
 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1]
Accuracy on the test data 0.7771428571428571
```

In [23]:
```
# Displaying the confusion matrix and classification report
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[ 15  34]
 [  5 121]]
              precision    recall  f1-score   support

           0       0.75      0.31      0.43        49
           1       0.78      0.96      0.86       126

    accuracy                           0.78       175
   macro avg       0.77      0.63      0.65       175
weighted avg       0.77      0.78      0.74       175
```

The confusion matrix and classification report provide a detailed assessment of the improved k-nearest neighbors (KNN) classifier's ability to diagnose liver disease. The confusion matrix shows that 15 of the 175 occurrences were accurately identified as not having liver illness (True Negatives), whereas 121 were correctly labeled as having liver disease (True Positives). However, the model predicted 34 occurrences as having liver illness when they did not (False Positives) and missed 5 examples with liver disease (False Negatives). The classification report expands on these findings, showing that while the model has high precision (0.78) and recall (0.96) for patients with liver disease (class 1), its performance for patients without liver disease (class 0) is lower, with precision at 0.75 and recall at 0.31. The model has an overall accuracy of 78%, illustrating that it is effective at correctly classifying individuals with liver disease.

In [24]:
```
# Get predicted probablities for the positive class
y_prob = best_model.predict_proba(X_test)[:,1]

# Compute ROC curve and ROC area
fpr,tpr,thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

#Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color = 'darkorange', lw = 2, label = 'ROC curve (area = %0.2
plt.plot([0,1],[0,1], color = 'navy', lw = 2 , linestyle = '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic Curve")
plt.legend(loc= "lower right")
plt.show()

# Calculate AUC using roc_auc_score
auc_score = roc_auc_score(y_test, y_prob)
print("AUC:", auc_score)
```




```
AUC: 0.7766439909297053
```

To evaluate the predictive model's performance, a Receiver Operating Characteristic (ROC) curve analysis was performed on the liver dataset. The model's positive class probabilities were used to calculate the true positive rate (TPR) and false positive rate (FPR) at various threshold values, which were then plotted to create the ROC curve. The graph shows the model's ability to distinguish between the two classes at various thresholds. The plot and the roc_auc_score function yield an AUC of roughly 0.76. This represents a moderate level of discrimination, implying that the model has a 76.1% chance of properly discriminating a positive and negative class occurrence in the liver dataset.

Decision Tree Classifier

In [25]:

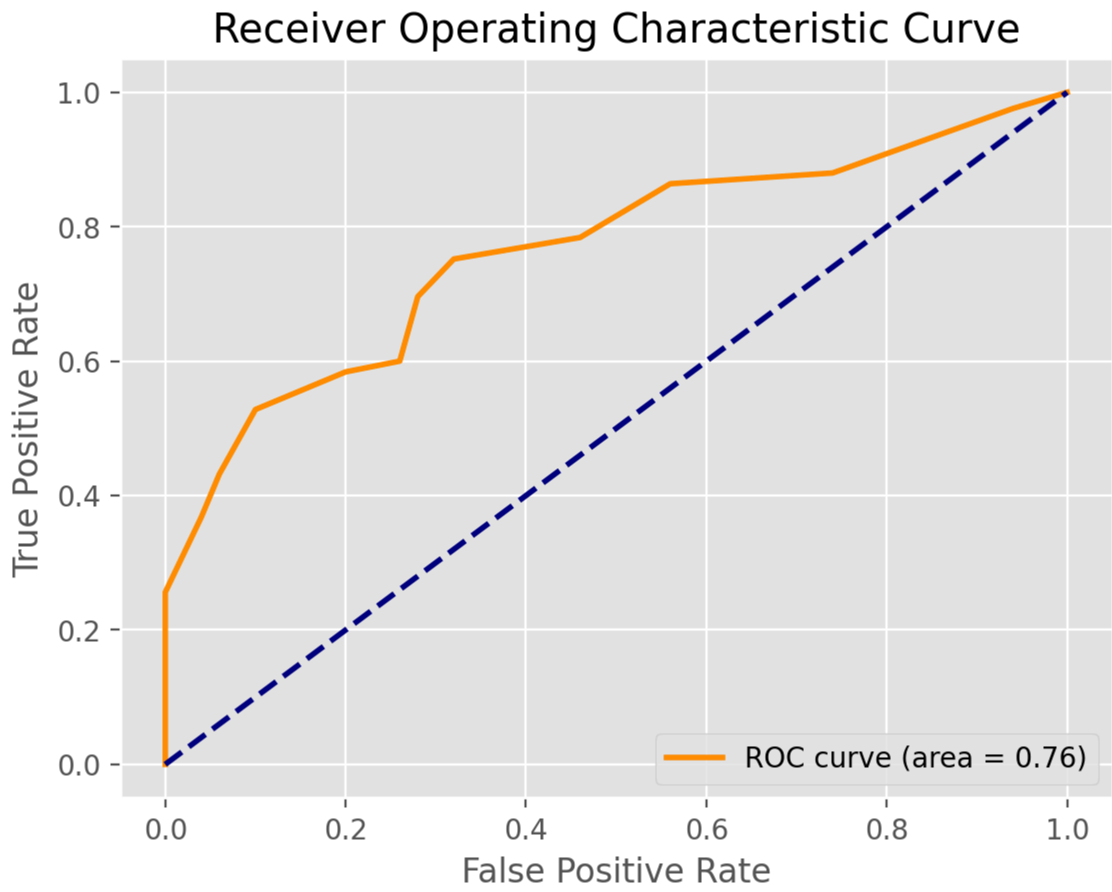
```
# Splitting the Data into Training and Test sets
D_1_train, D_1_test, z_1_train, z_1_test = train_test_split(X_selected, Y, test_

# Initializing the Decision Tree Classifier
dt = DecisionTreeClassifier(criterion = 'gini',random_state =123,min_samples_spl

# Defining the Cross Validation Method

cv_method_dt = StratifiedKFold(n_splits = 5,
                                    shuffle = True,
                                    random_state = 123)

# Performing Cross-Validation Method
cv_scores = cross_val_score(dt, D_1_train, z_1_train, cv = cv_method_dt)
```

```python
# Fitting the Model to the Training Data
dt.fit(D_1_train, z_1_train)


# Making Predictions on the Test Data
preds = dt.predict(D_1_test)

# Calculating the Training Accuracy
train_score = dt.score(D_1_train, z_1_train)

# Calculating the Test Accuracy
test_score = dt.score(D_1_test, z_1_test)

# Calculating the Accuracy Score
accuracy_entropy = accuracy_score(z_1_test, preds)

# Printing the Training Accuracy
print(f"Training Accuracy:", train_score)

# Printing the Test Accuracy
print(f"Test accuracy achieved by using entropy:{test_score:.3f}")

# Printing the confusion matrix
print(confusion_matrix(z_1_test, preds))


# Printing the classification report
print(classification_report(z_1_test, preds))
```

```
Training Accuracy: 0.7794117647058824
Test accuracy achieved by using entropy:0.743
[[ 22  28]
 [ 17 108]]
              precision    recall  f1-score   support

           0       0.56      0.44      0.49        50
           1       0.79      0.86      0.83       125

    accuracy                           0.74       175
   macro avg       0.68      0.65      0.66       175
weighted avg       0.73      0.74      0.73       175
```

The findings show that the Decision Tree classifier trained on the dataset has a training accuracy of roughly 77.94%, indicating that the model matches the training data pretty well. However, the test accuracy falls to 74.3%, implying little overfitting, in which the model performs better on training data than on unknown data. According to the confusion matrix, the model accurately predicts 22 out of 50 class 0 cases and 108 out of 125 class 1 instances. The precision, recall, and F1-score metrics provide more insight into the model's performance. Class 0 has a precision of 0.56, recall of 0.44, and F1-score of 0.49, suggesting lower accuracy than class 1, which has a precision of 0.79, recall of 0.86, and F1-score of 0.83. This disparity shows that the model performs significantly better with the majority class (class 1) than with the minority class (class 0).

In [23]:
```
# Get predicted probabilities for the positive class
D_prob = dt.predict_proba(D_1_test)[:, 1]

# Compute ROC curve and ROC area
fpr_dt, tpr_dt, thresholds = roc_curve(z_1_test, D_prob)
roc_auc = auc(fpr_dt, tpr_dt)
auc_score_decision_tree = roc_auc_score(z_1_test, D_prob)

#Plot ROC curve
plt.figure()
plt.plot(fpr_dt, tpr_dt, color = 'darkorange', lw = 2, label = 'ROC curve (area
plt.plot([0,1],[0,1], color = 'navy', lw = 2 , linestyle = '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic Curve")
plt.legend(loc= "lower right")
plt.show()

# Calculate AUC using roc_auc_score
print("AUC:", auc_score_decision_tree)
```




```
AUC: 0.76448
```

The Decision Tree classifier's performance is evaluated using the Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC) statistic. The AUC, a single scalar value characterizing the model's overall ability to distinguish between positive and negative classes, is determined using both the auc function and roc_auc_score, yielding an AUC of around 0.7645. The AUC score of 0.7645 indicates that the classifier is good, but not perfect, at distinguishing between positive and negative classes.

Logistic Regression

In [26]:
```
# Spliting the data
L_train, L_test, m_train, m_test = train_test_split(X, Y, stratify = Y, test_siz

# Define the Logistic Regression model
lr = LogisticRegression(random_state = 123)

# Define the parameter grid for the grid search
param_grid_lr= {"C":[0.001,0.01,0.1,1,10,100],
            "penalty":["l1","l2"]}

# Initilize stratified k-fold cross-validation
cv_method_lr = StratifiedKFold(n_splits = 5,
                               shuffle = True,
                               random_state = 123)

# Initialize GridSearchCV
grid_search_lr = GridSearchCV(estimator =lr, param_grid =param_grid_lr, cv= # In
cv_method_lr)


# Fit the Logistic Regression model
grid_search_lr.fit(L_train, m_train)

best_lr = grid_search_lr.best_estimator_

# Prediction on the test set
m_pred_lr = best_lr.predict(L_test)

# Calculating Training accuracy
training_accuracy_lr = best_lr.score(L_train, m_train)
print("Training Accuracy Logistic Regression:",training_accuracy_lr)


# Calculate accuracy
accuracy_lr = accuracy_score(m_test, m_pred_lr)
print("Test Accuracy Logistic Regression:", accuracy_lr)

# Generate Confusion matrix
print("Confusion Matrix Logistic Regression:")
confusion_lr = confusion_matrix(m_test, m_pred_lr)
print(confusion_lr)


# Generate Classification matrix
print("Classification Report Logistic Regression:")
print(classification_report(m_test, m_pred_lr))
```

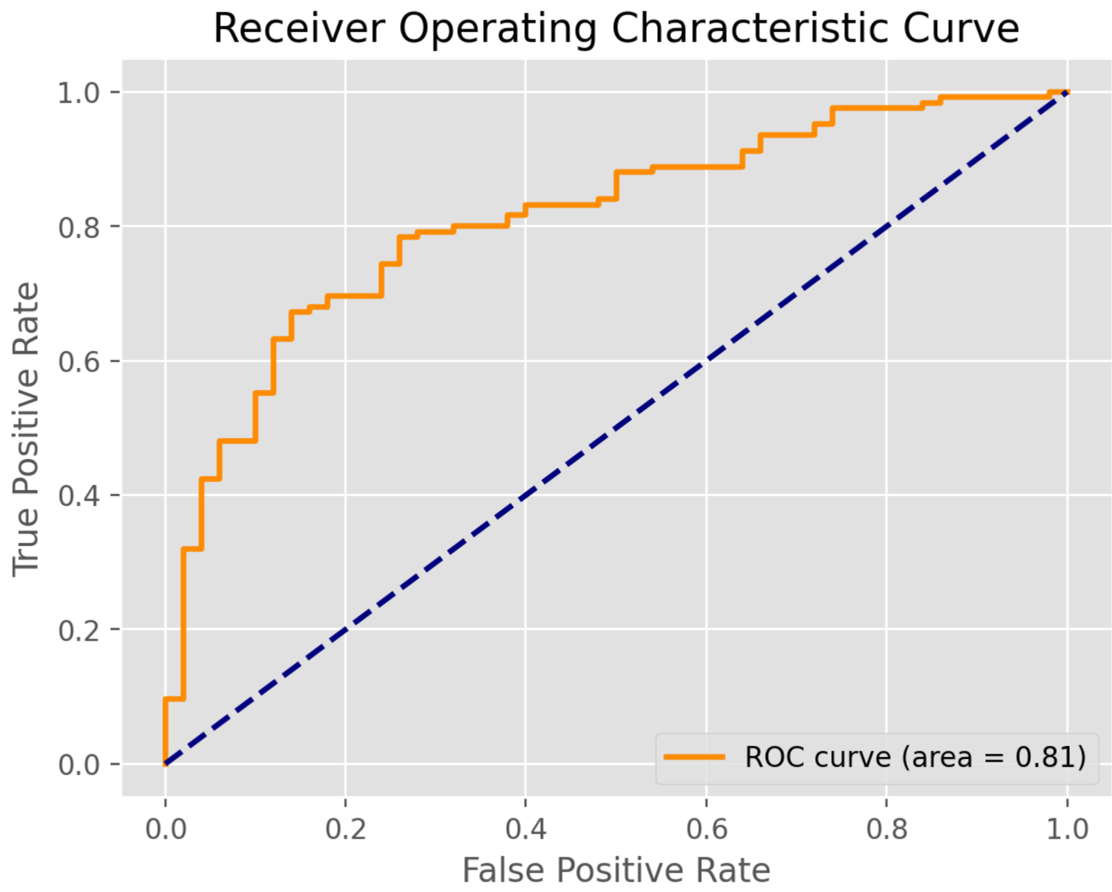
```
Training Accuracy Logistic Regression: 0.7083333333333334
Test Accuracy Logistic Regression: 0.7657142857142857
Confusion Matrix Logistic Regression:
[[ 17  33]
 [  8 117]]
Classification Report Logistic Regression:
              precision    recall  f1-score   support

           0       0.68      0.34      0.45        50
           1       0.78      0.94      0.85       125

    accuracy                           0.77       175
   macro avg       0.73      0.64      0.65       175
weighted avg       0.75      0.77      0.74       175
```

The model's training accuracy is at 70.83%, indicating that it fits the training data reasonably well. The test accuracy is higher (76.57%), indicating that the model generalizes reasonably well to previously unseen data. The confusion matrix shows that the model accurately predicts 17 out of 50 cases for class 0 and 117 out of 125 instances for class 1, indicating great performance for the majority class (class 1) but poor performance for the minority class (class 0). According to the classification report, class 0 has a precision of 0.68, recall of 0.34, and F1-score of 0.45, showing that the model struggles with false positives and false negatives. Class 1 has a higher precision of 0.78, recall of 0.94, and F1-score of 0.85, indicating that the model is effective in detecting the majority class. The macro average scores, which account for performance in all classes equally, show an overall balanced performance, however the weighted average scores, which give greater weight to the majority class, show slightly better performance. These findings imply that, while the Logistic Regression model performs well overall, but need to improve its capacity to correctly categorize minority classes.

In [27]:
```
# Compute the predicted probabilities for the postive class
m_pred_prob_lr = best_lr.predict_proba(L_test)[:,1]

# Calculating the AUC score
auc_lr = roc_auc_score(m_test, m_pred_prob_lr)
print("AUC Score Logistic Regression:", auc_lr)


# Compute ROC curve and ROC area
fpr_lr, tpr_lr, thresholds_lr = roc_curve(m_test, m_pred_prob_lr)
roc_auc = auc(fpr_lr, tpr_lr)
auc_score_logistic_regression = roc_auc_score(m_test, m_pred_prob_lr)

#Plot ROC curve
plt.figure()
plt.plot(fpr_lr, tpr_lr, color = 'darkorange', lw = 2, label = 'ROC curve (area
plt.plot([0,1],[0,1], color = 'navy', lw = 2 , linestyle = '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic Curve")
plt.legend(loc= "lower right")
plt.show()
```

```
# Calculate AUC using roc_auc_score
print("AUC:", auc_score_logistic_regression)
```

```
AUC Score Logistic Regression: 0.8121600000000001
```




```
AUC: 0.8121600000000001
```

The Logistic Regression model's AUC score is roughly 0.8121 which is 81.22%, indicating a high level of distinction between classes; a value closer to one indicates superior model performance. The ROC curve is then shown, with the AUC value supplied in the explanation to provide clarification. The curve is much higher than the diagonal line, demonstrating that the model outperforms random chance. This visual representation verifies the model's ability to differentiate between classes well. Overall, the high AUC score indicates that the Logistic Regression model has great predictive potential and distinguishes well between the dataset's two classes.

Random Forest

In [28]:

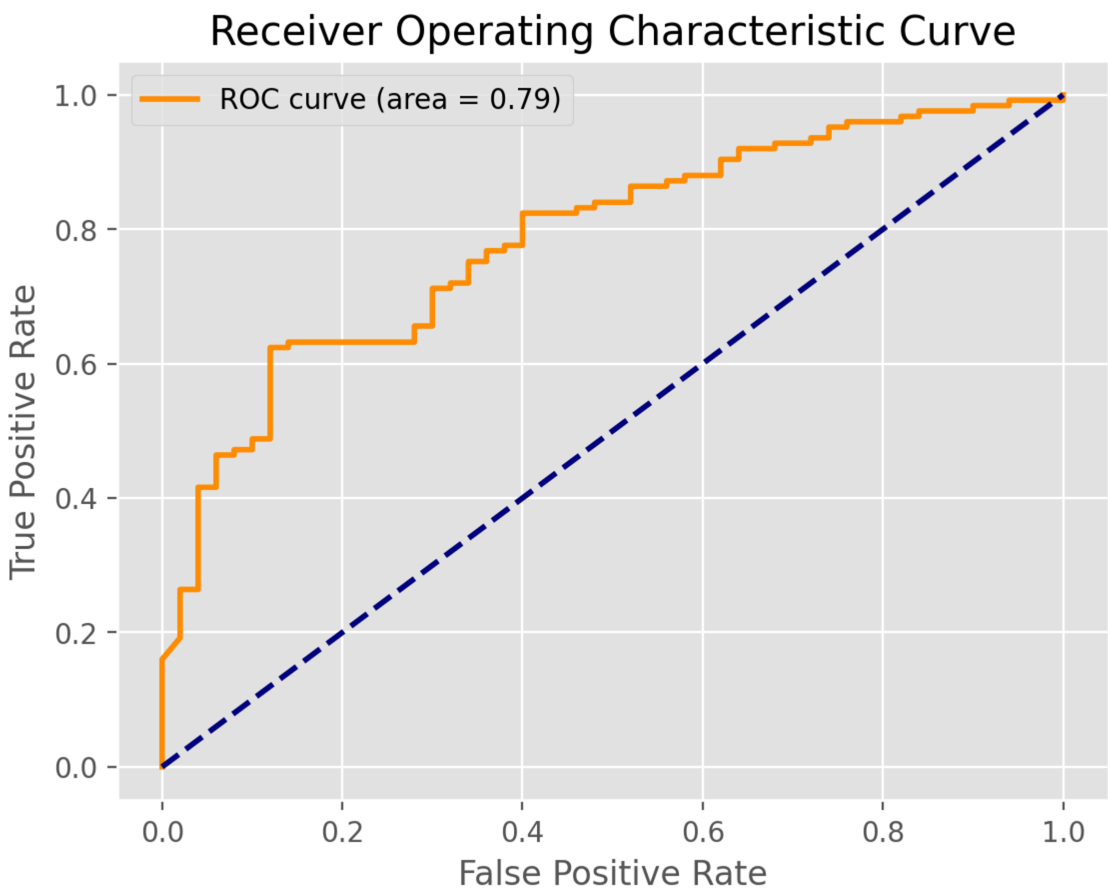
```
R_train, R_test, s_train, s_test = train_test_split(X_selected, Y, stratify = Y,

# Defining the Random Forest Classifier

# Define the Random Forest classifier
rf_classifier = RandomForestClassifier(random_state=42)

# Define the parameter grid to search

param_rf = {'n_estimators':[10,20],
            'max_depth':[None],
            'min_samples_split':[10,20],
```

```
              'min_samples_leaf':[10,20]}

stratified_kfold_rf = StratifiedKFold(n_splits= 5)

# Perform Grid Search
grid_search_rf = GridSearchCV(estimator = rf_classifier, param_grid = param_rf,
grid_search_rf.fit(R_train, s_train)


# Extract cross-validation results
cv_results = grid_search_rf.cv_results_

# Print the mean cross-validation score
mean_cv_score = cv_results["mean_test_score"].mean()
print("Mean Cross Validation Score", mean_cv_score)


# Print the best parameters and best score
print("Best Parameters:", grid_search_rf.best_params_)
print("Best Score:",grid_search_rf.best_score_)

# Make predictions on the testing data with the best model
best_rf_model = grid_search_rf.best_estimator_
s_pred = best_rf_model.predict(R_test)

# Calculate the training accuracy using the trained model
training_accuracy = best_rf_model.score(R_train, s_train)
print('Training Accuracy:',training_accuracy )

# Evaluate the model
test_accuracy = accuracy_score(s_test, s_pred)
print("test_accuracy:", test_accuracy)

# print confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(s_test, s_pred))

# Print classification report
print("Calssification Report")
print(classification_report(s_test, s_pred))
```

```
Mean Cross Validation Score 0.6876392652815416
Best Parameters: {'max_depth': None, 'min_samples_leaf': 10, 'min_samples_split': 
10, 'n_estimators': 20}
Best Score: 0.6888888888888889
Training Accuracy: 0.7867647058823529
test_accuracy: 0.76
Confusion Matrix:
[[ 18  32]
 [ 10 115]]
Calssification Report
              precision    recall  f1-score   support

           0       0.64      0.36      0.46        50
           1       0.78      0.92      0.85       125

    accuracy                           0.76       175
   macro avg       0.71      0.64      0.65       175
weighted avg       0.74      0.76      0.74       175
```

To maintain class distribution, the data is divided into training and testing sets by stratified sampling. Grid search is performed over a predetermined parameter grid, with accuracy as the primary consideration, and the best model is chosen based on the highest mean cross-validation score. The best model is then evaluated on the test set to see how well it generalizes. The mean cross-validation score, which estimates the model's performance on unseen data, is roughly 0.686. The best model got a training accuracy of 78.68% and a test accuracy of 76%, indicating good generalization to new data. However, the model's performance differs among classes, with class 1 having higher precision, recall, and F1-score than class 0. According to the confusion matrix, the model correctly predicted 18 out of 50 class 0 cases and 115 out of 125 class 1 instances. Overall, the model performs quite well, although there appears to be potential for improvement, notably in finding instances of class 0.

In [29]:
```
s_prob = best_rf_model.predict_proba(R_test)[:,1]

# Compute ROC curve and ROC area
fpr_rf, tpr_rf, thresholds = roc_curve(s_test,s_prob)
roc_auc_value = roc_auc_score(s_test, s_prob)

# Plot ROC curve
plt.figure()
plt.plot(fpr_rf,tpr_rf, color = "darkorange", lw = 2, label = 'ROC curve (area =
plt.plot([0,1],[0,1], color = 'navy', lw = 2 , linestyle = '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic Curve")
plt.legend()
plt.show()

# Print AUC score
print("AUC:", roc_auc_value)
```




```
AUC: 0.78528
```

The AUC score is roughly 0.785, showing that the Random Forest classifier can discriminate between the two classes. A greater AUC value closer to one indicates better discrimination, whereas an AUC of 0.5 denotes no better performance than random chance. As a result, the AUC value of 0.785 indicates that the Random Forest classifier is moderately good at differentiating between positive and negative classes.

Voting Classifier

In [28]:

```
F_train, F_test, g_train, g_test = train_test_split(X, Y, test_size = 0.3, strat

# Instantiate lr
lr = LogisticRegression(random_state = 123)

# Instantiate dt

dt = DecisionTreeClassifier(random_state = 123, criterion = 'gini', min_samples_

# Define the list classifiers
models = [('Logistic Regression',lr), ('Classification Tree', dt)]
stratified_kfold_vc = StratifiedKFold(n_splits= 5)


for model_name, model in models:
    cv_scores = cross_val_score(model, X, Y, cv = stratified_kfold_vc)
    print(f'{model_name} Cross-Validation Mean Accuracy:{cv_scores.mean()}')

    model.fit(F_train, g_train)
```

```
    g_pred_1 = model.predict(F_test)

    accuracy = accuracy_score(g_test, g_pred_1)

    print('{:s}:{:.3f}'.format(model_name,accuracy))

# Deriving Voting Classifier
voting_class = VotingClassifier(estimators = models, voting = 'soft')

# Fitting the training set
voting_class.fit(F_train, g_train)


g_pred_vc = voting_class.predict(F_test)

train_accuracy = voting_class.score(F_train,g_train)
print("Training Accuracy Voting Classifier:",train_accuracy)

# Accuracy score
test_accuracy = accuracy_score(g_test,g_pred_vc)
print('Test accuracy score:{:.3f}'.format(test_accuracy))

# Confusion Matrix
print(confusion_matrix(g_test, g_pred_vc))

# Classification report
print(classification_report(g_test, g_pred_vc))
```

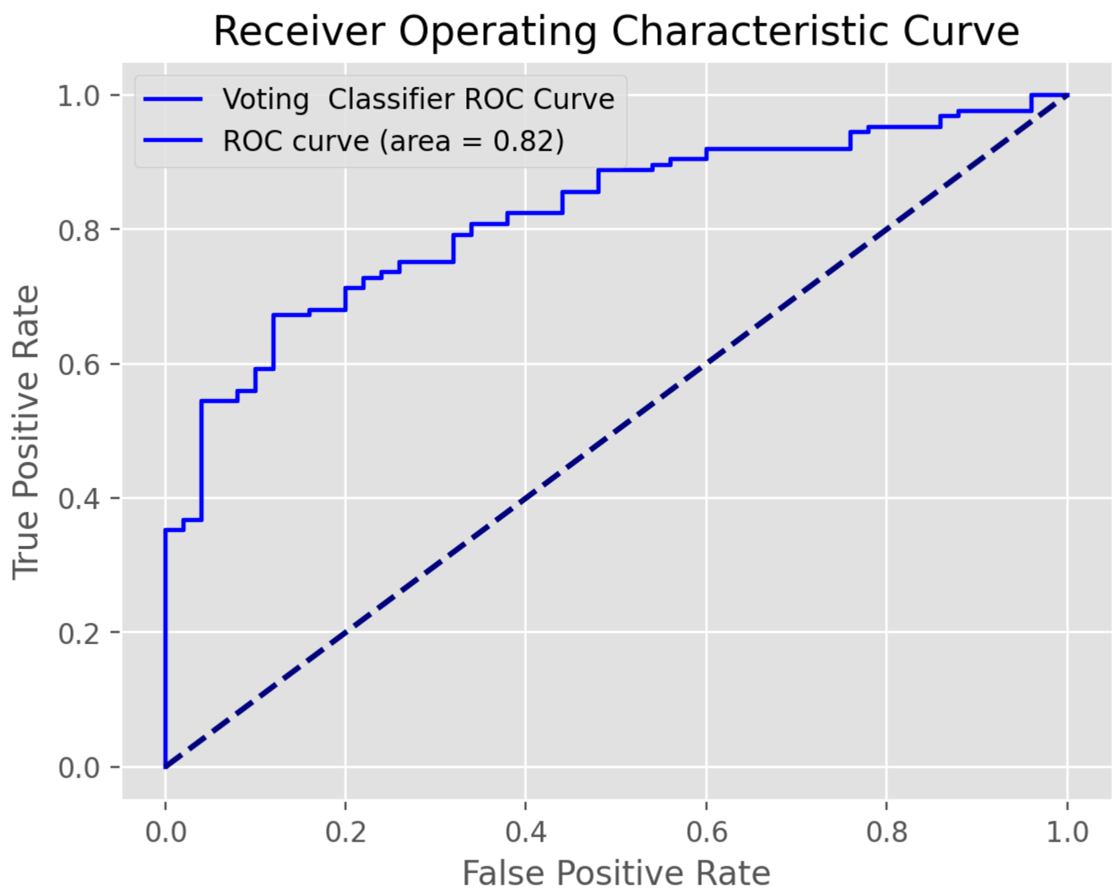
```
Logistic Regression Cross-Validation Mean Accuracy:0.7170055997642205
Logistic Regression:0.766
Classification Tree Cross-Validation Mean Accuracy:0.6688623636899498
Classification Tree:0.760
Training Accuracy Voting Classifier: 0.8137254901960784
Test accuracy score:0.771
[[ 26  24]
 [ 16 109]]
              precision    recall  f1-score   support

           0       0.62      0.52      0.57        50
           1       0.82      0.87      0.84       125

    accuracy                           0.77       175
   macro avg       0.72      0.70      0.71       175
weighted avg       0.76      0.77      0.77       175
```

The following code combines Logistic Regression and Decision Tree classifiers, using both individual models and a Voting Classifier. The dataset is initially divided into two sets: training and testing. The Logistic Regression and Decision Tree classifiers are then instantiated. The models are evaluated separately using cross-validation, and their accuracies are calculated on the test set. Both models are then used to generate a Voting Classifier. The Voting Classifier improves overall performance by combining the predictions from multiple weighted or unweighted models. The confusion matrix and classification report provide information on the model's performance in classifying occurrences into different classes.

The model accurately predicted 26 cases in class 0 (the first row) but mistakenly forecasted 24 instances in class 1 (false positives). The model accurately predicted 109 instances (true positives) and missed 16 instances (false negatives) in class 1 (the second row). Precision is the fraction of genuine positive predictions out of all positive predictions generated by the model, demonstrating how accurate the model is at predicting a specific class.

Recall, also known as sensitivity, is the proportion of real positive cases properly detected by the model out of all actual positive instances, assessing the model's ability to capture all positive instances. The F1-score is the harmonic mean of precision and recall, which provides a balanced evaluation of model performance. In this situation, class 0 has poorer precision, recall, and F1-score than class 1, showing that the model is less effective at detecting occurrences of class 0. The model's overall accuracy is 0.77, indicating the proportion of properly identified occurrences out of the total number of instances in the test set.

In [29]:
```
# Predicting probabilities for the positive class from the voting classifier

voting_class_prob = voting_class.predict_proba(F_test)[:,1]

fpr_vc , tpr_vc, thresholds_vc = roc_curve(g_test, voting_class_prob)

plt.plot(fpr_vc , tpr_vc, color = 'blue', label = 'Voting  Classifier ROC Curve'


# Calculate the false positive rate (FPR) and true positive rate (TPR) for the v
fpr_vc, tpr_vc, thresholds_vc = roc_curve(g_test, voting_class_prob)
auc_vc = roc_auc_score(g_test, voting_class_prob)

# Plot ROC curve for the voting classifier
plt.plot(fpr_vc, tpr_vc, color='blue', label='ROC curve (area = %0.2f)' % auc_vc
plt.plot([0,1],[0,1], color = 'navy', lw = 2 , linestyle = '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic Curve")
plt.legend()
plt.show()

# Calculate the area under the ROC curve (AUC) for the voting classifier
print("AUC for Voting Classifier:", auc_vc)
```




```
AUC for Voting Classifier: 0.8208
```

The ROC curve graphically depicts the trade-off between true positive rate (sensitivity) and false positive rate (1-specificity) across different levels. The AUC score measures the classifier's overall performance in differentiating between positive and negative classifications. In this scenario, the Voting Classifier's AUC score is roughly 0.8208, showing a good level of class separability. A greater AUC value closer to one indicates better model compared to the previous models.

XGBoost

In [30]:
```
print(ILPD_1)
print(ILPD[ILPD["Selector"]==1])
```

```
     Age    TB   DB  Alkphos  Sgpt  Sgot   TP  ALB  A/G Ratio  Selector  \
0     65   0.7  0.1      187    16    18  6.8  3.3       0.90         1
1     62  10.9  5.5      699    64   100  7.5  3.2       0.74         1
2     62   7.3  4.1      490    60    68  7.0  3.3       0.89         1
3     58   1.0  0.4      182    14    20  6.8  3.4       1.00         1
4     72   3.9  2.0      195    27    59  7.3  2.4       0.40         1
..   ...   ...  ...      ...   ...   ...  ...  ...        ...       ...
578   60   0.5  0.1      500    20    34  5.9  1.6       0.37         0
579   40   0.6  0.1       98    35    31  6.0  3.2       1.10         1
580   52   0.8  0.2      245    48    49  6.4  3.2       1.00         1
581   31   1.3  0.5      184    29    32  6.8  3.4       1.00         1
582   38   1.0  0.3      216    21    24  7.3  4.4       1.50         0

     Gender_Female  Gender_Male
0                1            0
1                0            1
2                0            1
3                0            1
4                0            1
..             ...          ...
578              0            1
579              0            1
580              0            1
581              0            1
582              0            1

[583 rows x 12 columns]
     Age  Gender    TB   DB  Alkphos  Sgpt  Sgot   TP  ALB  A/G Ratio  \
0     65  Female   0.7  0.1      187    16    18  6.8  3.3       0.90
1     62    Male  10.9  5.5      699    64   100  7.5  3.2       0.74
2     62    Male   7.3  4.1      490    60    68  7.0  3.3       0.89
3     58    Male   1.0  0.4      182    14    20  6.8  3.4       1.00
4     72    Male   3.9  2.0      195    27    59  7.3  2.4       0.40
..   ...     ...   ...  ...      ...   ...   ...  ...  ...        ...
576   32    Male  15.0  8.2      289    58    80  5.3  2.2       0.70
577   32    Male  12.7  8.4      190    28    47  5.4  2.6       0.90
579   40    Male   0.6  0.1       98    35    31  6.0  3.2       1.10
580   52    Male   0.8  0.2      245    48    49  6.4  3.2       1.00
581   31    Male   1.3  0.5      184    29    32  6.8  3.4       1.00

     Selector
0           1
1           1
2           1
3           1
4           1
..        ...
576         1
577         1
579         1
580         1
581         1

[416 rows x 11 columns]
```

In [31]:
```
# Splitting the dataset into training and testing sets
Xg_train, Xg_test, yg_train, yg_test = train_test_split(X,Y, test_size = 0.3, ra


# Defining the parameter grid for XGBoost hyperparameter tuning
```

```
xgb_param_grid = { 'max_depth':[5],
                   'min_child_weight':[20],
                   'gamma':[0.4],
                   "subsample":[1.0],
                   "colsample_bytree":[0.8],
                   'n_estimator':[200],
                   'reg_alpha':[4.5],
                   'reg_lambda':[7]

}

# Initializing the XGBoost classifier

xg_classifier = xgb.XGBClassifier(object = "binary:logistic",n_estimators = 200,

# Initializing Stratified K-Fold cross-validator
skf = StratifiedKFold(n_splits =10, shuffle = True, random_state =999)

# Performing grid search for hyperparameter tuning
xgb_grid_search = GridSearchCV(estimator = xg_classifier, param_grid = xgb_param

# Fitting the grid search to the training data
xgb_grid_search.fit(Xg_train, yg_train)

# Getting the best parameters from the grid search
best_params = xgb_grid_search.best_params_
print("Best parameters XG Boost", best_params)

# Training the model with the best parameters
xg_classifier = xgb.XGBClassifier(** best_params)
```

```
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Best parameters XG Boost {'colsample_bytree': 0.8, 'gamma': 0.4, 'max_depth': 5,
'min_child_weight': 20, 'n_estimator': 200, 'reg_alpha': 4.5, 'reg_lambda': 7, 's
ubsample': 1.0}
```

The dataset is split into training and testing sets using a 70-30 split ratio. Following that, a parameter grid is created for hyperparameter customization of an XGBoost classifier. The grid includes the parameters'max_depth','min_child_weight', 'gamma','subsample', 'colsample_bytree', 'n_estimator','reg_alpha', and'reg_lambda'. The XGBoost classifier is initialized with default values, such as 'object' set to "binary:logistic", 'n_estimators' set to 200, and'seed' set to 999. For a more rigorous evaluation of the model's performance throughout the grid search phase, stratified K-Fold cross-validation with 10 folds is used. Grid search is used over the parameter grid to discover the hyperparameter combination that optimizes the accuracy score. The optimal parameters are found by fitting the grid search to the training data. These best parameters are then used to train a new XGBoost classifier model, yielding a model that has been optimized for the current dataset using the hyperparameters of choice.

In [32]:
```
scores = cross_val_score(xg_classifier, Xg_train, yg_train, cv = 10, scoring = '
print(scores.mean())
```

```
0.7158536585365853
```

Following cross-validation, the mean accuracy score across all folds is calculated and printed. In this scenario, the mean accuracy score is roughly 0.716, suggesting that the model correctly identifies approximately 71.6% of the training data. This score is used to estimate the model's performance and evaluate its generalization capabilities.

In [33]:
```
xg_classifier_new = xgb.XGBClassifier(** best_params)
xg_classifier_new.fit(Xg_train, yg_train)

# Predicting on the training set
train_predict = xg_classifier_new.predict(Xg_train)
train_accuracy = accuracy_score(yg_train, train_predict)
print("Train Accuracy:%f" % train_accuracy)

# predicting on the test set
test_predict = xg_classifier_new.predict(Xg_test)
test_accuracy = accuracy_score(yg_test,test_predict)
print("Test Accuracy:%f" % test_accuracy)
```

```
Train Accuracy:0.737745
Test Accuracy:0.714286
```

Following the training phase, the trained model is used to predict outcomes on both the training and test sets. The accuracy_score function is used to calculate the proportion of accurately predicted instances in both the training and test sets by comparing the predicted labels to the actual identifiers. The training accuracy of around 0.738 means that the model properly identifies roughly 73.8% of the items in the training set. Similarly, the test accuracy of roughly 0.714 indicates that the model works comparably on unknown data, accurately identifying approximately 71.4% of the samples in the test set.

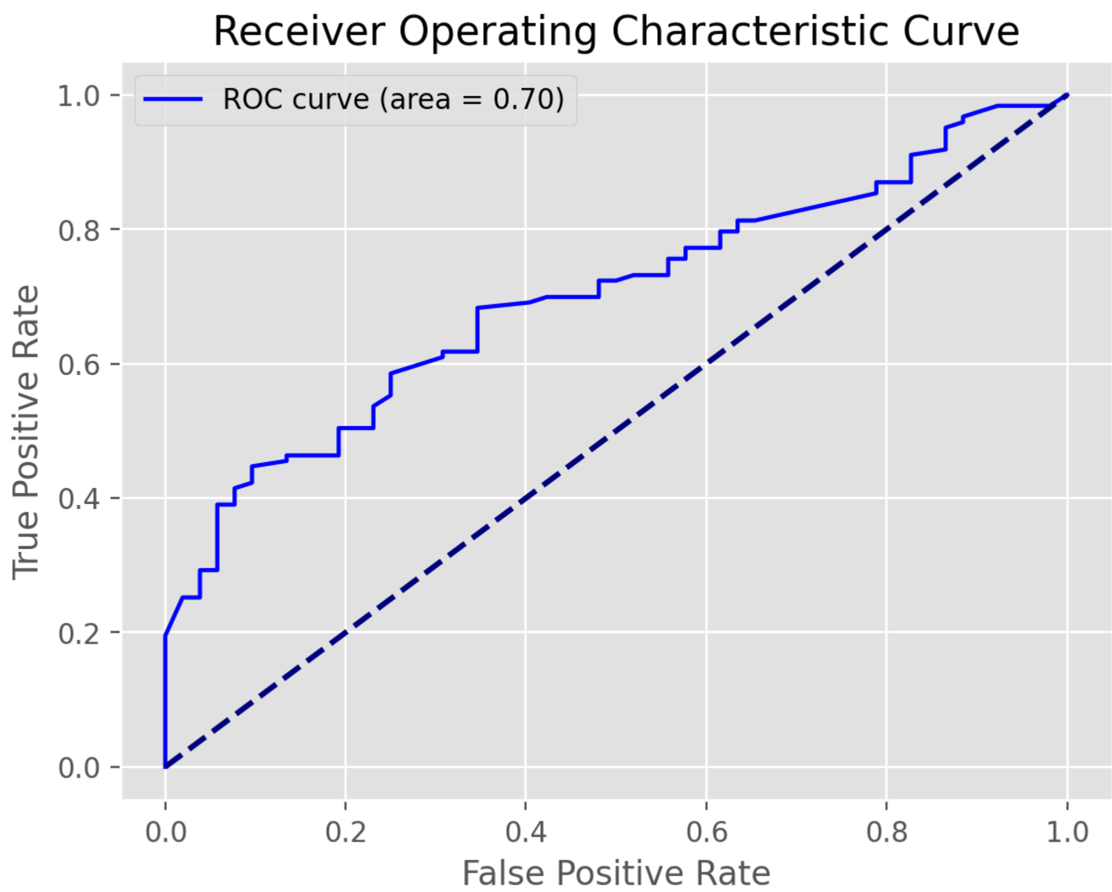
In [34]:
```
xg_classifier_new.fit(Xg_train, yg_train)
test_predict = xg_classifier_new.predict(Xg_test)

# Compute confusion matrix
conf_matrix = confusion_matrix(yg_test,test_predict)
print("Confusion Matrix:")
print(conf_matrix)

# Generate and print classification report
class_report = classification_report(yg_test, test_predict)
print("Classification Report")
print(class_report)
```

```
Confusion Matrix:
[[  6  46]
 [  4 119]]
Classification Report
              precision    recall  f1-score   support

           0       0.60      0.12      0.19        52
           1       0.72      0.97      0.83       123

    accuracy                           0.71       175
   macro avg       0.66      0.54      0.51       175
weighted avg       0.69      0.71      0.64       175
```

The confusion matrix reveals the classifier's performance by displaying the number of true positive, true negative, false positive, and false negative predictions. A classification report is also created, with metrics such as precision, recall, and F1-score for each class (0 and 1). Precision is the proportion of correctly predicted instances among all instances predicted as belonging to a specific class, whereas recall is the fraction of accurately anticipated occurrences among all actual occurrences of the class. The F1-score is the harmonic mean of precision and recall, providing a balanced evaluation of the classifier's performance.

In this scenario, the classification report shows that the classifier has a greater precision and recall for class 1 than for class 0, implying that it performs better at correctly recognizing instances of class 1. However, the classifier's performance for class 0 is quite poor, as evidenced by low precision and recall values, resulting in a lower F1-score for class 0.

In [35]:
```
# Predict probabilities for the positive class from the XGBoost classifier
xg_classifier_new.fit(Xg_train, yg_train)
xg_probabilities = xg_classifier_new.predict_proba(Xg_test)[:,1]

# Compute the ROC curve and AUC score
fpr, tpr, thresholds = roc_curve(yg_test, xg_probabilities)
auc_score_xg = roc_auc_score(yg_test, xg_probabilities)


plt.plot(fpr, tpr, color='blue', label='ROC curve (area = %0.2f)' %auc_score_xg)
plt.plot([0,1],[0,1], color = 'navy', lw = 2 , linestyle = '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic Curve")
plt.legend()
plt.show()

# Calculate the area under the ROC curve (AUC) for the voting classifier
print("AUC for XGBooster Classifier:", auc_score_xg)
```




```
AUC for XGBooster Classifier: 0.6996560350218886
```

The AUC score evaluates the classifier's overall performance by indicating the likelihood that a randomly chosen positive instance would rank higher than a randomly picked negative case. In this situation, the XGBoost classifier's AUC score is around 0.70, indicating that it does relatively well at differentiating between positive and negative instances.

LIGHT GBM

In [36]:
```
ILPD["Selector"] = ILPD["Selector"].replace(2,0).values

Y_LGBM = ILPD["Selector"]

X_LGBM = ILPD.drop("Selector", axis = 1)

X_LGBM["Gender"] = X_LGBM["Gender"].astype("category")


X_LGBM = X_LGBM[['DB','TB',"Alkphos","Sgpt","A/G Ratio"]]
print(X_LGBM.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 583 entries, 0 to 582
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   DB         583 non-null    float64
 1   TB         583 non-null    float64
 2   Alkphos    583 non-null    int64  
 3   Sgpt       583 non-null    int64  
 4   A/G Ratio  583 non-null    float64
dtypes: float64(3), int64(2)
memory usage: 22.9 KB
None
```

In [37]:

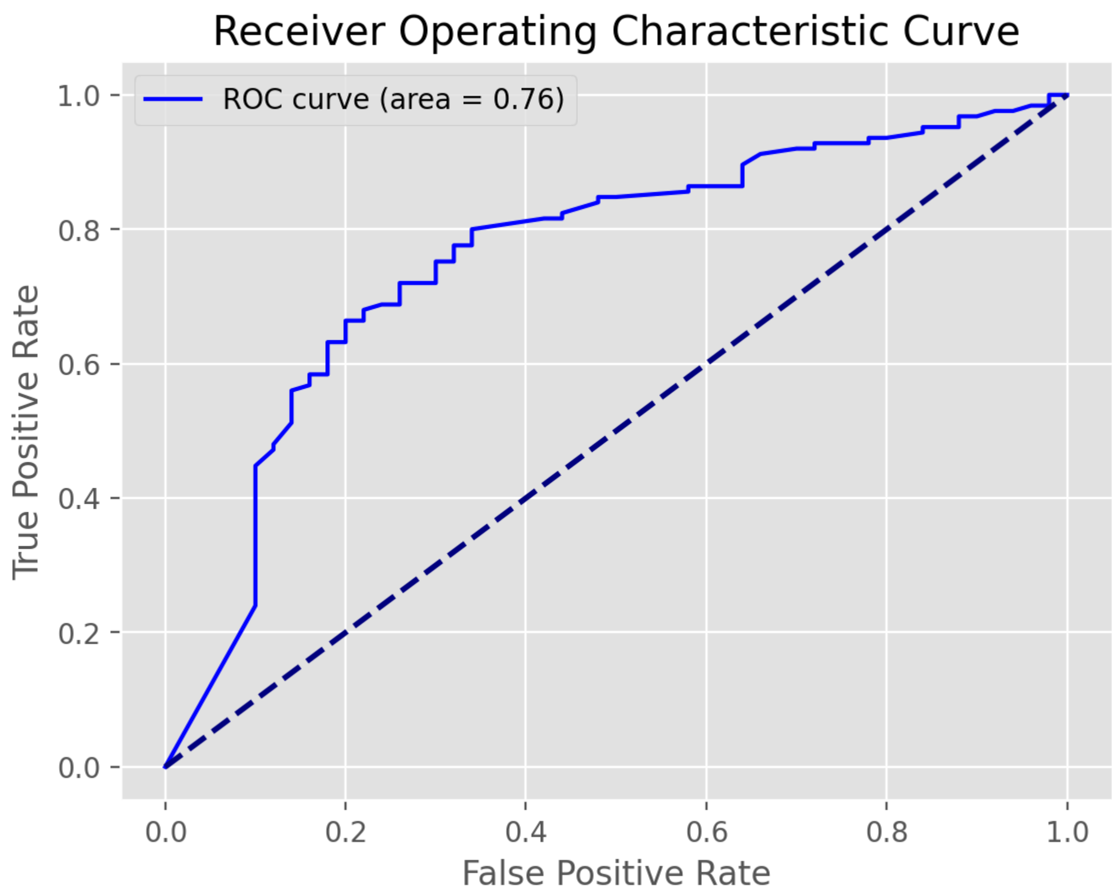
```
Data_train_LGBM, Data_test_LGBM, t_train_LGBM, t_test_LGBM = train_test_split(X_
                                                                              te
                                                                              st


# Initialize and train the LightGBM model
LGBM = LGBMClassifier()

param_grid_LGBM = {
    'n_estimators':[10,15,20],
    'max_depth':[8,10,12],
    'reg_alpha':[0.1,0.5,1.0],
    'reg_lambda':[0.1,0.5,1.0],
    'num_leaves':[6,8,10]
}

# Initialize GridsearchCV
grid_search = GridSearchCV(estimator = LGBM, param_grid = param_grid_LGBM , cv =

# Fit GridSearchCV to find the best hyperparameters
grid_search.fit(Data_train_LGBM, t_train_LGBM)

# Get the best model
best_LGBM = grid_search.best_estimator_


# Evaluating the moodel
train_accuracy = best_LGBM.score(Data_train_LGBM,t_train_LGBM)
test_accuracy = best_LGBM.score(Data_test_LGBM,t_test_LGBM)
print("Training accuracy:", train_accuracy)
print("Testing accuracy:", test_accuracy)


# Generating and printing confusion matrix
predictions = best_LGBM.predict(Data_test_LGBM)
conf_matrix = confusion_matrix(t_test_LGBM, predictions)
print("Confusion Matrix:")
print(conf_matrix)

# Generating and printing classification matrix
class_report = classification_report(t_test_LGBM, predictions)
print("Classification Report:")
print(class_report)
```

```
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Number of positive: 291, number of negative: 117
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing
was 0.000242 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 299
[LightGBM] [Info] Number of data points in the train set: 408, number of used fea
tures: 5
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.713235 -> initscore=0.911149
[LightGBM] [Info] Start training from score 0.911149
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
Training accuracy: 0.7647058823529411
Testing accuracy: 0.72
Confusion Matrix:
[[  5  45]
 [  4 121]]
Classification Report:
              precision    recall  f1-score   support

           0       0.56      0.10      0.17        50
           1       0.73      0.97      0.83       125

    accuracy                           0.72       175
   macro avg       0.64      0.53      0.50       175
weighted avg       0.68      0.72      0.64       175
```

Using the train_test_split function, the dataset is divided between training and testing sets, with 30% set aside for testing. Following that, a LightGBM classifier is initialized, and a parameter grid for hyperparameter adjustment is established. GridSearchCV is then used to cross-validate and determine the best hyperparameters. To assess performance, the best model obtained from grid searching is evaluated on both the training and testing datasets. The model has a training accuracy of around 76.47% and a testing accuracy of 72.00%.

In [38]:

```
# Predicting probabilities for the positive class
probs_lgbm = best_LGBM.predict_proba(Data_test_LGBM)[:,1]

# Computing ROC curve and AUC score
fpr_lgbm, tpr_lbm, thresholds = roc_curve(t_test_LGBM,probs_lgbm)
auc_score_lgbm = roc_auc_score(t_test_LGBM,probs_lgbm)


# Plot ROC curve for the voting classifier
plt.plot(fpr_lgbm, tpr_lbm, color='blue', label='ROC curve (area = %0.2f)' % auc
plt.plot([0,1],[0,1], color = 'navy', lw = 2 , linestyle = '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic Curve")
plt.legend()
plt.show()

# Calculate the area under the ROC curve (AUC) for the voting classifier
print("AUC for LGBM:", auc_score_lgbm)
```




```
AUC for LGBM: 0.7588
```

The AUC (Area Under the Curve) value measures the classifier's overall performance, with a larger AUC suggesting superior discrimination abilities. In this scenario, the LightGBM model has an AUC value of roughly 0.78, indicating that it has moderate discriminatory ability in distinguishing between positive and negative classes.

LIGHT GBM with undersampling positive class

In [39]:

```
Data_train_LGBM, Data_test_LGBM, t_train_LGBM, t_test_LGBM = train_test_split(X_
                                                                                te
                                                                                st

# Applying Random Undersampling for class 1
undersample = RandomUnderSampler(sampling_strategy =1,random_state = 999)
Data_train_undersampled, t_train_undersampled = undersample.fit_resample(Data_tr


# Initialize and train the LightGBM model
LGBM = LGBMClassifier()

param_grid_LGBM = {
    'n_estimators':[10,15,20],
    'max_depth':[8,10,12],
    'reg_alpha':[0.1,0.5,1.0],
    'reg_lambda':[0.1,0.5,1.0],
    'num_leaves':[6,8,10]
}
```

```
# Initialize GridsearchCV
grid_search = GridSearchCV(estimator = LGBM, param_grid = param_grid_LGBM , cv =

# Fit GridSearchCV to find the best hyperparameters
grid_search.fit(Data_train_undersampled, t_train_undersampled)

# Get the best model
best_LGBM_2 = grid_search.best_estimator_


# Evaluating the moodel
train_accuracy = best_LGBM_2.score(Data_train_undersampled, t_train_undersampled
test_accuracy = best_LGBM_2.score(Data_test_LGBM,t_test_LGBM)
print("Training accuracy:", train_accuracy)
print("Testing accuracy:", test_accuracy)


# Generating and printing confusion matrix
predictions = best_LGBM_2.predict(Data_test_LGBM)
conf_matrix = confusion_matrix(t_test_LGBM, predictions)
print("Confusion Matrix:")
print(conf_matrix)

# Generating and printing classification matrix
class_report = classification_report(t_test_LGBM, predictions)
print("Classification Report:")
print(class_report)
```

```
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Number of positive: 133, number of negative: 133
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing
was 0.000345 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 208
[LightGBM] [Info] Number of data points in the train set: 266, number of used fea
tures: 5
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscore=0.000000
Training accuracy: 0.7443609022556391
Testing accuracy: 0.6752136752136753
Confusion Matrix:
[[26  8]
 [30 53]]
Classification Report:
              precision    recall  f1-score   support

           0       0.46      0.76      0.58        34
           1       0.87      0.64      0.74        83

    accuracy                           0.68       117
   macro avg       0.67      0.70      0.66       117
weighted avg       0.75      0.68      0.69       117
```

In this revised strategy, Random Undersampling was used to balance the class distribution in the training set by randomly clearing instances from the majority class (class 1 in this case) to match the number of instances in the minority class (class 0). After resampling, the LightGBM model was trained on the undersampled training data. Despite the increased class balance, testing accuracy was slightly lower than in the previous model, indicating a trade-off between class balance and overall predictive ability.

According to the model, 26 instances of class 0 (negative class) and 53 instances of class 1 (positive class) were correctly classified. Class 0 was misclassified 30 instances and class 1 was misclassified 8 instances.

The classification report includes more specific performance information for each class. For class 0, its precision (the classifier's ability not to classify a sample as positive when it is negative) is 0.46, suggesting that only 46% of all occurrences predicted as class 0 were in fact class 0. The classifier's recall (capacity to discover all positive samples) is 0.76, indicating that the model successfully identified 76% of all actual class 0 cases. The F1-score, or harmonic mean of precision and recall, is 0.58 for class 0.

For class 1, the precision is 0.87, which means that 87% of all occurrences predicted as class 1 are truly class 1. The recall is 0.64, which suggests the model successfully identified 64% of all actual class 1 cases. The F1 score for class 1 is 0.74.

The model's overall accuracy, defined as the proportion of properly categorized occurrences among all instances, is 0.68. Precision, recall, and F1-score are aggregated performance metrics over both classes, with the macro average taking into account equal relevance for each class (macro average) and the weighted average reflecting class imbalance.

In [40]:

```
# Get the predicted probabilities for the positive class
predicted_probabilities = best_LGBM_2.predict_proba(Data_test_LGBM)[:,1]

# Calculate the AUC score
auc_score = roc_auc_score(t_test_LGBM, predicted_probabilities)
print("AUC Score:", auc_score)
```

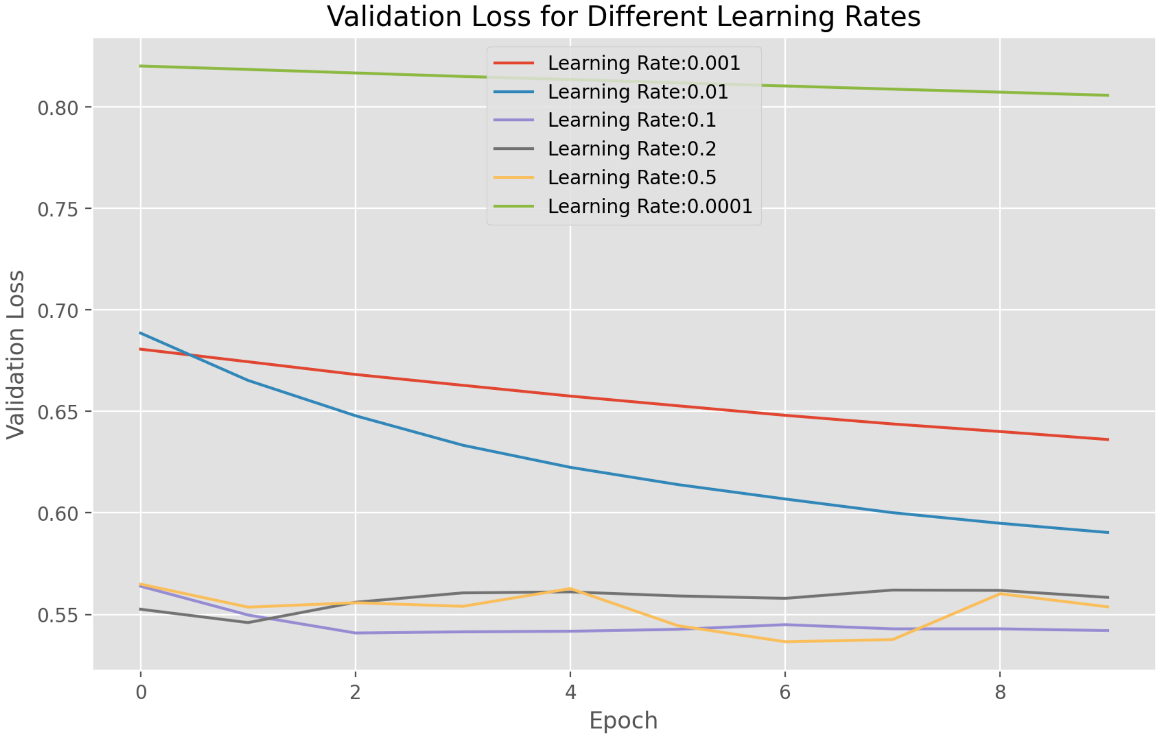
```
AUC Score: 0.7312189936215451
```

Neural Networks Model

In [35]:

```
N_train, N_test , n_train, n_test = train_test_split(X_selected, Y, test_size =

learning_rates = [0.001,0.01, 0.1,0.2,0.5,0.0001]

losses_per_lr = []

for lr in learning_rates:
    temp_optimizer = SGD(learning_rate= lr, momentum = 0.5)
    temp_model = Sequential()
    temp_model.add(Dense(64, input_shape=(N_train.shape[-1],), activation = 'rel
    temp_model.add(Dropout(0.2))
    temp_model.add(Dense(32, activation = "relu"))
    temp_model.add(Dropout(0.2))
    temp_model.add(Dense(16, activation = "relu"))
    temp_model.add(Dropout(0.2))
    temp_model.add(Dense(1, activation = "sigmoid"))
    temp_model.compile(loss = "binary_crossentropy", optimizer = temp_optimizer,
    history = temp_model.fit(N_train, n_train, epochs = 10, batch_size = 32, val
    losses_per_lr.append(history.history["val_loss"])

plt.figure(figsize = (10,6))
```

```
for i, lr in enumerate(learning_rates):
    plt.plot(losses_per_lr[i], label = f'Learning Rate:{lr}')
    plt.title("Validation Loss for Different Learning Rates")
    plt.xlabel('Epoch')
    plt.ylabel("Validation Loss")
    plt.legend()
plt.show()
```

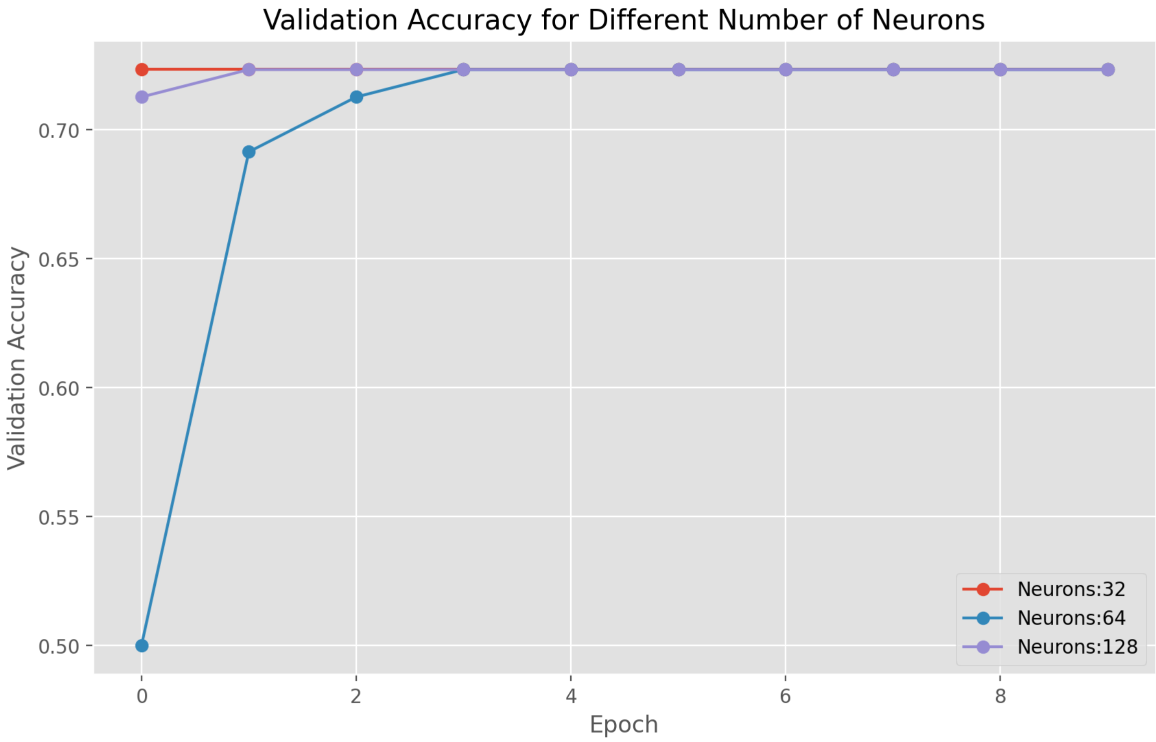


The experiment with different learning rates gives useful information about how the learning rate you choose influences the training dynamics and performance of your neural network model. Learning rates 0.001, 0.01, 0.1, 0.2, 0.5, and 0.0001 were found to result in decreased validation loss throughout the training phase. This suggests that higher learning rates are more suited to properly updating model parameters while avoiding excessive overfitting in the optimization process.

On the other hand, learning rates 0.1, 0.2, and 0.5 behave similarly, with validation loss being considerably lower. However, at epoch 4, learning rate 0.1 had the lowest validation loss in this group, implying that it may have achieved a better compromise between convergence speed and stability at that point in training.

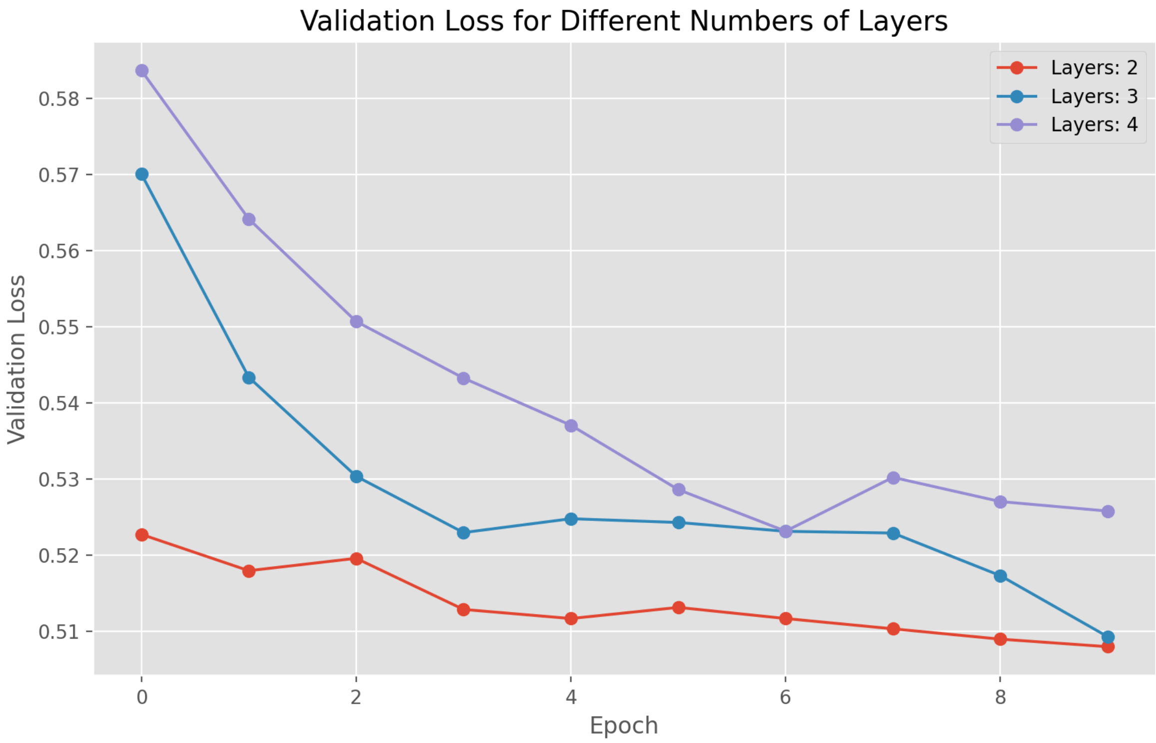
In [23]:
```
neurons = [32,64,128]
accuracies_per_neuron = []

N_train, N_test , n_train, n_test = train_test_split(X_selected, Y, test_size =

for n in neurons:
    optimizer = SGD(learning_rate = 0.01, momentum = 0.5)
    model = Sequential()
    model.add(Dense(128, input_shape=(N_train.shape[-1],), activation = 'relu'))
    model.add(Dropout(0.2))
    model.add(Dense(64, activation = "relu"))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation = "relu"))
```

```
    model.add(Dropout(0.2))
    model.add(Dense(1, activation = "sigmoid"))
    model.compile(loss = "binary_crossentropy", optimizer = optimizer, metrics =

    history = model.fit(N_train, n_train, epochs = 10, batch_size = 32, validati
    accuracies_per_neuron.append(history.history["val_accuracy"])

plt.figure(figsize = (10,6))

for i, n in enumerate(neurons):
    plt.plot(accuracies_per_neuron[i], label = f'Neurons:{n}', marker = 'o')
    plt.title("Validation Accuracy for Different Number of Neurons")
    plt.xlabel('Epoch')
    plt.ylabel("Validation Accuracy")
    plt.legend()
plt.show()
```



In [34]:
```
layers = [2, 3, 4]
losses_per_layer = []

for l in layers:
    optimizer = SGD(learning_rate=0.1, momentum=0.5)

    model = Sequential()
    model.add(Dense(64, input_shape=(N_train.shape[1],), activation='relu'))
    model.add(Dropout(0.2))
    for _ in range(l - 1):
        model.add(Dense(32, activation="relu"))
        model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['acc
    history = model.fit(N_train, n_train, epochs=10, batch_size=32, validation_s
    losses_per_layer.append(history.history['val_loss'])

# Plotting the results
plt.figure(figsize=(10, 6))
```

```
for i, l in enumerate(layers):
    plt.plot(losses_per_layer[i], label=f'Layers: {l}', marker = 'o')

plt.title('Validation Loss for Different Numbers of Layers')
plt.xlabel('Epoch')
plt.ylabel('Validation Loss')
plt.legend()
plt.show()
```

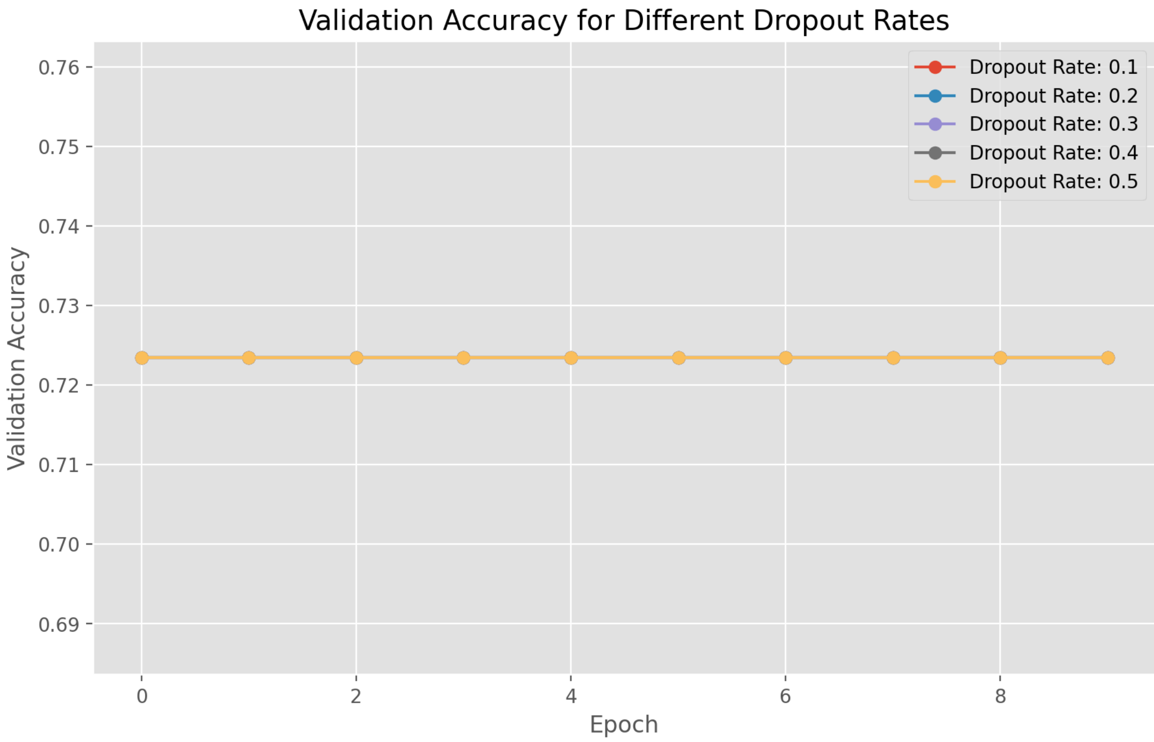


The analysis highlights the impact of layering on validation loss during model training. Among the configurations tested - 2, 3, and 4 layers - layer 2 consistently had the lowest validation loss, especially during epochs 3 to 7. A neural network architecture with two hidden layers may be more suitable for this task than deeper architectures. The validation loss for layer 4 was consistently higher than for the other configurations, as well. As a result of these findings, it was concluded that the best configuration is a neural network with two hidden layers trained for four epochs.

In [27]:

```
drop_rates = [0.1,0.2,0.3,0.4,0.5]
accuracies_per_dropout = []

for dr in drop_rates:
    optimizer_n = SGD(learning_rate = 0.1, momentum = 0.5)
    model_n = Sequential()
    model_n.add(Dense(64, input_shape=(N_train.shape[-1],), activation = 'relu')
    model_n.add(Dropout(dr))
    model_n.add(Dense(32, activation = "relu"))
    model_n.add(Dropout(dr))
    model_n.add(Dense(16, activation = "relu"))
    model_n.add(Dropout(dr))
    model_n.add(Dense(1, activation = "sigmoid"))
    model_n.compile(loss = "binary_crossentropy", optimizer = optimizer_n, metri

    history = model_n.fit(N_train, n_train, epochs=10, batch_size=32, validation
    accuracies_per_dropout.append(history.history['val_accuracy'])
```

```
plt.figure(figsize=(10, 6))

for i, dr in enumerate(drop_rates):
    plt.plot(accuracies_per_dropout[i], label=f'Dropout Rate: {dr}', marker = 'c

plt.title('Validation Accuracy for Different Dropout Rates')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.show()
```

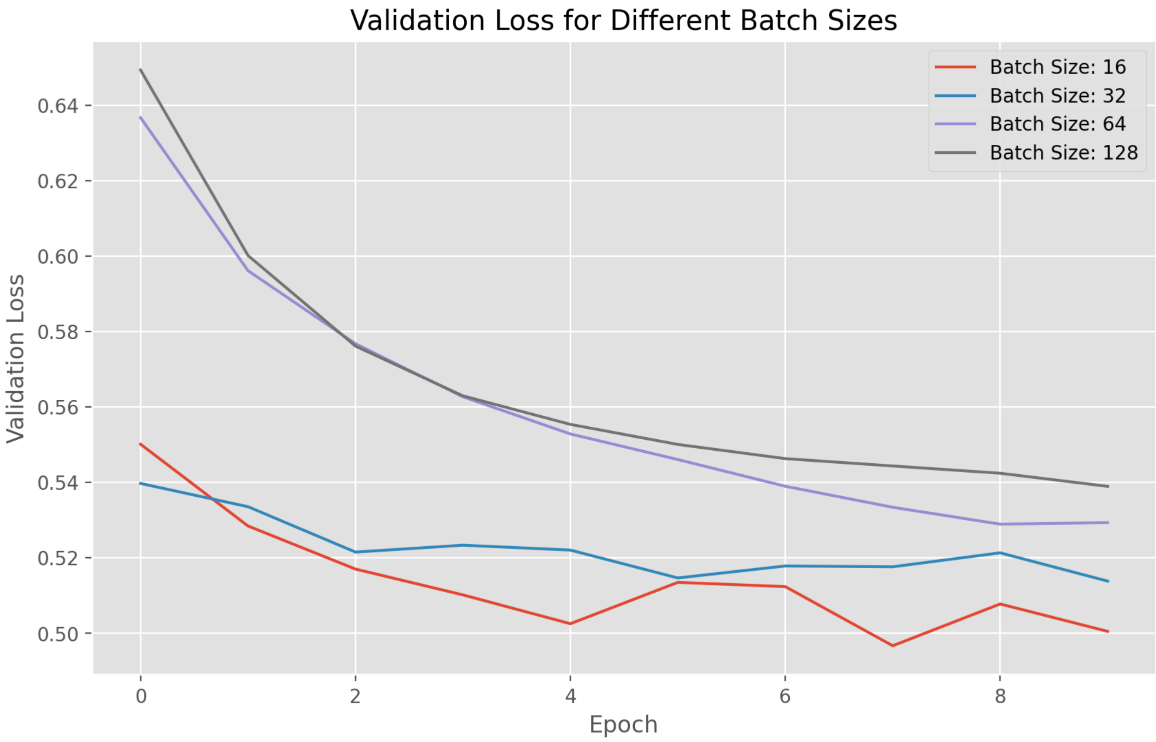


The experiment with varying dropout rates sheds light on how dropout regularization influences the training dynamics and performance of your neural network model. Surprisingly, all evaluated dropout rates (0.1, 0.2, 0.3, 0.4, and 0.5) showed identical validation accuracy trends throughout the training procedure. This finding implies that, for this specific model architecture and dataset, changing the dropout rate has no meaningful effect on the model's capacity to generalize to previously unseen data.

In [126…

```
batch_sizes = [16, 32, 64, 128]
losses_per_batch = []

for bs in batch_sizes:
    optimizer = SGD(learning_rate=0.1, momentum=0.5)
    model = Sequential()
    model.add(Dense(64, input_shape=(N_train.shape[1],), activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation="relu"))
    model.add(Dropout(0.2))
    model.add(Dense(16, activation="relu"))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['acc
    history = model.fit(N_train, n_train, epochs=10, batch_size=bs, validation_s
    losses_per_batch.append(history.history['val_loss'])
plt.figure(figsize=(10, 6))
```

```
for i, bs in enumerate(batch_sizes):
    plt.plot(losses_per_batch[i], label=f'Batch Size: {bs}')

plt.title('Validation Loss for Different Batch Sizes')
plt.xlabel('Epoch')
plt.ylabel('Validation Loss')
plt.legend()
plt.show()
```



Over successive iterations, the model refines its parameters to better capture the underlying patterns in the data, as validated by the decreasing validation loss as epochs increase. At epoch 4, batch size 16 displayed the lowest validation loss, exhibiting the best performance. Therefore, smaller batch sizes may facilitate convergence to an optimal or near-optimal solution within a shorter timeframe. This result suggests that smaller batches may offer advantages in terms of faster convergence, allowing the model to extract more relevant features from the data earlier in the training process.

In [41]:
```
N_train, N_test , n_train, n_test = train_test_split(X_selected, Y, test_size =

classification_reports_nn = []
confusion_matrices_nn = []

train_accuracies_nn = []
test_accuracies_nn = []

learning_rate = 0.1


kf = KFold(n_splits=5, shuffle=True, random_state=999)
accuracies = []

for train_index, test_index in kf.split(X_selected):
    N_train, N_test = X_selected[train_index], X_selected[test_index]
    n_train, n_test = Y[train_index], Y[test_index]
```

```
    optimizer = SGD(learning_rate = learning_rate)
    # Create a sequential model
    model_nn = Sequential()

    # Add a dense layer
    model_nn.add(Dense(32, input_shape=(N_train.shape[1],),activation = 'relu'))
    model_nn.add(Dropout(0.5))
    model_nn.add(Dense(16, activation = "relu"))
    model_nn.add(Dropout(0.5))
    model_nn.add(Dense(1, activation = 'sigmoid'))
    # Compile the model
    model_nn.compile(loss = "binary_crossentropy", optimizer = optimizer , metri

    # Display model summary
    model_nn.summary()
    model_nn.fit(N_train, n_train, epochs=4, batch_size=16, validation_split=0.2
    accuracy = model_nn.evaluate(N_test, n_test, verbose =0)[1]
    accuracies.append(accuracy)

    train_accuracy_nn =  model_nn.evaluate(N_train, n_train, verbose = 0)[1]
    test_accuracy_nn = model_nn.evaluate(N_test, n_test, verbose = 0)[1]

    train_accuracies_nn.append(train_accuracy_nn)
    test_accuracies_nn.append(test_accuracy_nn)

    n_pred_prob = model_nn.predict(N_test)
    n_pred = (n_pred_prob >0.5).astype(int)

    report = classification_report(n_test, n_pred)
    classification_reports_nn.append(report)

    matrix = confusion_matrix(n_test, n_pred)
    confusion_matrices_nn.append(matrix)
```

Model: "sequential_57"

Layer (type)	Output Shape	
dense_218 (Dense)	(None, 32)	
dropout_161 (Dropout)	(None, 32)	
dense_219 (Dense)	(None, 16)	
dropout_162 (Dropout)	(None, 16)	
dense_220 (Dense)	(None, 1)	

Total params: 737 (2.88 KB)

Trainable params: 737 (2.88 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.5493 - loss: 0.6983 - val_ac
curacy: 0.7660 - val_loss: 0.5225
Epoch 2/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7086 - loss: 0.5897 - val_ac
curacy: 0.7660 - val_loss: 0.4942
Epoch 3/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7042 - loss: 0.5873 - val_ac
curacy: 0.7660 - val_loss: 0.4853
Epoch 4/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6783 - loss: 0.5790 - val_ac
curacy: 0.7660 - val_loss: 0.4713
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step
```

Model: "sequential_58"

Layer (type)	Output Shape	
dense_221 (Dense)	(None, 32)	
dropout_163 (Dropout)	(None, 32)	
dense_222 (Dense)	(None, 16)	
dropout_164 (Dropout)	(None, 16)	
dense_223 (Dense)	(None, 1)	

Total params: 737 (2.88 KB)

Trainable params: 737 (2.88 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5500 - loss: 0.7094 - val_ac
curacy: 0.7447 - val_loss: 0.5224
Epoch 2/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7059 - loss: 0.6242 - val_ac
curacy: 0.7447 - val_loss: 0.5029
Epoch 3/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6679 - loss: 0.6188 - val_ac
curacy: 0.7447 - val_loss: 0.4821
Epoch 4/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6900 - loss: 0.5980 - val_ac
curacy: 0.7447 - val_loss: 0.4791
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
```

Model: "sequential_59"

Layer (type)	Output Shape	
dense_224 (Dense)	(None, 32)	
dropout_165 (Dropout)	(None, 32)	
dense_225 (Dense)	(None, 16)	
dropout_166 (Dropout)	(None, 16)	
dense_226 (Dense)	(None, 1)	

Total params: 737 (2.88 KB)

Trainable params: 737 (2.88 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.5680 - loss: 0.6712 - val_a
ccuracy: 0.7340 - val_loss: 0.5111
Epoch 2/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7469 - loss: 0.5592 - val_ac
curacy: 0.7340 - val_loss: 0.4947
Epoch 3/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.6649 - loss: 0.5902 - val_ac
curacy: 0.7340 - val_loss: 0.4888
Epoch 4/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7015 - loss: 0.5951 - val_ac
curacy: 0.7340 - val_loss: 0.4841
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
```

Model: "sequential_60"

Layer (type)	Output Shape	
dense_227 (Dense)	(None, 32)	
dropout_167 (Dropout)	(None, 32)	
dense_228 (Dense)	(None, 16)	
dropout_168 (Dropout)	(None, 16)	
dense_229 (Dense)	(None, 1)	

Total params: 737 (2.88 KB)

Trainable params: 737 (2.88 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.6096 - loss: 0.6516 - val_a
ccuracy: 0.7128 - val_loss: 0.5560
Epoch 2/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6776 - loss: 0.6491 - val_ac
curacy: 0.7128 - val_loss: 0.5267
Epoch 3/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6863 - loss: 0.5937 - val_ac
curacy: 0.7128 - val_loss: 0.5162
Epoch 4/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7206 - loss: 0.5453 - val_ac
curacy: 0.7128 - val_loss: 0.5108
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
```

Model: "sequential_61"

Layer (type)	Output Shape
dense_230 (Dense)	(None, 32)
dropout_169 (Dropout)	(None, 32)
dense_231 (Dense)	(None, 16)
dropout_170 (Dropout)	(None, 16)
dense_232 (Dense)	(None, 1)

Total params: 737 (2.88 KB)

Trainable params: 737 (2.88 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.7062 - loss: 0.6068 - val_a
ccuracy: 0.7021 - val_loss: 0.5552
Epoch 2/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6793 - loss: 0.6032 - val_ac
curacy: 0.7021 - val_loss: 0.5499
Epoch 3/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6778 - loss: 0.6335 - val_ac
curacy: 0.7021 - val_loss: 0.5451
Epoch 4/4
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.6728 - loss: 0.5967 - val_ac
curacy: 0.7021 - val_loss: 0.5400
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
```

Neural network models was trained using K-Fold cross-validation, comprising two dense layers with 128 and 64 neurons, respectively, followed by dropout regularization with a dropout rate of 0.5 to mitigate overfitting. Binary classification tasks were performed using a single neuron with sigmoid activation. During training, the models showed converging behavior, as evidenced by decreasing losses and increasing accuracy over epochs. The accuracies across test sets varied across folds, suggesting that different data splits contributed to performance variability. The classification reports and confusion matrices provided detailed insight into model performance, showing how well they classified instances within each class.

In [42]:
```
# Print classification reports and consfusion matrices for each fold

for i in range(len(classification_reports_nn)):
    print(f"\nClassification Report for Fold {i+1}:\n{classification_reports_nn[
    print(f"\nConfusion Matrix for Fold {i+1}:\n{confusion_matrices_nn[i]}")

# Calculate and print the average accuracy
average_accuracy = np.mean(accuracies)
print("\nAverage accuracy:", average_accuracy)
print(accuracies)


print("Train_accuracies:", train_accuracies_nn)
print("Test_accuracies:", test_accuracies_nn)
```

```
Classification Report for Fold 1:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        37
           1       0.68      1.00      0.81        80

    accuracy                           0.68       117
   macro avg       0.34      0.50      0.41       117
weighted avg       0.47      0.68      0.56       117


Confusion Matrix for Fold 1:
[[ 0 37]
 [ 0 80]]

Classification Report for Fold 2:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        33
           1       0.72      1.00      0.84        84

    accuracy                           0.72       117
   macro avg       0.36      0.50      0.42       117
weighted avg       0.52      0.72      0.60       117


Confusion Matrix for Fold 2:
[[ 0 33]
 [ 0 84]]

Classification Report for Fold 3:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        34
           1       0.71      1.00      0.83        83

    accuracy                           0.71       117
   macro avg       0.35      0.50      0.41       117
weighted avg       0.50      0.71      0.59       117


Confusion Matrix for Fold 3:
[[ 0 34]
 [ 0 83]]

Classification Report for Fold 4:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        30
           1       0.74      1.00      0.85        86

    accuracy                           0.74       116
   macro avg       0.37      0.50      0.43       116
weighted avg       0.55      0.74      0.63       116


Confusion Matrix for Fold 4:
[[ 0 30]
 [ 0 86]]
```

```
Classification Report for Fold 5:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        33
           1       0.72      1.00      0.83        83

    accuracy                           0.72       116
   macro avg       0.36      0.50      0.42       116
weighted avg       0.51      0.72      0.60       116


Confusion Matrix for Fold 5:
[[ 0 33]
 [ 0 83]]

Average accuracy: 0.713601541519165
[0.6837607026100159, 0.7179487347602844, 0.7094017267227173, 0.7413793206214905,
0.7155172228813171]
Train_accuracies: [0.721030056476593, 0.7124463319778442, 0.7145922780036926, 0.7
066380977630615, 0.7130621075630188]
Test_accuracies: [0.6837607026100159, 0.7179487347602844, 0.7094017267227173, 0.7
413793206214905, 0.7155172228813171]
```

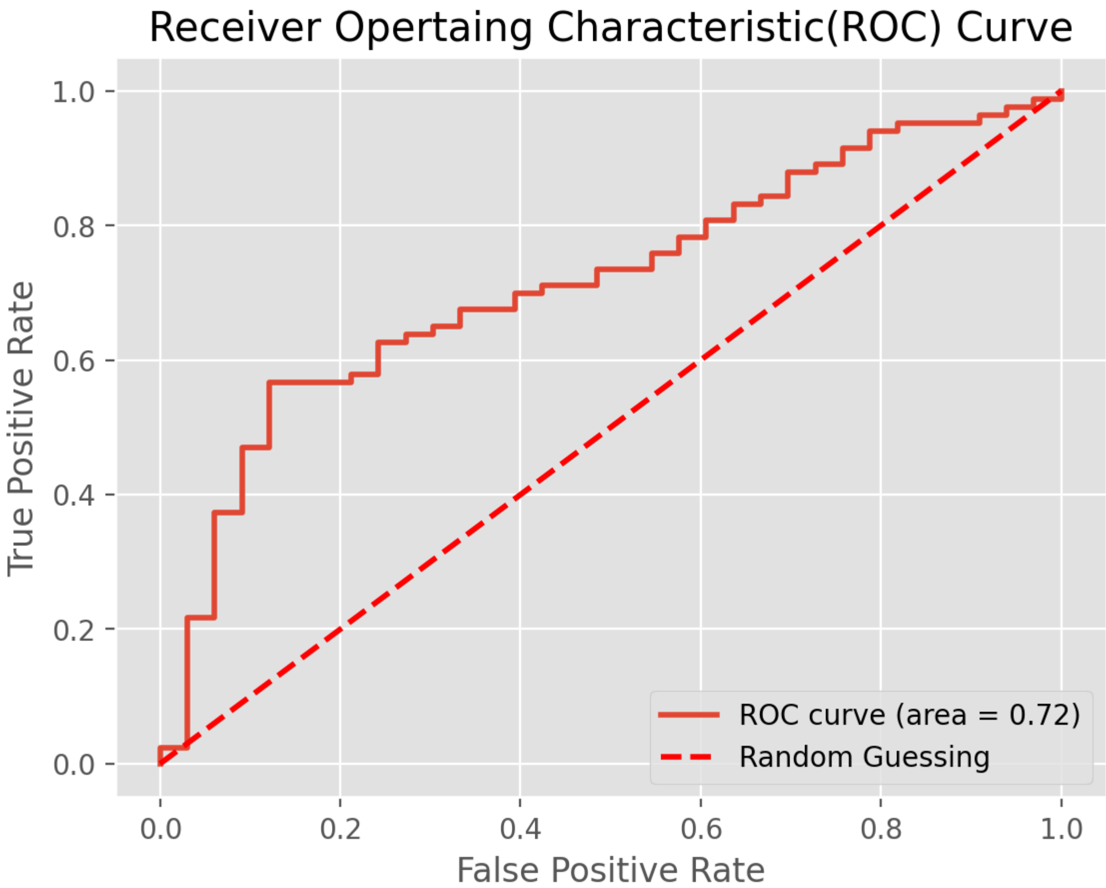
The provided code shows how k-fold cross-validation can be used to evaluate neural networks. Each fold serves as both the validation set and the test set exactly once, in this approach. Several subsets of data are used in the training and testing of the neural network. Each fold provides classification reports and confusion matrices, providing detailed information on the model's accuracy, recall, and precision.

Classification reports provide detailed insights into the model's performance across five folds, including precision, recall, and F1-score. In class 1, precision scores range from 0.68 to 0.74, which demonstrates consistency across folds. For class 1, recall scores consistently achieve a perfect 1, indicating the model captures all positive instances. For class 1, F1 scores range from 0.81 to 0.85, reflecting a balance between precision and recall. In terms of precision and recall, this consistency indicates the model is reliable in correctly classifying positive instances.

According to the classification reports, the model fails to capture instances of the negative class (class 0). Class 0 consistently registers a F1-score of 0.00 across all folds, indicating that no true negatives are identified. It seems that the model has a bias towards predicting positive instances, as evidenced by its consistently poor performance in detecting the negative class. While the model is very accurate in classifying positive instances, it is less accurate in identifying negative ones, raising concerns about its reliability in real-world scenarios where both classes must be accurately predicted.It is therefore necessary to further refine the model to achieve a more balanced and reliable predictive capability across both classes, even though it may excel in certain contexts.

In [43]:
```
n_test_last_fold = n_test
n_pred_prob_last_fold = n_pred_prob

fpr_nn, tpr_nn, thresholds = roc_curve(n_test_last_fold,n_pred_prob_last_fold )
roc_auc = auc(fpr_nn, tpr_nn)

plt.plot(fpr_nn, tpr_nn, lw =2, label = "ROC curve (area = %0.2f)"% roc_auc)
```

```
# Plot ROC curve for random guessing
plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', label='Random Guessing

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Opertaing Characteristic(ROC) Curve")
plt.legend(loc = "lower right")
plt.show()
```



Model Comparison

In class 1, logistic regression achieves an accuracy of 0.766 and a precision of 0.78, demonstrating its accuracy in identifying positive cases. However, it struggles with recall for class 0, indicating that true negatives are difficult to capture. While Logistic Regression achieves an accuracy of 0.760, Classification Tree achieves 0.760 with improved precision and recall for both classes. With an accuracy of 0.771, the Voting Classifier outperforms both Logistic Regression and Classification Tree. With this approach, the precision for both classes remains relatively high while the recall for class 0 is significantly improved, which results in a more balanced performance. In addition, the confusion matrix of the Voting Classifier shows fewer false negatives than the other models for class 0, suggesting that this class is more sensitive than the others. Based on these results, the Voting Classifier emerges as the best model, offering a robust balance between precision, recall, and overall accuracy.

The Voting Classifier outperforms the LGBM and XGBoost models based on the metrics provided. The Voting Classifier has a higher accuracy score of roughly 0.77, compared to the LGBM model's accuracy of about 0.72 and the XGBoost model's accuracy of about 0.71. The Voting Classifier also has superior precision, recall, and F1-score in both classes (0 and 1). In terms of the confusion matrix, the Voting Classifier has less misclassifications (false positives and false negatives) than the Neural Networks model, LGBM and XGBoost model. According to these criteria, the Voting Classifier is the better-performing model in this circumstance.

Critique & Limitations

Strengths:

1. Hyperparameter Tuning: Utilization of GridSearchCV to tune hyperparameters to optimize the performance of each model. A fine-tuned model is more likely to generalize well to new data, so it is more likely to perform well in new data.

2. Cross-validation: All models were cross-validated, ensuring robust model evaluation and reducing overfitting risks. Our models showed little difference between training and test accuracy scores, indicating strong generalization performance. This consistency shows that the models efficiently learned from the training data without overfitting, resulting in reasonable predictions for unseen test data.

3. Detailed Evaluation Metrics: For each model, the report was computed and reported with various evaluation metrics, such as accuracy, confusion matrix, and classification report. As a result, we can gain a comprehensive understanding of the performance of the models.

4. Utilization of Different Algorithms: A variety of algorithms were tested, including Decision Trees, Logistic Regression, Random Forests, XGBoost, and LightGBM. Comparing different modeling approaches allows for deeper insights and more accurate model selection.

5. Handling of Imbalanced Data: By setting the stratify parameter in train_test_split, we explicitly addressed class imbalance in LightGBM. The training and test sets are thus maintained at the same class distribution as the original data.

6. In many situations, the Voting Classifier, which consists of simpler models such as Logistic Regression and Decision Trees, outperforms black-box models. For example, its constituent models are often more interpretable, giving stakeholders insights into the underlying decision-making process, which is critical in fields where predictions must be transparent. Furthermore, the ensemble approach reduces overfitting by collecting several features of the data distribution, resulting in more accurate predictions on previously unseen data. Furthermore, the Voting Classifier is less vulnerable to outliers because to the resilience of its simpler models, ensuring consistent performance even in noisy data. This robustness makes the Voting Classifier appropriate for dealing with imbalanced data, where biases toward the

majority class may have an impact on model performance. The Voting Classifier, which combines multiple base models with varying learning biases, provides a balanced and reliable approach for solving imbalanced data problems, as seen in the confusion and classification report for the voting classifier model, while maintaining scalability and computing efficiency.

7. Neural Networks: This neural network model performs admirably in identifying positive instances (class 1), as demonstrated by its high precision, recall, and F1-scores. Specifically, the model achieves F1-scores between 0.81 and 0.85 for class 1, precision scores of 0.68 to 0.74, and recall scores of 1.0. Consequently, the model is highly effective at identifying positive instances, which is crucial for applications in which detecting positive instances is critical.

Weaknesses:

1. Imbalanced Data Handling: The class imbalance in the dataset is not addressed. In situations where positive class considerably outnumbers the negative class, models has demonstrate biased behavior in favor of the positive class.

2. A further limitation found among the algorithms that they tend to predict the majority class (positive class) with greater frequency than the minority class (negative class). This is especially important in circumstances with imbalanced datasets, where one class is much more common than the other. The models' bias towards the majority class can cause an imbalance in predictions, with a greater proportion of events categorized as positive. As a result, predictive performance measurements may not adequately reflect the models' ability to categorize examples from the minority class, thus overestimating total model performance."

3. Optimization Limitations: GridSearchCV's hyperparameter tuning procedure can be computationally demanding, particularly when dealing with large parameter grids and data sets. Due to computational constraints, exploring a large number of hyperparameter combinations may not have been possible, thereby limiting the optimization process and limiting the finding of optimal hyperparameters.

4. Some of the models utilized, such as Random Forest, XGBoost, and LightGBM, are considered black-box models, limiting interpretability when compared to simpler models like Logistic Regression or Decision Trees. Unlike simpler models such as Logistic Regression or Decision Trees, which provide obvious criteria for making predictions, black-box models conceal the underlying mechanisms that drive their predictions. This lack of transparency can be a disadvantage in situations such as liver illness. Biased predictions in a sensitive domain such as healthcare may result in inequitable treatment or misdiagnoses, can raise serious ethical implications.

5. Neural Networks: It is important to note that this outstanding performance on class 1 has a significant cost. Across all folds, the model fails to identify any instances of the negative class (class 0), resulting in a consistently poor F1-score of 0.00. In real-world scenarios in which accurate identification of both classes is crucial, this could pose a substantial bias towards predicting positive instances. According to the

classification reports, the model does not capture any true negatives, indicating overconfidence.

Summary & Conclusions

Project Summary:

In this project, we aimed to develop and evaluate machine learning models for the diagnosis of liver disease. The project comprised two main phases:

In Phase 1 gain a better understanding of the dataset's properties and distributions, exploratory data analysis was conducted to clean up the dataset, correct missing values, and encode categorical variables. As a result of identifying and addressing various data quality issues, including missing values, outliers, and inconsistencies, we significantly improved data quality. As a result of EDA, we gained valuable insights into variable distributions, correlations, and potential relationships between features. Understanding the data structure and identifying important patterns helped us understand the data. Several preprocessing techniques were used to prepare the dataset for modeling, including normalization, scaling, and categorical coding. By using these techniques, the data could be standardized and machine learning algorithms could be adapted to work with it.

In Phase 2, the project focused on model development and evaluation, using a variety of machine learning methods such as Logistic Regression, Decision Trees, Random Forests, XGBoost, LightGBM, and a Voting Classifier. We refined model performance and examined predictive capacities by rigorously adjusting hyperparameters and cross-validation. In addition, we addressed class imbalance using Random Undersampling and conducted detailed model evaluations, resulting in insights into each model's strengths, limitations, and prospective improvements. Our findings provide practical advice for enhancing liver disease prediction models, ultimately advancing responsible and effective machine learning methods in healthcare and beyond.

Summary of Findings:

Using a dataset containing biochemical parameters, various machine learning models were trained and evaluated for liver disease classification. In addition to achieving moderate discriminatory abilities, the XGBoost had an AUC score of around 0.70, whereas LightGBM achieved an AUC score of about 0.76. These models demonstrated similar performance in distinguishing between positive and negative instances of liver disease despite their different approaches, including hyperparameter tuning and undersampling to address class imbalance. With an AUC score of 0.82, the Voting Classifier outperformed both XGBoost and LightGBM with a combination of simpler models, demonstrating better discrimination between positive and negative cases while maintaining a balance between precision, recall, and accuracy overall.Nonetheless, issues such as class imbalance and model interpretability emphasize the significance of continual refinement and ethical considerations in healthcare predictive modeling.

Concludion: In this study, machine learning models were developed and evaluated for the classification of liver disease based on biochemical parameters. In this study, the Voting Classifier, combining simpler models, had the highest AUC score, 0.82. It maintains a balance between precision, recall, and overall accuracy while accurately detecting positive and negative liver disease cases. In healthcare predictive modeling, model selection and optimization are critical factors, and continued refinement and ethical considerations must be considered in leveraging machine learning.

References

1. Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.
2. Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining (pp. 785-794).
3. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T. Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. In Advances in neural information processing systems (pp. 3146-3154).
4. H. Drucker, C. J. C. Burges, L. Kaufman, A. J. Smola, and V. Vapnik. “Support Vector Regression Machines.” In Advances in Neural Information Processing Systems, 1997. (PDF)
5. L. Breiman. “Bagging Predictors.” Machine Learning, 24(2), 1996. (PDF)
6. DataCamp. Machine Learning Scientist with Python. DataCamp, 2022. DataCamp website.

In [ ]: