TABLE OF CONTENT

1.0 EXECUTIVE SUMMARY..... 3
1.1 PROJECT DESCRIPTION.....3
1.2 PROBLEM TO BE SOLVED.....4
1.3 DATASET DESCRIPTION..... 5
2.0 SUMMARY OF PROJECT CONTEXT AND OBJECTIVES.....7
2.1 SUMMARY OF THE PROJECT CONTEXT.....7
2.2 OBJECTIVES.....7
3.0 METHODOLOGY.....8
3.1 DATA FRAMEWORK.....8
3.2 PACKAGES REQUIRED..... 9
3.3 DATA PREPROCESSING.....10
3.4 DATA EXPLORATION..... 13
3.5 MODELLING.....17
3.5.1 DECISION TREE..... 17
3.5.2 RANDOM FOREST.....21
3.5.3 SUPPORT VECTOR MACHINE.....26
3.5.4 LOGISTIC REGRESSION.....29
3.5.5 NAIVE BAYES.....32
3.6 MODEL COMPARISON.....35
3.7 USER GUI.....41
4.0 RESULT AND DISCUSSION.....44
5.0 CONCLUSION.....46
6.0 REFERENCES.....47
7.0 APPENDX----------48

1.0 EXECUTIVE SUMMARY

1.1 PROJECT DESCRIPTION

The escalating prevalence of Chronic Kidney Disease (CKD) in India underscores the urgent need for early detection and proactive management to prevent its progression to end-stage kidney disease. This research aims to leverage advanced data mining and machine learning techniques to discern patterns and estimate the potential occurrence of CKD within the Indian population.

The data for this study is sourced from 400 real-world clinical cases, encompassing 250 patients diagnosed with CKD and 150 without the condition. The dataset comprises 25 attributes, incorporating numerical values like age, blood pressure, and blood glucose levels, as well as categorical information such as the presence of hypertension, diabetes mellitus, and anemia. This detailed dataset provides a comprehensive view of patients' health profiles, forming a robust foundation for subsequent analysis.

The project employs the versatile Python programming language and the Scikit-learn library, known for its rich suite of machine-learning algorithms. In the preparatory phase, the dataset undergoes meticulous cleaning and preprocessing to address missing values and normalize the range of numerical inputs, ensuring the integrity and quality of the analysis.

Feature selection techniques, including recursive feature elimination and mutual information, are applied to identify the most significant predictors of CKD. This not only enhances predictive accuracy but also sheds light on key risk factors associated with the disease. Various machine learning models, such as decision trees, random forests, and support vector machines, logistic regression, and naive bayes, are evaluated through cross-validation to determine the most effective approach for this specific dataset.

The model's performance is rigorously assessed based on sensitivity, specificity, and overall accuracy in classifying patients correctly. The development of explainability

techniques, such as SHapley Additive exPlanations (SHAP), facilitates the interpretation of model decisions, fostering trust among healthcare practitioners. Ultimately, the project aims to develop a predictive tool empowering physicians to make informed decisions about a patient's risk for CKD, facilitating early treatment strategies and significantly improving patient quality of life while alleviating the burden on healthcare systems. In summary, the fusion of clinical expertise with advanced analytical methodologies strives to create a scalable, effective, and reliable solution for the early detection of Chronic Kidney Disease in India, with potentially transformative impacts on renal healthcare. The insights gained may contribute not only to improved patient outcomes but also serve as a foundation for future research in predictive healthcare analytics.

1.2 PROBLEM TO BE SOLVED

The goal of the initiative is to address the increasing problem of Chronic Kidney Disease (CKD) in India, where the ailment is becoming more common. The main focus is on the urgency of early detection and treatment, since the development of chronic kidney disease (CKD) into end-stage kidney disease puts a heavy strain on healthcare systems. The research uses machine learning and data mining approaches to create an advanced forecasting tool in order to solve this problem. By foreseeing the emergence of CKD in its early stages, this tool enables medical personnel to take preventative action. The project's goals are to lessen the burden on healthcare resources, improve patient outcomes, and improve overall quality of life. Essentially, the project endeavours to present an effective solution to the critical issue of delayed CKD diagnosis, addressing its impacts on both individuals and the broader healthcare scenario in India.

1.3 DATASET DESCRIPTION

The dataset used encompasses a comprehensive set of 25 attributes in a csv file, “ckd.csv”. The dataset is obtained from an open-source resource, GitHub, and is sufficiently and comprehensive to facilitate meaningful analysis. 400 real-world clinical patients information involved 250 patients diagnosed with chronic kidney disease while 150 patients were not. The numerical data and nominal data involved. The nominal data is qualitative characteristics with no rank between the categories.

The context of this project revolves around utilizing a diverse dataset comprising various patient attributes, such as demographic information (e.g., age), laboratory test results (e.g., blood pressure readings), and other relevant parameters that are indicative of kidney health. The explanation about the attributes occur in the dataset are described in the table below:

No.	Attribute Name	Data Type	Description
1	age	numerical	age of the patient
2	bp	numerical	blood pressure of the patient
3	sg	nominal	specific gravity, the level values of the concentration of particles in urine
4	al	nominal	albumin, the level values of the albuminuria in urine
5	su	nominal	sugar, the level values of glycosuria in urine
6	rbc	nominal	the patient has normal status of red blood cells in urine or not
7	pc	nominal	the patient has normal status of pus cell in urine or not
8	pcc	nominal	the patient has pus cell clumps in urine or not
9	ba	nominal	the patient has bacteria in urine or not

10	bgr	numerical	the concentration of blood glucose random in urine
11	bu	numerical	the concentration of blood urea in urine
12	sc	numerical	the concentration of creatinine in the blood serum
13	sod	numerical	the concentration of sodium ions in the blood serum
14	pot	numerical	the concentration of potassium ions in the blood serum
15	hemo	numerical	the concentration of hemoglobin in the blood serum
16	pcv	numerical	packed cell volume, the volume percentage of red blood cells in the blood
17	wc	numerical	white blood cell count
18	rc	numerical	red blood cell count
19	htn	nominal	the patient having hypertension or not
20	dm	nominal	the patient having diabetes mellitus or not
21	cad	nominal	the patient having coronary artery disease or not
22	appe	nominal	the patient having appetite or not
23	pe	nominal	the patient having pedal edema or not
24	ane	nominal	the patient having anemia or not
25	class	nominal	the patient having chronic kidney disease or not

2.0 SUMMARY OF PROJECT CONTEXT AND OBJECTIVES

2.1 SUMMARY OF THE PROJECT CONTEXT

The project, titled "Predict Chronic Kidney Disease of Patients," aims to leverage data mining techniques to develop a predictive model for identifying the likelihood of chronic kidney disease (CKD) in patients.

The project encompasses several key aspects. Firstly, the data will undergo several data mining process, which are the data preprocessing process: cleaning, transformation, and load, to ensure its quality and relevance for model development. Subsequently, the modelling part will be using five models which are Decision Tree, Random Forest, Support Vector Machine, Logistic Regression, and Naive Bayes. The evaluation phase will involve several testing and validation. We compare the models' results by using 4 aspects, which are the model's accuracy score, performance metrics in the classification report, the values of MSE and RMSE, and the ROC Curve with AUC value for each five models.

Moreover, the project intends to create a user-friendly interface Graphical User Interface, GUI for healthcare professionals, enabling them to input patient data easily and obtain real-time predictions regarding CKD. The GUI is made and named "Chronic Kidney Disease of Patient Prediction". The user may enter the data value required to execute the prediction. The output displayed on the GUI will be notckd or ckd. This deployment phase is crucial, as it aims to bridge the gap between data analysis and practical application in the healthcare domain.

In summary, the project's context revolves around utilizing advanced data mining techniques to develop a predictive model for chronic kidney disease diagnosis. By leveraging comprehensive patient data and employing robust methodologies, the project aims to contribute significantly to early detection and better management of CKD, thereby positively impacting patient care and healthcare systems.

2.2 OBJECTIVES

1. To develop the most accurate model for predicting the ckd disease.
2. To evaluate the developed models' accuracies and performance using patients' data to ensure reliability and effectiveness in predicting chronic kidney disease.
3. To discover the risk factors that are associated with the early onset of chronic kidney disease among the Indian population.

3.0 METHODOLOGY

3.1 DATA FRAMEWORK

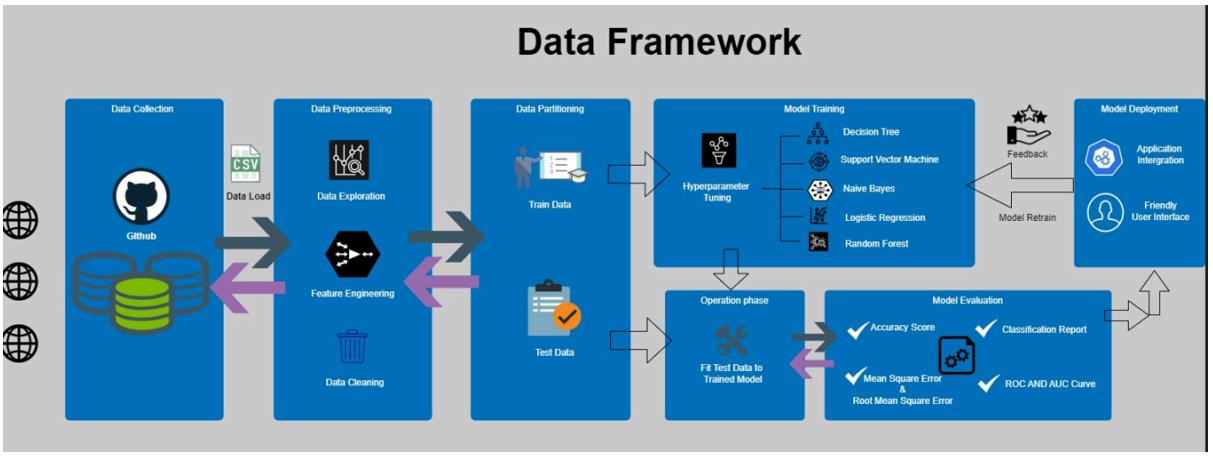


Figure 3.0 Data Framework

In the initial project phase, data was extracted from GitHub to form the basis for subsequent analysis. A crucial step in ensuring the reliability of the analysis involved meticulous data cleaning to address errors and inconsistencies. Following this, data transformation was conducted to make it compatible with machine learning algorithms. Feature engineering was then applied to extract relevant information, enhancing the dataset with meaningful variables for the prediction task. To accurately assess model performance, the data was divided into training and test sets which aligns more with ETL, enabling model training on the former and evaluation on the latter.

In the modeling phase, five distinct machine learning models were utilized: Decision Tree, Random Forest, Support Vector Machine (SVM), Logistic Regression, and Naive Bayes. Each model, trained on the prepared data, demonstrated its respective accuracy. Rigorous evaluation was conducted using diverse metrics to comprehensively understand their performance.

During the evaluation and deployment phase, accuracy, F1-score, and ROC AUC were employed as performance metrics. The model that demonstrated superior performance during the evaluation process emerged as the top-performing model., justifying its selection for deployment in the prediction task. The final step involved integrating the best performing model into a user-friendly interface or application, ensuring accessibility and usability for end-users. This successful deployment marks the completion of a data-driven journey, covering collection, preprocessing, modeling, and deployment, with the most effective algorithm identified as the optimal solution for the given prediction task.

3.2 PACKAGES REQUIRED

The packages used in this study divided into two parts: utilisation in modeling and utilisation in GUI development.

The packages utilised in modeling are:

1. **scikit-learn:**
   It used for machine learning tasks, including model training (DecisionTreeClassifier, RandomForestClassifier, SVC), model evaluation (accuracy_score, confusion_matrix, classification_report, etc.), and hyperparameter tuning (GridSearchCV).
2. **NumPy and Pandas:**
   It used for data manipulation, preprocessing, and organizing data into formats suitable for modeling.

The packages utilised in GUI development are:

1. **Tkinter, ttk, and ttkthemes:**
   It utilized for building the graphical user interface for the application.
2. **Matplotlib and Seaborn:**
   It used for embedding visualizations or plots within the GUI, providing a visual representation of data or model performance.

3.3 DATA PREPROCESSING

In the part data preprocessing, the goal is to clean the dataset and prepare it for further analysis and predictive modelling of a dataset named "ckd.csv" related to chronic kidney disease. Firstly, Figure 3.1 shows the Pandas library is used to read the dataset file "ckd.csv" into a Pandas DataFrame, which allows for easy data manipulation and analysis. The dataset contains various features related to kidney health and has missing values represented by question marks ("?").

```
In [1]: import numpy as np
        import pandas as pd

        # Read dataset file ckd.csv
        data = pd.read_csv("ckd.csv",header=0, na_values="?")
        data
```

Out[1]:

	Age	Bp	Sg	Al	Su	Rbc	Pc	Pcc	Ba	Bgr	...	Pcv	Wbcc	Rbcc	Htn	Dm	Cad	Appet	pe	Ane	Class
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44.0	7800.0	5.2	yes	yes	no	good	no	no	ckd
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38.0	6000.0	NaN	no	no	no	good	no	no	ckd
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31.0	7500.0	NaN	no	yes	no	poor	no	yes	ckd
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...	32.0	6700.0	3.9	yes	no	no	poor	yes	yes	ckd
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...	35.0	7300.0	4.6	no	no	no	good	no	no	ckd
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
395	55.0	80.0	1.020	0.0	0.0	normal	normal	notpresent	notpresent	140.0	...	47.0	6700.0	4.9	no	no	no	good	no	no	notckd
396	42.0	70.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	75.0	...	54.0	7800.0	6.2	no	no	no	good	no	no	notckd
397	12.0	80.0	1.020	0.0	0.0	normal	normal	notpresent	notpresent	100.0	...	49.0	6600.0	5.4	no	no	no	good	no	no	notckd
398	17.0	60.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	114.0	...	51.0	7200.0	5.9	no	no	no	good	no	no	notckd
399	58.0	80.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	131.0	...	53.0	6800.0	6.1	no	no	no	good	no	no	notckd

400 rows × 25 columns

Figure 3.1 Import related library and read csv file

The next step Figure 3.2 involves replacing these question marks ("?") with NumPy NaN values, which are recognized as missing or null values.

```
In [2]: # Replace null values "?" by numpy.NaN
        data.replace("?", np.NaN)
```

Out[2]:

	Age	Bp	Sg	Al	Su	Rbc	Pc	Pcc	Ba	Bgr	...	Pcv	Wbcc	Rbcc	Htn	Dm	Cad	Appet	pe	Ane	Class
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44.0	7800.0	5.2	yes	yes	no	good	no	no	ckd
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38.0	6000.0	NaN	no	no	no	good	no	no	ckd
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31.0	7500.0	NaN	no	yes	no	poor	no	yes	ckd
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...	32.0	6700.0	3.9	yes	no	no	poor	yes	yes	ckd
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...	35.0	7300.0	4.6	no	no	no	good	no	no	ckd
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
395	55.0	80.0	1.020	0.0	0.0	normal	normal	notpresent	notpresent	140.0	...	47.0	6700.0	4.9	no	no	no	good	no	no	notckd
396	42.0	70.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	75.0	...	54.0	7800.0	6.2	no	no	no	good	no	no	notckd
397	12.0	80.0	1.020	0.0	0.0	normal	normal	notpresent	notpresent	100.0	...	49.0	6600.0	5.4	no	no	no	good	no	no	notckd
398	17.0	60.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	114.0	...	51.0	7200.0	5.9	no	no	no	good	no	no	notckd
399	58.0	80.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	131.0	...	53.0	6800.0	6.1	no	no	no	good	no	no	notckd

400 rows × 25 columns

Figure 3.2 Replace question marks "?" with NaN

The dataset contains nominal categorical variables, such as "Rbc", "Pc", "Pcc", "Ba", "Htn", "Dm", "Cad", "Appet", "pe" and "Ane" which have multiple categories. The Figure 3.3 indicates to prepare the data for modelling, these nominal values are mapped to binary values using a predefined mapping dictionary. For instance, categorical values like "normal," "abnormal," "present," or "notpresent" are transformed into binary 1s and 0s.

```
In [3]: # Mapping of nominal values to binary values
        nominal_to_binary_mapping = {
            "Rbc": {"normal": 1, "abnormal": 0},
            "Pc": {"normal": 1, "abnormal": 0},
            "Pcc": {"present": 1, "notpresent": 0},
            "Ba": {"present": 1, "notpresent": 0},
            "Htn": {"yes": 1, "no": 0},
            "Dm": {"yes": 1, "no": 0},
            "Cad": {"yes": 1, "no": 0},
            "Appet": {"good": 1, "poor": 0},
            "pe": {"yes": 1, "no": 0},
            "Ane": {"yes": 1, "no": 0}
        }

        # Replace nominal values with binary values in the dataset
        data.replace(nominal_to_binary_mapping, inplace=True)
```

Figure 3.3 Mapping of nominal values to binary values

Once the nominal values are converted into binary representations, Figure 3.4 shows the missing values in the dataset are handled. The missing values are filled using the mean value of the respective columns. This imputation strategy helps to ensure that the dataset is complete and ready for analysis or modelling.

In [11]:
```
# Fill null values with mean value of the respective column

data.fillna(round(data.mean(),2), inplace=True)
data
```

Out[11]:

	Age	Bp	Sg	Al	Su	Rbc	Pc	Pcc	Ba	Bgr	...	Pcv	Wbcc	Rbcc	Htn	Dm	Cad	Appet	pe	Ane	Class
0	48.0	80.0	1.020	1.0	0.0	0.81	1.0	0.0	0.0	121.00	...	44.0	7800.0	5.20	1.0	1	0.0	1.0	0.0	0.0	ckd
1	7.0	50.0	1.020	4.0	0.0	0.81	1.0	0.0	0.0	148.04	...	38.0	6000.0	4.71	0.0	0	0.0	1.0	0.0	0.0	ckd
2	62.0	80.0	1.010	2.0	3.0	1.00	1.0	0.0	0.0	423.00	...	31.0	7500.0	4.71	0.0	1	0.0	0.0	0.0	1.0	ckd
3	48.0	70.0	1.005	4.0	0.0	1.00	0.0	1.0	0.0	117.00	...	32.0	6700.0	3.90	1.0	0	0.0	0.0	1.0	1.0	ckd
4	51.0	80.0	1.010	2.0	0.0	1.00	1.0	0.0	0.0	106.00	...	35.0	7300.0	4.60	0.0	0	0.0	1.0	0.0	0.0	ckd
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
395	55.0	80.0	1.020	0.0	0.0	1.00	1.0	0.0	0.0	140.00	...	47.0	6700.0	4.90	0.0	0	0.0	1.0	0.0	0.0	notckd
396	42.0	70.0	1.025	0.0	0.0	1.00	1.0	0.0	0.0	75.00	...	54.0	7800.0	6.20	0.0	0	0.0	1.0	0.0	0.0	notckd
397	12.0	80.0	1.020	0.0	0.0	1.00	1.0	0.0	0.0	100.00	...	49.0	6600.0	5.40	0.0	0	0.0	1.0	0.0	0.0	notckd
398	17.0	60.0	1.025	0.0	0.0	1.00	1.0	0.0	0.0	114.00	...	51.0	7200.0	5.90	0.0	0	0.0	1.0	0.0	0.0	notckd
399	58.0	80.0	1.025	0.0	0.0	1.00	1.0	0.0	0.0	131.00	...	53.0	6800.0	6.10	0.0	0	0.0	1.0	0.0	0.0	notckd

400 rows × 25 columns

Figure 3.4 Fill null values with mean values

Finally, Figure 3.5 shows the cleaned dataset is saved into a new CSV file named "cleaned_data.csv". This file contains the preprocessed data, with missing values handled, nominal values converted to binary representations, and the dataset ready for further predictive modelling and analysis.

In [10]:
```
# Save this dataset as cleaned_data.csv for further prediction
data.to_csv("cleaned_data.csv", sep=',', index=False)
```

Figure 3.5 Save the cleaned dataset for further prediction

3.4 DATA EXPLORATION

Data exploration begins by reading the cleaned dataset file "cleaned_data.csv" into a Pandas DataFrame named ‘balance_data’. Then, Figure 3.6 prints out the length and shape of the dataset using len(balance_data) and balance_data.shape respectively to understand the number of rows and columns in the dataset. Additionally, a preview of the dataset using balance_data.head() to display the first few rows and balance_data.describe() to provide statistical information about each numerical column like mean, standard deviation, minimum, maximum, and quartile values.

```
In [2]: # Data Exploration
        balance_data = pd.read_csv('cleaned_data.csv', sep=',', header=0)
        print("Dataset Length: ", len(balance_data))
        print("Dataset Shape: ", balance_data.shape)
        print("Data Exploration:")
        print(balance_data.head())
        print(balance_data.describe())
```

Figure 3.6 Data exploration

The dataset contains 400 entries (rows) and 25 columns. Figure 3.7 shows the first five rows of the dataset including variable age, blood pressure, specific gravity, albumin, sugar levels, presence of red blood cells, pus cells, various diseases like hypertension, diabetes mellitus, coronary artery disease and more. The statistical summary reveals the central tendency and dispersion of numerical columns. For instance, the mean age is approximately 51.48, the mean blood pressure is around 76.47, and so on. These insights provide an understanding of the data distribution and potential ranges of values for different features.

```
Dataset Length:  400
Dataset Shape:  (400, 25)
Data Exploration:
    Age  blood_pressure  specific_gravity  albumin  sugar  red_blood_cells  \
0  48.0            80.0             1.020      1.0    0.0             0.81
1   7.0            50.0             1.020      4.0    0.0             0.81
2  62.0            80.0             1.010      2.0    3.0             1.00
3  48.0            70.0             1.005      4.0    0.0             1.00
4  51.0            80.0             1.010      2.0    0.0             1.00

   pus_cell  pus_cell_clumps  bacteria  blood_glucose_random  ...  \
0       1.0              0.0       0.0                121.00  ...
1       1.0              0.0       0.0                148.04  ...
2       1.0              0.0       0.0                423.00  ...
3       0.0              1.0       0.0                117.00  ...
4       1.0              0.0       0.0                106.00  ...

   packed_cell_volume  white_blood_cell_count  red_blood_cell_count  \
0                44.0                  7800.0                  5.20
1                38.0                  6000.0                  4.71
2                31.0                  7500.0                  4.71
3                32.0                  6700.0                  3.90
4                35.0                  7300.0                  4.60

   hypertension  diabetes_mellitus  coronary_artery_disease  appetite  \
0           1.0                  1                      0.0       1.0
1           0.0                  0                      0.0       1.0
2           0.0                  1                      0.0       0.0
3           1.0                  0                      0.0       0.0
4           0.0                  0                      0.0       1.0

   pedal_edema  anemia  class
0          0.0     0.0    ckd
1          0.0     0.0    ckd
2          0.0     1.0    ckd
3          1.0     1.0    ckd
4          0.0     0.0    ckd

[5 rows x 25 columns]
                Age  blood_pressure  specific_gravity      albumin        sugar  \
count  400.000000      400.000000        400.000000  400.000000  400.000000
mean    51.483300       76.469100          1.017712    1.017300    0.450125
std     16.974966       13.476298          0.005434    1.272318    1.029487
min      2.000000       50.000000          1.005000    0.000000    0.000000
25%     42.000000       70.000000          1.015000    0.000000    0.000000
50%     54.000000       78.235000          1.020000    1.000000    0.000000
75%     64.000000       80.000000          1.020000    2.000000    0.450000
max     90.000000      180.000000          1.025000    5.000000    5.000000

       red_blood_cells    pus_cell  pus_cell_clumps    bacteria  \
count       400.000000  400.000000       400.000000  400.000000
mean          0.810300    0.772625         0.106100    0.055600
std           0.308983    0.383751         0.306756    0.228199
min           0.000000    0.000000         0.000000    0.000000
25%           0.810000    0.770000         0.000000    0.000000
50%           1.000000    1.000000         0.000000    0.000000
75%           1.000000    1.000000         0.000000    0.000000
max           1.000000    1.000000         1.000000    1.000000
```

```
       blood_glucose_random  ...  hemoglobin  packed_cell_volume  \
count            400.000000  ...  400.000000          400.000000
mean             148.036900  ...   12.526900           38.883700
std               74.782634  ...    2.716171            8.151082
min               22.000000  ...    3.100000            9.000000
25%              101.000000  ...   10.875000           34.000000
50%              126.000000  ...   12.530000           38.880000
75%              150.000000  ...   14.625000           44.000000
max              490.000000  ...   17.800000           54.000000

       white_blood_cell_count  red_blood_cell_count  hypertension  \
count              400.000000            400.000000    400.000000
mean              8406.121800              4.708275      0.369350
std               2523.219976              0.840315      0.482023
min               2200.000000              2.100000      0.000000
25%               6975.000000              4.500000      0.000000
50%               8406.120000              4.710000      0.000000
75%               9400.000000              5.100000      1.000000
max              26400.000000              8.000000      1.000000

       diabetes_mellitus  coronary_artery_disease    appetite  pedal_edema  \
count          400.00000               400.000000  400.000000   400.000000
mean             0.34250                 0.085450    0.794475     0.190475
std              0.47514                 0.279166    0.404077     0.392677
min              0.00000                 0.000000    0.000000     0.000000
25%              0.00000                 0.000000    1.000000     0.000000
50%              0.00000                 0.000000    1.000000     0.000000
75%              1.00000                 0.000000    1.000000     0.000000
max              1.00000                 1.000000    1.000000     1.000000

          anemia
count  400.000000
mean     0.150375
std      0.357440
min      0.000000
25%      0.000000
50%      0.000000
75%      0.000000
max      1.000000

[8 rows x 24 columns]
```

Figure 3.7 Data exploration output

Figure 3.8 proceeds to split the dataset into input features (X) and the target variable (Y). It extracts all columns except the last one as input features (X) and considers the last column ('class') as the target variable (Y). This step prepares the data for model building.

```
In [3]: # Building Phase
        # Separating the target variable
        X = balance_data.values[:, 0:24]
        Y = balance_data.values[:, -1]
```

Figure 3.8 Building Phase

Figure 3.9 using train_test_split from Scikit-learn to further divide the dataset into training and testing sets. It allocates 70% of the data for training (X_train and y_train) and 30% for testing (X_test and y_test). The test_size=0.3 parameter specifies the proportion of the dataset to include in the test split, while random_state=100 ensures reproducibility by fixing the random seed.

```
In [4]: # Splitting the dataset into train and test
        X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=100)
```

Figure 3.9 Splitting dataset into train and test

3.5 MODELLING

The predictive modelling used to train and test the dataset are Decision Tree, Random Forest, Support Vector Machine (SVM), Logistic Regression, and Naive Bayes. This comprehensive analysis includes hyperparameter tuning, model training, evaluation metrics calculation, visualisation of performance, feature importance analysis, and cross-validation to ensure robustness and generalizability of the model. Each step contributes to understanding and optimising the model for the given dataset.

3.5.1 DECISION TREE

Figure 3.10 shows a grid of hyperparameters (param_grid) for the decision tree model. Initialise a decision tree classifier (clf) and use GridSearchCV to find the best hyperparameters based on accuracy (scoring='accuracy') using a 5-fold cross-validation (cv=5). The output indicates the best hyperparameters for 'criterion': 'gini', 'max_depth': 3, 'min_samples_leaf': 7.

```
In [5]: # Hyperparameter Tuning for decision tree
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 5, 7],
    'min_samples_leaf': [3, 5, 7]
}

clf = DecisionTreeClassifier(random_state=100)
grid_search = GridSearchCV(estimator=clf, param_grid=param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
print("Best Hyperparameters:", grid_search.best_params_)
tuned_clf = grid_search.best_estimator_

Best Hyperparameters: {'criterion': 'gini', 'max_depth': 3, 'min_samples_leaf': 7}
```

Figure 3.10 Hyperparameter tuning for decision tree

Figure 3.11 creates a decision tree classifier (clf_gini) with the best hyperparameters obtained from the grid search. Fit the classifier using the training data (X_train, y_train).

```
In [6]: # Training phase
clf_gini = DecisionTreeClassifier(criterion="gini", random_state=100, max_depth=3, min_samples_leaf=7)
clf_gini.fit(X_train, y_train)

Out[6]: DecisionTreeClassifier(max_depth=3, min_samples_leaf=7, random_state=100)
```

Figure 3.11 Create a decision tree classifier

Figure 3.12 predicts the target variable for the test data using the decision tree classifier trained with the Gini index. Then, transform the target variables (y_test,

y_pred_gini) to binary values for further metric calculations. In the results, the confusion matrix shows the counts of true positive (77), true negative (39), false positive (3), and false negative (1) predictions. Accuracy score computes the accuracy of the classifier is 96.6667. Classification report provides precision, recall, F1-score, and support for each class. F1 score the harmonic mean of precision and recall is 0.9747. The AUC score area under the ROC curve, which measures a classifier's ability to distinguish between classes, was 0.9688.

```
In [7]: # Operational Phase
        print("\nResults Using Gini Index:")
        y_pred_gini = clf_gini.predict(X_test)
        y_test_binary = np.where(y_test == 'ckd', 1, 0)
        y_pred_gini_binary = np.where(y_pred_gini == 'ckd', 1, 0)
        # Metrics for Decision Tree
        print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_gini))
        print("Accuracy:", accuracy_score(y_test,y_pred_gini) * 100)
        print("Report:\n", classification_report(y_test, y_pred_gini))
        print("F1 Score:", f1_score(y_test_binary, y_pred_gini_binary))
        print("AUC Score:", roc_auc_score(y_test_binary, y_pred_gini_binary))

Results Using Gini Index:
Confusion Matrix:
 [[77  3]
 [ 1 39]]
Accuracy: 96.66666666666667
Report:
              precision    recall  f1-score   support

         ckd       0.99      0.96      0.97        80
      notckd       0.93      0.97      0.95        40

    accuracy                           0.97       120
   macro avg       0.96      0.97      0.96       120
weighted avg       0.97      0.97      0.97       120

F1 Score: 0.9746835443037976
AUC Score: 0.96875
```

Figure 3.12 Operational Phase for decision tree

Figure 3.13 plots the Receiver Operating Characteristic (ROC) curve for the decision tree classifier to visualise its performance. The x-axis shows the False Positive Rate (FPR), and the y-axis shows the True Positive Rate (TPR). The orange line represents the ROC curve of the model, and has an area under the curve (AUC) of 0.97, indicating excellent performance. The orange line also lies above the random classifier dotted blue line that is considered good.

```
In [8]: # ROC Curve for Decision Tree
fpr_dt, tpr_dt, _ = roc_curve(y_test_binary,y_pred_gini_binary)
roc_auc_dt = auc(fpr_dt, tpr_dt)

plt.figure(figsize=(8, 8))
plt.plot(fpr_dt, tpr_dt, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc_dt))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Decision Tree)')
plt.legend(loc="lower right")
plt.show()
```

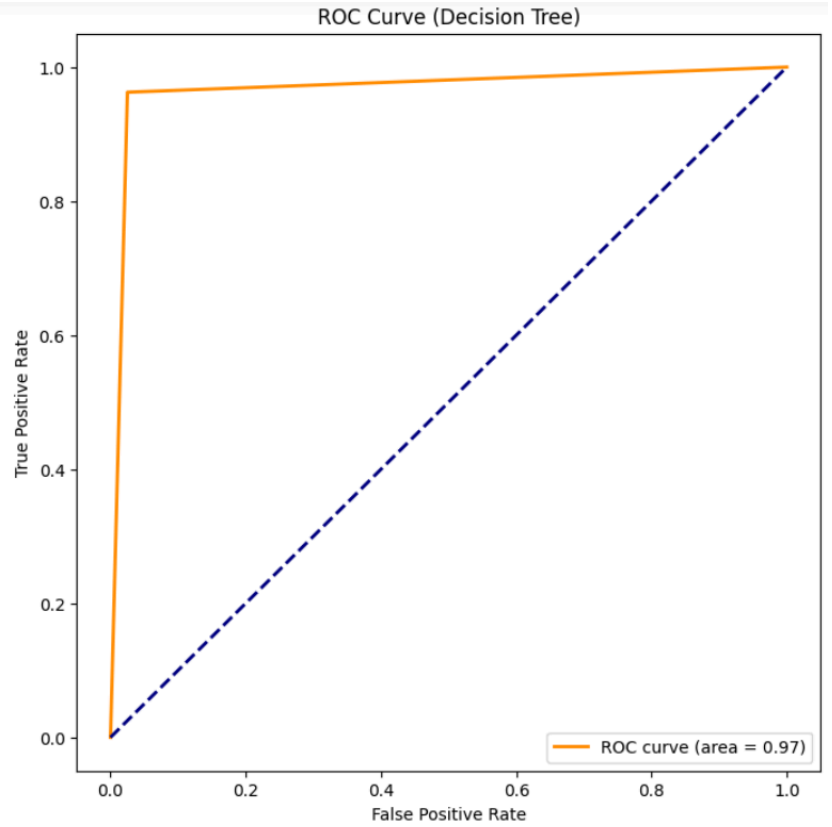


Figure 3.13 ROC curve

Figure 3.14 will calculate the MSE and RMSE for the decision tree classifier predictions. MSE is calculated by taking the average of the squares of the errors between predicted and actual values, which in this case is approximately 0.0333. RMSE is simply the square root of MSE, giving a more interpretable value of error in the units of the dependent variable, which here is approximately 0.1826.

```
In [9]: # Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Decision Tree
mse_dt = mean_squared_error(y_test_binary,y_pred_gini_binary)
rmse_dt = np.sqrt(mse_dt)
print("Mean Squared Error (MSE):", mse_dt)
print("Root Mean Squared Error (RMSE):", rmse_dt)

Mean Squared Error (MSE): 0.03333333333333333
Root Mean Squared Error (RMSE): 0.18257418583505536
```

Figure 3.14 Calculate MSE and RMSE

Figure 3.15 indicates the decision tree in the graphical representation used for decision-making. The decision tree diagram shows nodes containing conditions and outcomes. The top node contains a condition based on a feature from the dataset. Branching out from the root are two paths leading to other nodes with additional conditions. Orange nodes represent one class label ("0"), while blue nodes represent another class label ("1"). Each node displays information including the condition for splitting and the number of samples that fall into each category.

```
In [10]: # Visualizing Decision Trees using plot_tree
         plt.figure(figsize=(36, 24))
         plot_tree(clf_gini, filled=True, feature_names=balance_data.columns[:-1], class_names=["0", "1"], rounded=True, fontsize=10)
         plt.show()
```

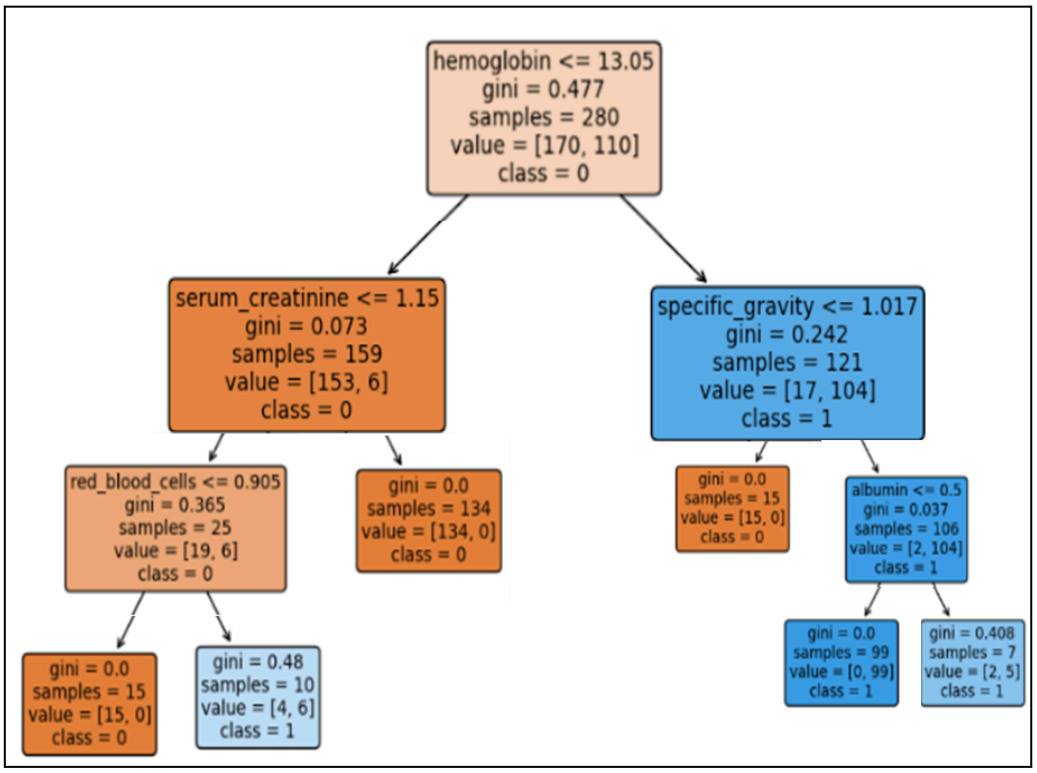


Figure 3.15 Decision tree diagram

Figure 3.16 computes and plots the feature importance for the decision tree classifier using bar plot, highlighting the most significant features in predicting the target. Hemoglobin has been identified as the most important feature, followed by specific gravity, red blood cells, serum creatinine, and albumin.

```
In [11]: # Feature Importance for Decision Tree
         feature_importance_dt = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Importance': clf_gini.feature_importances_})
         feature_importance_dt = feature_importance_dt.sort_values(by='Importance', ascending=False)

         plt.figure(figsize=(10, 6))
         sns.barplot(x='Importance', y='Feature', data=feature_importance_dt)
         plt.title('Feature Importance (Decision Tree)')
         plt.show()
```

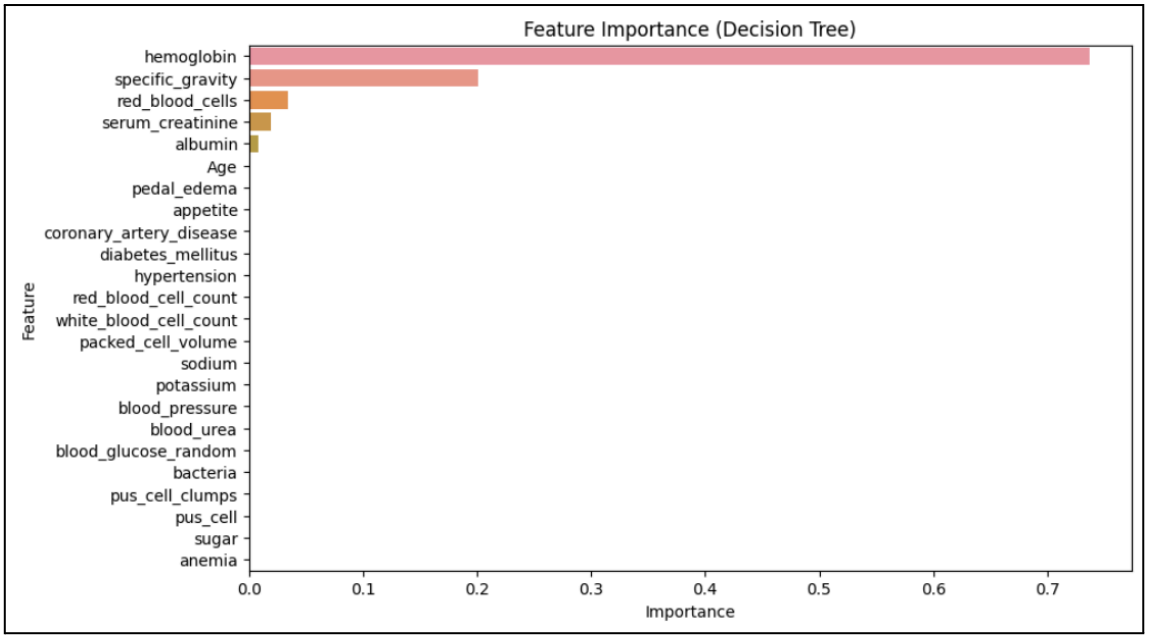


Figure 3.16 Future importance (Decision Tree)

Cross-validation is a technique used to evaluate the performance of a machine learning model. Figure 3.17 the cross-validation results show five different accuracy scores which are 0.95, 1, 0.9375, 0.975 and 0.9875. These values indicate how well the Decision Tree model performed on each fold of the dataset during cross-validation. The mean accuracy is calculated by taking the average of all these individual accuracy scores from each fold. Here, it is approximately 0.9701. This mean accuracy gives a general idea of how well the model is expected to perform on unseen data.

```
In [12]: # Cross-Validation for Decision Tree
         cv_results_dt = cross_val_score(clf_gini, X, Y, cv=5, scoring='accuracy')
         print("Cross-Validation Results (Decision Tree):", cv_results_dt)
         print("Mean Accuracy (Decision Tree):", np.mean(cv_results_dt))

         Cross-Validation Results (Decision Tree): [0.95   1.     0.9375 0.975  0.9875]
         Mean Accuracy (Decision Tree): 0.9700000000000001
```

Figure 3.17 Calculate cross validation and mean accuracy

3.5.2 RANDOM FOREST

Figure 3.18 defines a grid of hyperparameters for a Random Forest classifier as number of estimators, maximum depth, minimum samples per leaf, and maximum features. Use GridSearchCV to perform an exhaustive search for the best combination of hyperparameters based on accuracy, using 5-fold cross-validation.

```
In [13]: # Hyperparameter Tuning for Random Forest
param_grid_rf = {
    'n_estimators': [50, 100, 150],
    'max_depth': [3, 5, 7],
    'min_samples_leaf': [3, 5, 7],
    'max_features': ['sqrt', 'log2']
}

clf_rf = RandomForestClassifier(random_state=100)
grid_search_rf = GridSearchCV(estimator=clf_rf, param_grid=param_grid_rf, cv=5, scoring='accuracy')
grid_search_rf.fit(X_train, y_train)
print("Best Hyperparameters (Random Forest):", grid_search_rf.best_params_)
tuned_clf_rf = grid_search_rf.best_estimator_

Best Hyperparameters (Random Forest): {'max_depth': 3, 'max_features': 'sqrt', 'min_samples_leaf': 3, 'n_estimators': 50}
```

Figure 3.18 Hyperparameter tuning for random forest

Figure 3.19 initialises a Random Forest classifier using the best hyperparameters obtained from the grid search. Fit the classifier to the training data (X_train, y_train).

```
In [14]: # Training phase for Random Forest
clf_rf = RandomForestClassifier(n_estimators=50, max_depth=3, min_samples_leaf=3, max_features='sqrt', random_state=100)
clf_rf.fit(X_train, y_train)

Out[14]: RandomForestClassifier(max_depth=3, max_features='sqrt', min_samples_leaf=3,
                       n_estimators=50, random_state=100)
```

Figure 3.19 Create random forest classifier

Figure 3.20 predicts the target variable for the test data using the random forest classifier. Then, convert class labels to binary values for evaluation. In the results, the confusion matrix shows the counts of true positive (80), true negative (39), false positive (0), and false negative (1) predictions. Accuracy score computes the accuracy of the classifier is 96.1667. Classification report provides precision, recall, F1-score, and support for each class. F1 score the harmonic mean of precision and recall is 0.9938. The AUC score area under the ROC curve, which measures a classifier's ability to distinguish between classes, was 0.9875.

```
In [15]: # Operational Phase for Random Forest
print("\nResults Using Random Forest:")
y_pred_rf = clf_rf.predict(X_test)
y_test_binary = np.where(y_test == 'ckd', 1, 0)
y_pred_rf_binary = np.where(y_pred_rf == 'ckd', 1, 0)
# Metrics for Random Forest
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
print("Accuracy:# ROC Curve for Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_test_binary, y_pred_rf_binary)
roc_auc_rf = auc(fpr_rf, tpr_rf)

plt.figure(figsize=(8, 8))
plt.plot(fpr_rf, tpr_rf, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc_rf))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Random Forest)')
plt.legend(loc="lower right")
plt.show()", accuracy_score(y_test, y_pred_rf) * 100)
print("Report:\n", classification_report(y_test, y_pred_rf))
print("F1 Score:", f1_score(y_test_binary, y_pred_rf_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_rf_binary))
```

```
Results Using Random Forest:
Confusion Matrix:
 [[80  0]
 [ 1 39]]
Accuracy: 99.16666666666667
Report:
              precision    recall  f1-score   support

         ckd       0.99      1.00      0.99        80
      notckd       1.00      0.97      0.99        40

    accuracy                           0.99       120
   macro avg       0.99      0.99      0.99       120
weighted avg       0.99      0.99      0.99       120

F1 Score: 0.9937888198757764
AUC Score: 0.9874999999999999
```

Figure 3.20 Operational Phase for random forest

Figure 3.21 plots the Receiver Operating Characteristic (ROC) curve for the random forest classifier to visualise its performance. The x-axis shows the False Positive Rate (FPR), and the y-axis shows the True Positive Rate (TPR). The orange line represents the ROC curve of the model, and has an area under the curve (AUC) of 0.99, indicating excellent performance. The orange line also lies above the random classifier dotted blue line that is considered good.

```
In [16]: # ROC Curve for Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_test_binary, y_pred_rf_binary)
roc_auc_rf = auc(fpr_rf, tpr_rf)

plt.figure(figsize=(8, 8))
plt.plot(fpr_rf, tpr_rf, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc_rf))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Random Forest)')
plt.legend(loc="lower right")
plt.show()
```

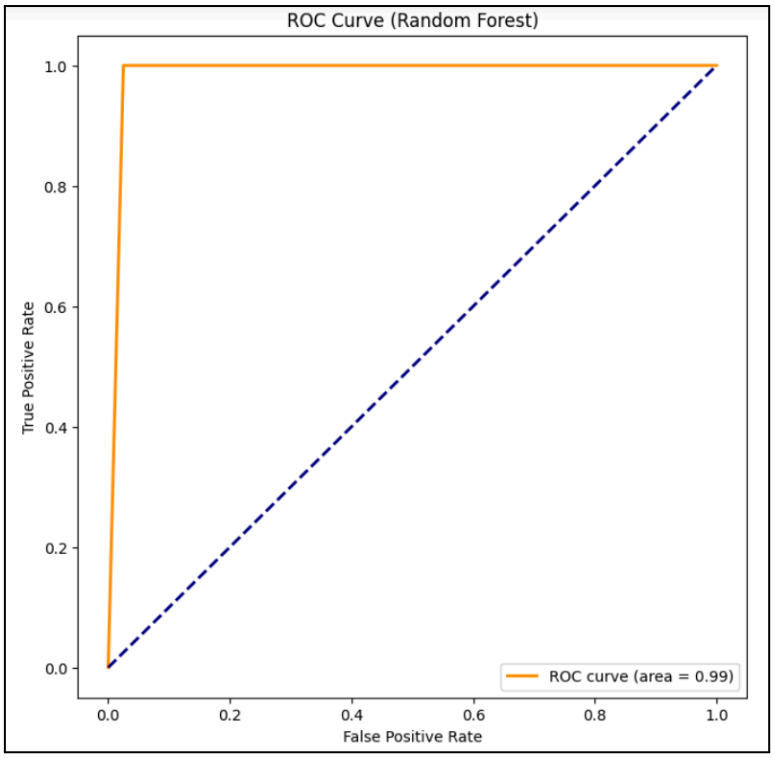


Figure 3.21 ROC curve

Figure 3.22 computes and plots the feature importance for the decision tree classifier using bar plot, highlighting the most significant features in predicting the target. Packed cell volume has been identified as the most important feature, followed by hemoglobin, red blood cell count, serum creatinine, blood glucose random, albumin, specific gravity, red blood cells, hypertension, sodium, diabetes mellitus, blood urea, appetite, sugar, blood pressure, pus cell, white blood cell count anemia, potassium and age.

```
In [17]: # Feature Importance for Random Forest
         feature_importance_rf = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Importance': clf_rf.feature_importances_})
         feature_importance_rf = feature_importance_rf.sort_values(by='Importance', ascending=False)

         plt.figure(figsize=(10, 6))
         sns.barplot(x='Importance', y='Feature', data=feature_importance_rf)
         plt.title('Feature Importance (Random Forest)')
         plt.show()
```

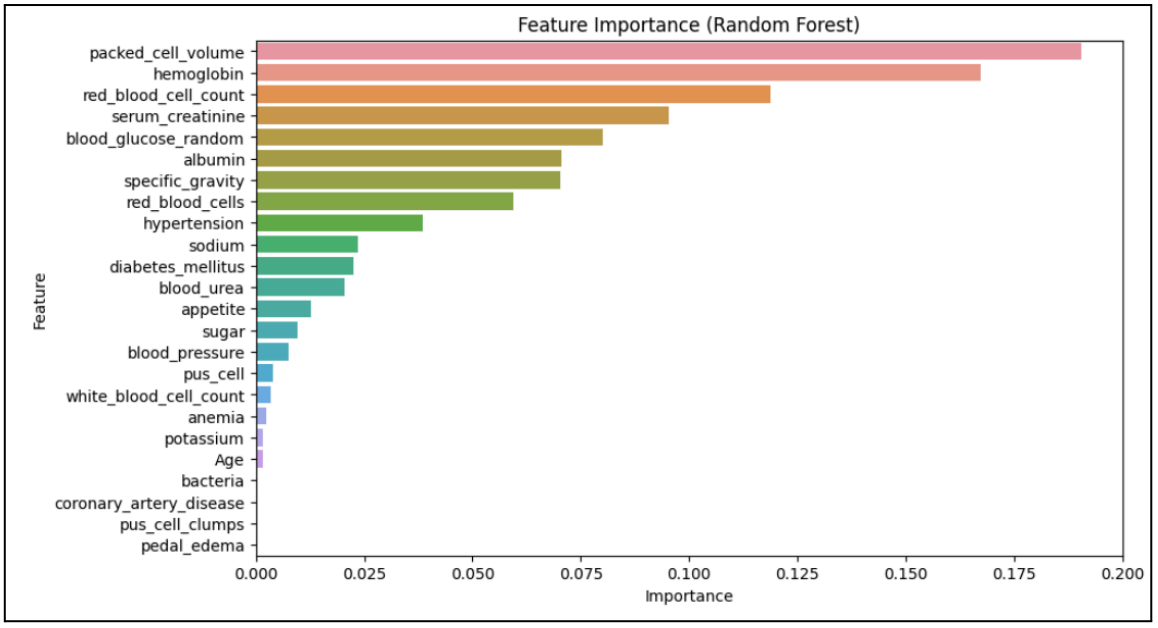


Figure 3.22 Future importance (Random Forest)

Figure 3.23 will calculate the MSE and RMSE for the random forest classifier predictions. MSE is calculated by taking the average of the squares of the errors between predicted and actual values, which in this case is approximately 0.0083. RMSE is simply the square root of MSE, giving a more interpretable value of error in the units of the dependent variable, which here is approximately 0.0913.

```
In [18]: # Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Random Forest
mse_rf = mean_squared_error(y_test_binary, y_pred_rf_binary)
rmse_rf = np.sqrt(mse_rf)
print("Mean Squared Error (MSE):", mse_rf)
print("Root Mean Squared Error (RMSE):", rmse_rf)

Mean Squared Error (MSE): 0.008333333333333333
Root Mean Squared Error (RMSE): 0.09128709291752768
```

Figure 3.23 Calculate MSE and RMSE

Cross-validation is a technique used to evaluate the performance of a machine learning model. Figure 3.24 the cross-validation results show five different accuracy scores which are 0.9875, 1, 0.9625, 1 and 1. These values indicate how well the Random Forest model performed on each fold of the dataset during cross-validation. The mean accuracy is calculated by taking the average of all these individual accuracy scores from each fold. Here, it is approximately 0.99. This mean accuracy gives a general idea of how well the model is expected to perform on unseen data.

```
In [19]: # Cross-Validation for Random Forest
         cv_results_rf = cross_val_score(clf_rf, X, Y, cv=5, scoring='accuracy')
         print("Cross-Validation Results (Random Forest):", cv_results_rf)
         print("Mean Accuracy (Random Forest):", np.mean(cv_results_rf))

         Cross-Validation Results (Random Forest): [0.9875 1.     0.9625 1.     1.    ]
         Mean Accuracy (Random Forest): 0.99
```

Figure 3.24 Calculate cross validation and mean accuracy

3.5.3 SUPPORT VECTOR MACHINE

Figure 3.25 initialises the SVM classifier (SVC) with the chosen kernel 'rbf' for a radial basis function kernel. Fit the classifier using the training data (X_train, y_train).

```
In [40]: # Training phase for Support Vector Machine (SVM)
         clf_svm = SVC(kernel='rbf', random_state=100)
         clf_svm.fit(X_train, y_train)

Out[40]: SVC
         SVC(random_state=100)
```

Figure 3.25 Create support vector machine classifier

Figure 3.26 predicts the target variable for the test data using the support vector machine classifier. Then, convert class labels to binary values for evaluation. In the results, the confusion matrix shows the counts of true positive (80), true negative (0), false positive (0), and false negative (40) predictions. Accuracy score computes the accuracy of the classifier is 66.6666. Classification report provides precision, recall, F1-score, and support for each class. F1 score the harmonic mean of precision and recall is 0.8. The AUC score area under the ROC curve, which measures a classifier's ability to distinguish between classes, was 0.5.

```
In [41]: # Operational Phase for SVM
         print("\nResults Using SVM:")
         y_pred_svm = clf_svm.predict(X_test)
         y_test_binary = np.where(y_test == 'ckd', 1, 0)
         y_pred_svm_binary = np.where(y_pred_svm == 'ckd', 1, 0)
         # Metrics for SVM
         print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_svm))
         print("Accuracy:", accuracy_score(y_test, y_pred_svm) * 100)
         print("Report:\n", classification_report(y_test, y_pred_svm, zero_division=1))
         print("F1 Score:", f1_score(y_test_binary, y_pred_svm_binary))
         print("AUC Score:", roc_auc_score(y_test_binary, y_pred_svm_binary))
```

```
Results Using SVM:
Confusion Matrix:
 [[80  0]
 [40  0]]
Accuracy: 66.66666666666666
Report:
              precision    recall  f1-score   support

         ckd       0.67      1.00      0.80        80
      notckd       1.00      0.00      0.00        40

    accuracy                           0.67       120
   macro avg       0.83      0.50      0.40       120
weighted avg       0.78      0.67      0.53       120

F1 Score: 0.8
AUC Score: 0.5
```

Figure 3.26 Operational Phase for support vector machine

Figure 3.27 plots the Receiver Operating Characteristic (ROC) curve for the support vector machine classifier to visualise its performance. The x-axis shows the False Positive Rate (FPR), and the y-axis shows the True Positive Rate (TPR). The curve is a diagonal line from the bottom left to the top right, indicating an area under the curve (AUC) of 0.50. In other words, this model does not have discriminative power to distinguish between the positive and negative classes.

```
In [35]: # ROC Curve for SVM
         fpr_svm, tpr_svm, _ = roc_curve(y_test_binary, y_pred_svm_binary)
         roc_auc_svm = auc(fpr_svm, tpr_svm)

         plt.figure(figsize=(8, 8))
         plt.plot(fpr_svm, tpr_svm, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc_svm))
         plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
         plt.xlabel('False Positive Rate')
         plt.ylabel('True Positive Rate')
         plt.title('ROC Curve (SVM)')
         plt.legend(loc="lower right")
         plt.show()
```

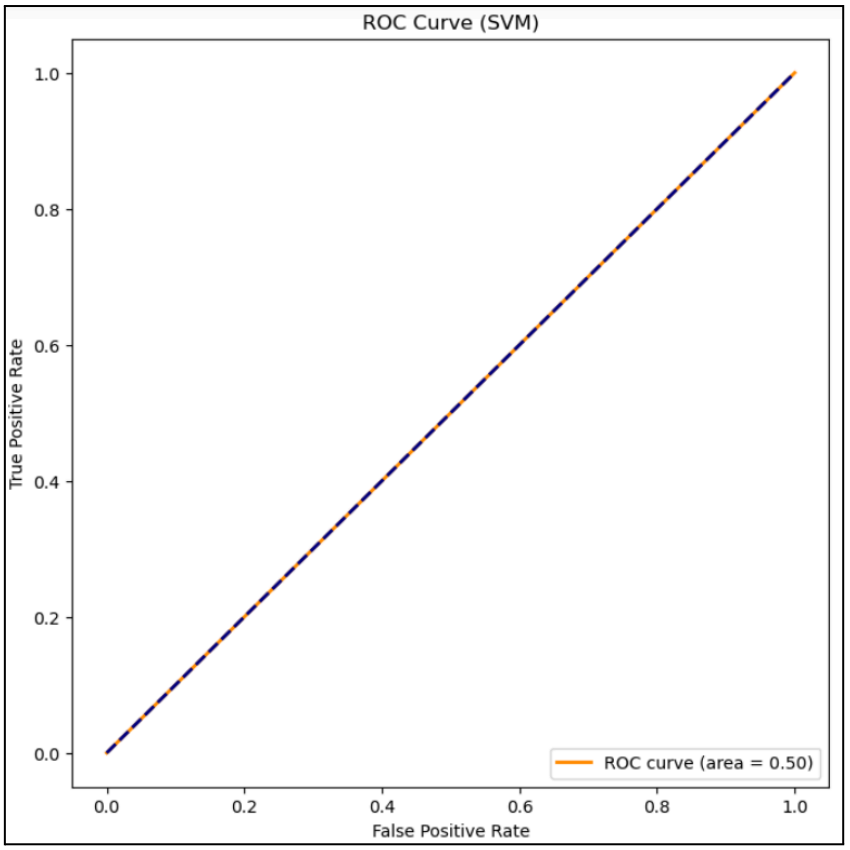


Figure 3.27 ROC curve

Figure 3.28 will calculate the MSE and RMSE for the support vector machine classifier predictions. MSE is calculated by taking the average of the squares of the errors between predicted and actual values, which in this case is approximately 0.3333. RMSE is simply the square root of MSE, giving a more interpretable value of error in the units of the dependent variable, which here is approximately 0.5774.

```
In [36]: # Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for SVM
         mse_svm = mean_squared_error(y_test_binary, y_pred_svm_binary)
         rmse_svm = np.sqrt(mse_svm)
         print("Mean Squared Error (MSE) for SVM:", mse_svm)
         print("Root Mean Squared Error (RMSE) for SVM:", rmse_svm)

         Mean Squared Error (MSE) for SVM: 0.3333333333333333
         Root Mean Squared Error (RMSE) for SVM: 0.5773502691896257
```

Figure 3.28 Calculate MSE and RMSE

Cross-validation is a technique used to evaluate the performance of a machine learning model. Figure 3.29 the cross-validation results show five different accuracy scores which are 0.625, 0.625, 0.625, 0.625 and 0.625. These values indicate how well the Random

Forest model performed on each fold of the dataset during cross-validation. The mean accuracy is calculated by taking the average of all these individual accuracy scores from each fold. Here, it is approximately 0.625. This mean accuracy gives a general idea of how well the model is expected to perform on unseen data.

```
In [37]: # Cross-Validation for SVM
         cv_results_svm = cross_val_score(clf_svm, X, Y, cv=5, scoring='accuracy')
         print("Cross-Validation Results (SVM):", cv_results_svm)
         print("Mean Accuracy (SVM):", np.mean(cv_results_svm))

         Cross-Validation Results (SVM): [0.625 0.625 0.625 0.625 0.625]
         Mean Accuracy (SVM): 0.625
```

Figure 3.29 Calculate cross validation and mean accuracy

3.5.4 LOGISTIC REGRESSION

Figure 3.30 defines a logistic regression model and uses GridSearchCV to find the best hyperparameters. The hyperparameters that are tuning the penalty term and the inverse of the regularisation strength. Then print the best hyperparameters and use them to create a new logistic regression model.

```
In [25]: # Hyperparameter Tuning for Logistic Regression
         param_grid_lr = {
             'penalty': ['l1', 'l2'],
             'C': [0.001, 0.01, 0.1, 1, 10, 100]
         }

         clf_lr = LogisticRegression(random_state=100, solver='liblinear')
         grid_search_lr = GridSearchCV(estimator=clf_lr, param_grid=param_grid_lr, cv=5, scoring='accuracy')
         grid_search_lr.fit(X_train, y_train)
         print("Best Hyperparameters (Logistic Regression):", grid_search_lr.best_params_)
         tuned_clf_lr = grid_search_lr.best_estimator_

         Best Hyperparameters (Logistic Regression): {'C': 1, 'penalty': 'l2'}
```

Figure 3.30 Hyperparameter tuning for logistic regression

Figure 3.31 initialises a logistic regression using the best hyperparameters obtained from the grid search. Fit the classifier to the training data (X_train, y_train).

```
In [26]: # Training phase for Logistic Regression
         clf_lr = LogisticRegression(C=1, penalty='l2', max_iter=100, random_state=100)
         clf_lr.fit(X_train, y_train)

Out[26]: LogisticRegression(C=1, random_state=100)
```

Figure 3.31 Create logistic regression

Figure 3.32 predicts the target variable for the test data using the logistic regression. Then, convert class labels to binary values for evaluation. In the results, the confusion matrix shows the counts of true positive (74), true negative (38), false positive (6), and false negative (2) predictions. Accuracy score computes the accuracy of the classifier is 93.3333. Classification report provides precision, recall, F1-score, and support for each class. F1 score the harmonic mean of precision and recall is 0.9487. The AUC score area under the ROC curve, which measures a classifier's ability to distinguish between classes, was 0.9375.

```
In [27]: # Operational Phase for Logistic Regression
         print("\nResults Using Logistic Regression:")
         y_pred_lr = clf_lr.predict(X_test)
         y_pred_lr_binary = np.where(y_pred_lr == 'ckd', 1, 0)
         # Metrics for Logistic Regression
         print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_lr))
         print("Accuracy:", accuracy_score(y_test, y_pred_lr) * 100)
         print("Report:\n", classification_report(y_test, y_pred_lr))
         print("F1 Score:", f1_score(y_test_binary, y_pred_lr_binary))
         print("AUC Score:", roc_auc_score(y_test_binary, y_pred_lr_binary))
```

```
Results Using Logistic Regression:
Confusion Matrix:
 [[74  6]
 [ 2 38]]
Accuracy: 93.33333333333333
Report:
              precision    recall  f1-score   support

         ckd       0.97      0.93      0.95        80
      notckd       0.86      0.95      0.90        40

    accuracy                           0.93       120
   macro avg       0.92      0.94      0.93       120
weighted avg       0.94      0.93      0.93       120

F1 Score: 0.9487179487179489
AUC Score: 0.9374999999999999
```

Figure 3.32 Operational Phase for logistic regression

Figure 3.33 plots the Receiver Operating Characteristic (ROC) curve for the logistic regression to visualise its performance. The x-axis shows the False Positive Rate (FPR), and the y-axis shows the True Positive Rate (TPR). The orange line represents the ROC curve of the model, and has an area under the curve (AUC) of 0.94, indicating excellent performance. The orange line also lies above the random classifier dotted blue line that is considered good.

```
In [28]: # ROC Curve for Logistic Regression
fpr_lr, tpr_lr, _ = roc_curve(y_test_binary, y_pred_lr_binary)
roc_auc_lr = auc(fpr_lr, tpr_lr)

plt.figure(figsize=(8, 8))
plt.plot(fpr_lr, tpr_lr, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc_lr))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Logistic Regression)')
plt.legend(loc="lower right")
plt.show()
```

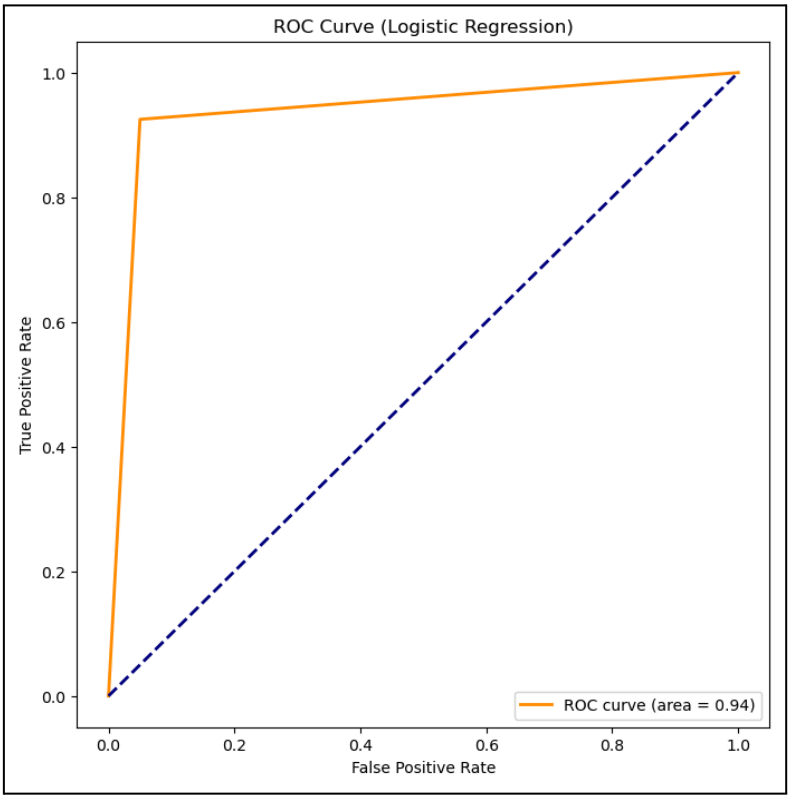


Figure 3.33 ROC curve

Figure 3.34 computes and plots the feature importance for the logistic regression using bar plot, highlighting the most significant features in predicting the target. The most important features in the model are Pcv, Hemo, Rbcc, Rbc, Appet, Age and Pc. These features have the largest coefficients and are therefore the most influential in predicting the target variable. The other features have smaller coefficients and are less influential in predicting the target variable.

```
In [29]: # Feature Coefficients for Logistic Regression
feature_coefficients_lr = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Coefficient': clf_lr.coef_[0]})
feature_coefficients_lr = feature_coefficients_lr.sort_values(by='Coefficient', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Coefficient', y='Feature', data=feature_coefficients_lr)
plt.title('Feature Coefficients (Logistic Regression)')
plt.show()
```

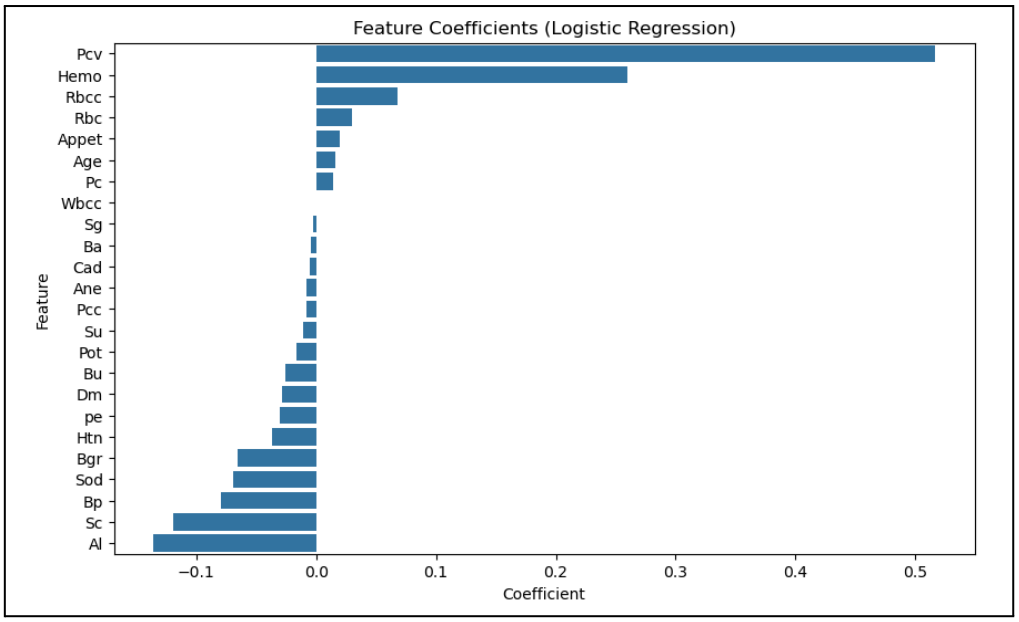


Figure 3.34 Future importance (Logistic Regression)

Figure 3.35 will calculate the MSE and RMSE for the logistic regression predictions. MSE is calculated by taking the average of the squares of the errors between predicted and actual values, which in this case is approximately 0.0667. RMSE is simply the square root of MSE, giving a more interpretable value of error in the units of the dependent variable, which here is approximately 0.2582.

```
In [30]: # Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Logistic Regression
         mse_lr = mean_squared_error(y_test_binary, y_pred_lr_binary)
         rmse_lr = np.sqrt(mse_lr)
         print("Mean Squared Error (MSE):", mse_lr)
         print("Root Mean Squared Error (RMSE):", rmse_lr)

         Mean Squared Error (MSE): 0.06666666666666667
         Root Mean Squared Error (RMSE): 0.2581988897471611
```

Figure 3.35 Calculate MSE and RMSE

3.5.5 NAIVE BAYES

Figure 3.36 initialises a naive bayes using the best hyperparameters obtained from the grid search. Fit the classifier to the training data (X_train, y_train).

```
In [31]: # Naive Bayes - Gaussian Naive Bayes does not have many hyperparameters to tune
         clf_nb = GaussianNB()
         clf_nb.fit(X_train, y_train)

Out[31]: GaussianNB()
```

Figure 3.36 Create naive bayes

Figure 3.37 predicts the target variable for the test data using the logistic regression. Then, convert class labels to binary values for evaluation. In the results, the confusion matrix shows the counts of true positive (78), true negative (40), false positive (2), and false negative (0) predictions. Accuracy score computes the accuracy of the classifier is 98.3333. Classification report provides precision, recall, F1-score, and support for each class. F1 score the harmonic mean of precision and recall is 0.9873. The AUC score area under the ROC curve, which measures a classifier's ability to distinguish between classes, was 0.9875.

```
In [32]: # Operational Phase for Naive Bayes
         print("\nResults Using Naive Bayes:")
         y_pred_nb = clf_nb.predict(X_test)
         y_pred_nb_binary = np.where(y_pred_nb == 'ckd', 1, 0)
```

```
In [33]: # Metrics for Naive Bayes
         print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_nb))
         print("Accuracy:", accuracy_score(y_test, y_pred_nb) * 100)
         print("Report:\n", classification_report(y_test, y_pred_nb))
         print("F1 Score:", f1_score(y_test_binary, y_pred_nb_binary))
         print("AUC Score:", roc_auc_score(y_test_binary, y_pred_nb_binary))
```

```
Confusion Matrix:
 [[78  2]
 [ 0 40]]
Accuracy: 98.33333333333333
Report:
              precision    recall  f1-score   support

         ckd       1.00      0.97      0.99        80
      notckd       0.95      1.00      0.98        40

    accuracy                           0.98       120
   macro avg       0.98      0.99      0.98       120
weighted avg       0.98      0.98      0.98       120

F1 Score: 0.9873417721518987
AUC Score: 0.9875
```

Figure 3.37 Operational Phase for naive bayes

Figure 3.38 plots the Receiver Operating Characteristic (ROC) curve for the naibe bayes to visualise its performance. The x-axis shows the False Positive Rate (FPR), and the y-axis shows the True Positive Rate (TPR). The orange line represents the ROC curve of the model, and has an area under the curve (AUC) of 0.99, indicating excellent performance. The orange line also lies above the random classifier dotted blue line that is considered good.

```
In [34]: # ROC Curve for Naive Bayes
fpr_nb, tpr_nb, _ = roc_curve(y_test_binary, y_pred_nb_binary)
roc_auc_nb = auc(fpr_nb, tpr_nb)

plt.figure(figsize=(8, 8))
plt.plot(fpr_nb, tpr_nb, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc_nb))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Naive Bayes)')
plt.legend(loc="lower right")
plt.show()
```

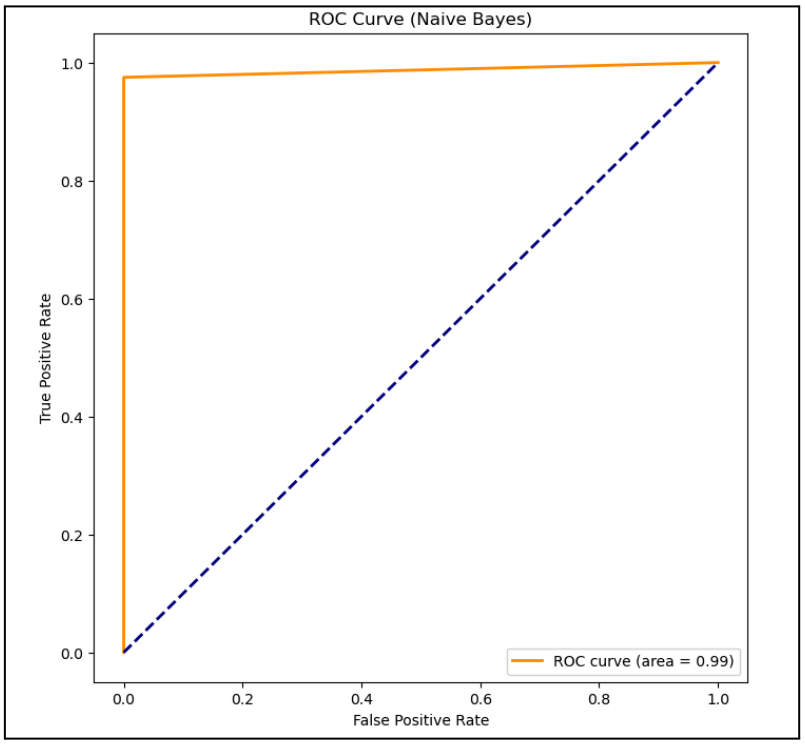


Figure 3.38 ROC curve

Figure 3.39 will calculate the MSE and RMSE for the naive bayes predictions. MSE is calculated by taking the average of the squares of the errors between predicted and actual values, which in this case is approximately 0.0166. RMSE is simply the square root of MSE, giving a more interpretable value of error in the units of the dependent variable, which here is approximately 0.1291.

```
In [35]: # Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Naive Bayes
mse_nb = mean_squared_error(y_test_binary, y_pred_nb_binary)
rmse_nb = np.sqrt(mse_nb)
print("Mean Squared Error (MSE):", mse_nb)
print("Root Mean Squared Error (RMSE):", rmse_nb)

Mean Squared Error (MSE): 0.016666666666666666
Root Mean Squared Error (RMSE): 0.12909944487358055
```

Figure 3.39 Calculate MSE and RMSE

3.6 MODEL COMPARISON

Figure 3.30 illustrates the comparative accuracy scores of five distinct machine learning models: Support Vector Machine (SVM), Decision Tree, Random Forest, Logistic Regression, and Naive Bayes. The Random Forest model emerges as the top performer, achieving an accuracy score of approximately 99.17%. Following closely is the Decision Tree model with an accuracy of about 96.67%. Naive Bayes demonstrates commendable performance, securing a notable accuracy score with 98.33%. Logistic Regression, despite its simplicity, proves competitive in this scenario for about 96.67% of accuracy score. However, the Support Vector Machine lags behind, registering the lowest accuracy at around 66.67%. In this particular task, the Random Forest model exhibits superior accuracy, making it a compelling choice for accurate predictions. Nonetheless, the selection of the most suitable model may hinge on factors such as interpretability, computational efficiency, and dataset characteristics.

```
results = pd.DataFrame({
    'Model': ['Support Vector Machine', 'Decision Tree', 'Random Forest', 'Logistic Regression', 'Naive Bayes'],
    'Accuracy Score': [
        accuracy_score(y_test, y_pred_svm) * 100,
        accuracy_score(y_test, y_pred_gini) * 100,
        accuracy_score(y_test, y_pred_rf) * 100,
        accuracy_score(y_test, y_pred_lr) * 100,  # Logistic Regression
        accuracy_score(y_test, y_pred_nb) * 100   # Naive Bayes
    ]
})

result_df = results.sort_values(by='Accuracy Score', ascending=False)
result_df = result_df.set_index('Accuracy Score')
result_df
```

Accuracy Score	Model
99.166667	Random Forest
98.333333	Naive Bayes
96.666667	Decision Tree
93.333333	Logistic Regression
66.666667	Support Vector Machine

Figure 3.30 Compare accuracy

Figure 3.31 depicts the performance metrics of five distinct machine learning models Random Forest, Decision Tree, Support Vector Machine (SVM), Logistic Regression, and Naive Bayes. Notably, Random Forest continues to demonstrate superior performance, achieving the highest precision, recall, and f1-score of 0.99 across all metrics. This implies a

high degree of accuracy and reliability in its predictions. The Decision Tree model closely follows with a precision, recall, and f1-score of 0.97, indicating a commendable level of accuracy, albeit slightly lower than Random Forest. Introducing Logistic Regression and Naive Bayes into the comparison, both models exhibit strong performance with precision, recall, and f1-score values above 0.93. Logistic Regression achieves a precision of 0.94, a recall of 0.93, and an f1-score of 0.93, showcasing reliable predictive capabilities. Naive Bayes performs similarly well, attaining a precision, recall, and f1-score of 0.98. In contrast, the Support Vector Machine model continues to exhibit lower scores, with a precision of 0.78, recall of 0.67, and f1-score of 0.53. These metrics suggest that SVM may not be as accurate or reliable in its predictions compared to the other models. In conclusion, considering the expanded set of models and their performance metrics, Random Forest and Naive Bayes emerge as strong performers, with Random Forest maintaining its position as the top-performing model among the five.

```python
from sklearn.metrics import classification_report

# Create a DataFrame for Classification Report
classification_reports = []

# SVM
report_svm = classification_report(y_test, y_pred_svm, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'SVM', **report_svm['weighted avg']})

# Decision Tree
report_dt = classification_report(y_test, y_pred_gini, output_dict=True)
classification_reports.append({'Model': 'Decision Tree', **report_dt['weighted avg']})

# Random Forest
report_rf = classification_report(y_test, y_pred_rf, output_dict=True)
classification_reports.append({'Model': 'Random Forest', **report_rf['weighted avg']})

# Logistic Regression
report_lr = classification_report(y_test, y_pred_lr, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'Logistic Regression', **report_lr['weighted avg']})

# Naive Bayes
report_nb = classification_report(y_test, y_pred_nb, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'Naive Bayes', **report_nb['weighted avg']})

# Create DataFrame
classification_df = pd.DataFrame(classification_reports)
classification_df = classification_df.set_index('Model')

# Sort DataFrame by Score in descending order
classification_df = classification_df.sort_values(by='f1-score', ascending=False).round(2)

# Print the DataFrame
print(classification_df)
```

```
                    precision  recall  f1-score  support
Model
Random Forest            0.99    0.99      0.99      120
Naive Bayes              0.98    0.98      0.98      120
Decision Tree            0.97    0.97      0.97      120
Logistic Regression      0.94    0.93      0.93      120
SVM                      0.78    0.67      0.53      120
```

Figure 3.31 Compare performance metrics

Figure 3.32 illustrates the Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for five distinct machine learning models including Decision Tree, Random Forest, Support Vector Machine (SVM), Logistic Regression, and Naive Bayes. Examining the results, the Support Vector Machine model exhibits the highest errors with an MSE of 0.33 and an RMSE of 0.58. These metrics suggest significant deviations between the SVM model's predictions and the actual values. The Decision Tree model demonstrates lower errors, yielding an MSE of 0.03 and an RMSE of 0.18, indicating closer proximity between its predictions and the actual values compared to the SVM model. Notably, the Random Forest model outperforms the others with the lowest errors of MSE of 0.01 and RMSE of 0.09 that implying the closest alignment between its predictions and the actual values. Introducing Logistic Regression and Naive Bayes into the analysis, both models exhibit competitive error metrics, with Logistic Regression achieving an MSE of 0.07 and an RMSE of 0.26, while Naive Bayes attains an MSE of 0.02 and an RMSE of 0.13. In conclusion, considering these error metrics, the Random Forest model emerges as the top performer among the five, demonstrating the closest predictions to the actual values.

```python
from sklearn.metrics import mean_squared_error
from math import sqrt

# Create a DataFrame for MSE and RMSE
mse_rmse_results = []

# SVM
mse_svm = mean_squared_error(y_test_binary, y_pred_svm_binary)
rmse_svm = sqrt(mse_svm)
mse_rmse_results.append({'Model': 'SVM', 'MSE': mse_svm, 'RMSE': rmse_svm})

# Decision Tree
mse_dt = mean_squared_error(y_test_binary, y_pred_gini_binary)
rmse_dt = sqrt(mse_dt)
mse_rmse_results.append({'Model': 'Decision Tree', 'MSE': mse_dt, 'RMSE': rmse_dt})

# Random Forest
mse_rf = mean_squared_error(y_test_binary, y_pred_rf_binary)
rmse_rf = sqrt(mse_rf)
mse_rmse_results.append({'Model': 'Random Forest', 'MSE': mse_rf, 'RMSE': rmse_rf})

# Logistic Regression
mse_lr = mean_squared_error(y_test_binary, y_pred_lr_binary)
rmse_lr = sqrt(mse_lr)
mse_rmse_results.append({'Model': 'Logistic Regression', 'MSE': mse_lr, 'RMSE': rmse_lr})

# Naive Bayes
mse_nb = mean_squared_error(y_test_binary, y_pred_nb_binary)
rmse_nb = sqrt(mse_nb)
mse_rmse_results.append({'Model': 'Naive Bayes', 'MSE': mse_nb, 'RMSE': rmse_nb})

# Create DataFrame
mse_rmse_df = pd.DataFrame(mse_rmse_results)
mse_rmse_df = mse_rmse_df.set_index('Model')

# Sort DataFrame by MSE in ascending order
mse_rmse_df = mse_rmse_df.sort_values(by='MSE', ascending=False).round(2)

# Print the DataFrame
print(mse_rmse_df)
```

```
                     MSE  RMSE
Model
SVM                 0.33  0.58
Logistic Regression 0.07  0.26
Decision Tree       0.03  0.18
Naive Bayes         0.02  0.13
Random Forest       0.01  0.09
```

Figure 3.32 Compare MSE and RMSE

Figure 3.33 presents the Receiver Operating Characteristic (ROC) curve comparison and Area Under the Curve (AUC) values for five distinct machine learning models, Decision Tree, Random Forest, Support Vector Machine (SVM), Logistic Regression, and Naive Bayes. Examining the results, the Support Vector Machine model shows an AUC of 0.50, equivalent to that of a random predictor. This suggests that the SVM model's performance is no better than random guessing, indicating limitations in its predictive capabilities. In contrast, the Decision Tree model exhibits a strong AUC of 0.97, reflecting a high level of accuracy and reliability in its predictions. Remarkably, the Random Forest model outperforms others with the highest AUC of 0.99, signifying excellent performance and the most accurate predictions among the three models. Introducing Logistic Regression and Naive Bayes into the analysis, both models also showcase competitive AUC values, Logistic Regression at 0.94 and Naive Bayes at 0.99. In conclusion, based on these AUC values, the

Random Forest and Naive Bayes models emerge as the top performers among the five, demonstrating superior accuracy in their predictions.

AUC Curve

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Create a DataFrame for AUC
auc_results = []

# SVM
fpr_svm, tpr_svm, _ = roc_curve(y_test_binary, y_pred_svm_binary)
roc_auc_svm = auc(fpr_svm, tpr_svm)
auc_results.append({'Model': 'SVM', 'AUC': roc_auc_svm})

# Decision Tree
fpr_dt, tpr_dt, _ = roc_curve(y_test_binary, y_pred_gini_binary)
roc_auc_dt = auc(fpr_dt, tpr_dt)
auc_results.append({'Model': 'Decision Tree', 'AUC': roc_auc_dt})

# Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_test_binary, y_pred_rf_binary)
roc_auc_rf = auc(fpr_rf, tpr_rf)
auc_results.append({'Model': 'Random Forest', 'AUC': roc_auc_rf})

# Logistic Regression
fpr_lr, tpr_lr, _ = roc_curve(y_test_binary, y_pred_lr_binary)
roc_auc_lr = auc(fpr_lr, tpr_lr)
auc_results.append({'Model': 'Logistic Regression', 'AUC': roc_auc_lr})

# Naive Bayes
fpr_nb, tpr_nb, _ = roc_curve(y_test_binary, y_pred_nb_binary)
roc_auc_nb = auc(fpr_nb, tpr_nb)
auc_results.append({'Model': 'Naive Bayes', 'AUC': roc_auc_nb})

# Create DataFrame
auc_df = pd.DataFrame(auc_results)
auc_df = auc_df.set_index('Model').round(2)

# Sort DataFrame by AUC in descending order
auc_df = auc_df.sort_values(by='AUC', ascending=False)

# Print the DataFrame
print(auc_df)

# Plot ROC curves
plt.figure(figsize=(10, 8))
plt.plot(fpr_svm, tpr_svm, label=f'SVM (AUC = {roc_auc_svm:.2f})')
plt.plot(fpr_dt, tpr_dt, label=f'Decision Tree (AUC = {roc_auc_dt:.2f})')
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {roc_auc_rf:.2f})')
plt.plot(fpr_lr, tpr_lr, label=f'Logistic Regression (AUC = {roc_auc_lr:.2f})')
plt.plot(fpr_nb, tpr_nb, label=f'Naive Bayes (AUC = {roc_auc_nb:.2f})')

plt.plot([0, 1], [0, 1], linestyle='--', color='grey', label='Random')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend(loc='lower right')
plt.show()
```

```
                     AUC
Model
Random Forest       0.99
Naive Bayes         0.99
Decision Tree       0.97
Logistic Regression 0.94
SVM                 0.50
```

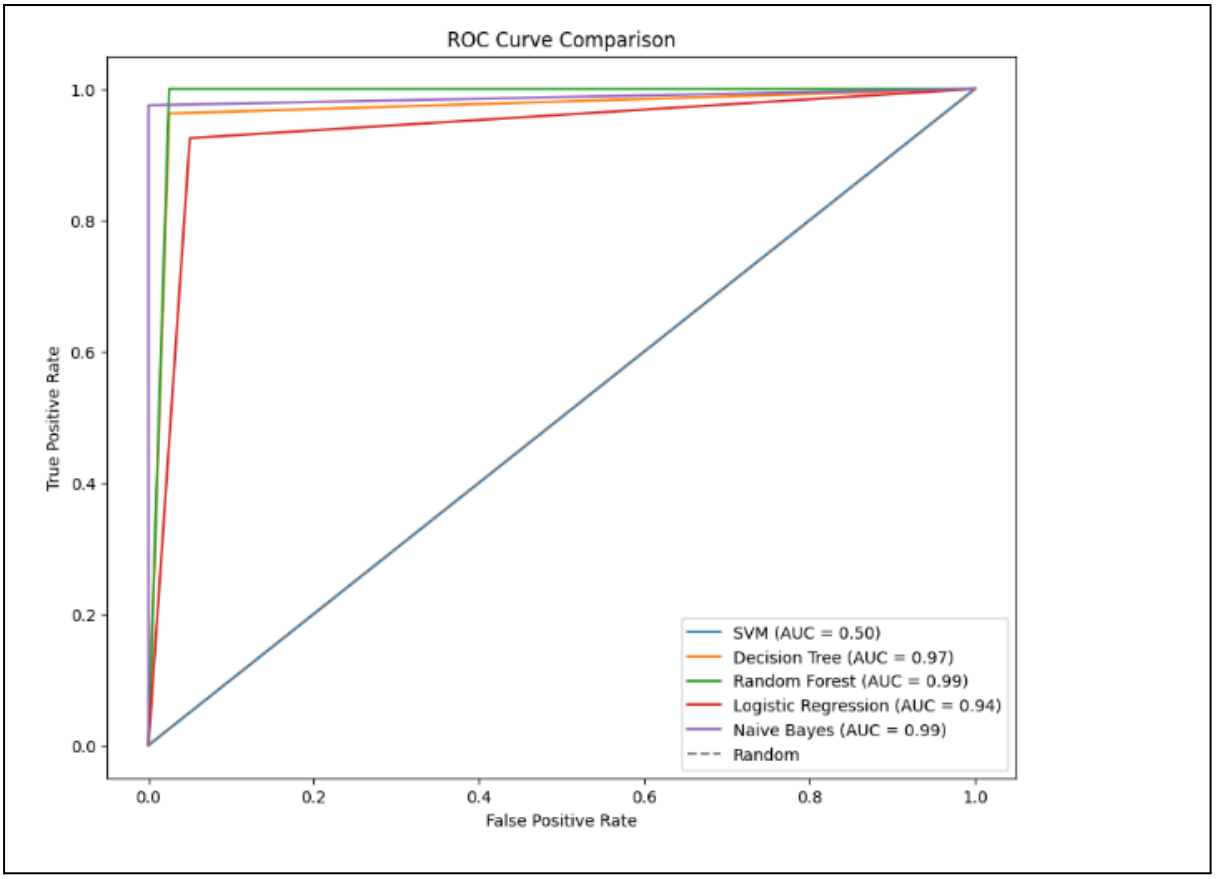


Figure 3.33 Compare ROC curve

3.7 USER GUI

The user GUI for CKD Prediction comprises of several parts. We utilise the 'ThemedStyle' to configurate the themed stype. The components that exist in the GUI are title label, a guideline label that providing the instruction to the user. Besides, the entry widgets for the user to input the data, a predict button, a clear input and output button, and a label for displaying the prediction result. We also utilise some styling configurations to make a better appearance of the GUI. The function of 'get_user_input' used to extract user input from every entry widgets while the lambda function of 'predict_lambda' associated with the Predict button. The functions of 'clear_input' and 'clear_output' used at the clear button. Furthermore, the part of event handling for the function of 'on_entry_return', it handle the function for each entry. Ultimately, the execution of the Tkinter event loop which is root.mainloop().

USER GUI

In [*]:
```
# Create the main window
root = tk.Tk()

# Create the main window with a themed style
style = ThemedStyle(root)
style.set_theme("blue")

# Set the title and icon
root.title("CKD Prediction")
root.iconbitmap("Erix-Subyarko-Medical-Medicine-Medicine-Tool-Doctor-Hospital-Stethoscope.512.ico")

# Set the size of the window (width x height)
root.geometry("1250x500")  # Change the size as needed

# Load the background image
background_image = tk.PhotoImage(file="23-Blood-test_1400x700-Copy.png")  # Replace with the path to your image
background_label = tk.Label(root, image=background_image)
background_label.place(relwidth=1, relheight=1)  # Make the label cover the entire window

# Create and place the title label
title_label = ttk.Label(root, text="Chronic Kidney Disease of Patients Prediction", font=('Arial', 24, 'bold'),
                        style="TLabel")
title_label.grid(row=0, column=0, columnspan=8, padx=10, pady=20)

# Create and place the guideline label with increased font size
guideline_label = ttk.Label(root, text="Please fill in the patient's health information in the provided fields. "
                                       "Then, click on the 'Predict' button below to generate and show the result.",
                            font=('Arial', 14), style="TLabel")
guideline_label.grid(row=1, column=0, columnspan=8, padx=10, pady=20)



# Create and place entry widgets for each feature in a 6x4 grid
entries = []
feature_labels = [
    "Age", "Blood Pressure mm/Hg", "Specific Gravity (1.005-1.025)", "Albumin (0-5)", "Sugar (0-5)",
    "Red Blood Cells (normal-1/abnormal-0)", "Pus Cell (normal-1/abnormal-0)", "Pus Cell Clumps (present-1/not present-0)", "Bac
    "Blood Glucose Random mgs/dl", "Blood Urea mgs/dl", "Serum Creatinine mgs/dl", "Sodium mEq/L",
    "Potassium mEq/L", "Hemoglobin gms", "Packed Cell Volume", "White Blood Cell Count cells/cumm",
    "Red Blood Cell Count cells/cumm", "Hypertension (yes-1/no-0)", "Diabetes Mellitus (yes-1/no-0)",
    "Coronary Artery Disease (yes-1/no-0)", "Appetite (good-1/poor-0)", "Pedal Edema (yes-1/no-0)", "Anemia (yes-1/no-0)"
]

row, col = 2, 0 # Start from row 2 (below the guideline label)
for i, label_text in enumerate(feature_labels):
    label = ttk.Label(root, text=label_text, style="TLabel")
    label.grid(row=row, column=col, padx=10, pady=5)
    entry = ttk.Entry(root, style="TEntry",width=10)
    entry.grid(row=row, column=col + 1, padx=10, pady=5)
    entries.append(entry)

    col += 2  # Move to the next column
    if col > 6:
        col = 0
        row += 1  # Move to the next row

# Create and place the predict button with a themed style
predict_button = ttk.Button(root, text="Predict", command=lambda: predict(entries, clf_gini, clf_rf, clf_svm, result_label),
                            style='TButton', takefocus=False)
predict_button.grid(row=row + 2, column=0, columnspan=8, pady=10)

# Create and place the result label
result_label = ttk.Label(root, text="Prediction for User Input:", style="PLabel.TLabel")
result_label.grid(row=row + 3, column=0, columnspan=8, pady=10)

# Add styling for labels and entries
style = ThemedStyle(root)
style.configure("TLabel", background=style.lookup("TFrame", "background"))
style.configure("TEntry", background=style.lookup("TFrame", "background"))
style.configure("PLabel.TLabel", font=('Arial', 12, 'bold'))
```

```
# Function to get user input for prediction
def get_user_input(entries):
    user_input = [float(entry.get()) for entry in entries]
    user_input_array = np.array(user_input).reshape(1, -1)
    return user_input_array


# Lambda function to predict and update the result label
predict_lambda = lambda: (
    user_input := get_user_input(entries),
    result_label.config(text=f"The model predicts: {clf_rf.predict(user_input)[0]}")
)

# Create and place the predict button without using the function
predict_button = ttk.Button(root, text="Predict", command=predict_lambda, style='TButton', takefocus=False)
predict_button.grid(row=row + 2, column=0, columnspan=8, pady=10)

def predict(entries, clf_gini, clf_rf, clf_svm, result_label):
    pass

def get_user_input(entries):
    user_input = [float(entry.get()) for entry in entries]
    user_input_array = np.array(user_input).reshape(1, -1)
    return user_input_array

def clear_input(entries):
    for entry in entries:
        entry.delete(0, tk.END)

def clear_output(result_label):
    result_label.config(text="Prediction for User Input:")

def on_entry_return(event):
    current_entry = event.widget
    current_row, current_col = current_entry.grid_info()["row"], current_entry.grid_info()["column"]

    # Calculate the next entry's row and column
    next_row = current_row
    next_col = current_col + 2
    if next_col > 6:
        next_col = 0
        next_row += 1

    # Move the focus to the next entry
    entries[next_row][next_col].focus_set()

def clear_input_and_output(entries, result_label):
    clear_input(entries)
    clear_output(result_label)

# Create and place the clear button
clear_button = ttk.Button(root, text="Clear", command=lambda: clear_input_and_output(entries, result_label),
                          style='TButton', takefocus=False)
clear_button.grid(row=row + 4, column=0, columnspan=8, pady=10)

# Run the Tkinter event loop
root.mainloop()
```

Figure 4.1 USER GUI Coding

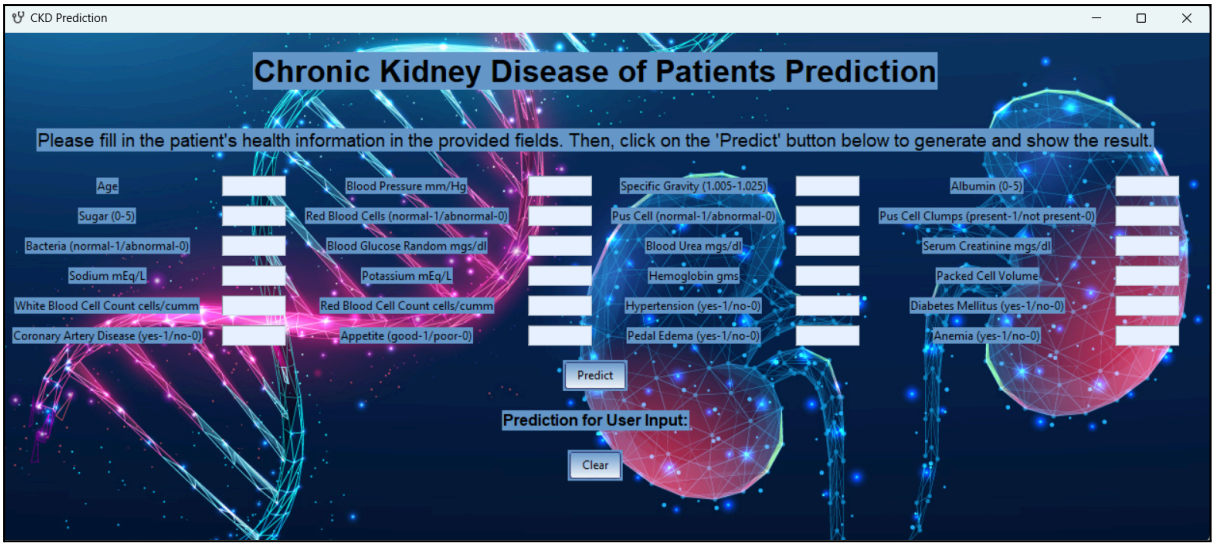


Figure 4.2 USER GUI

This graphical user interface (GUI) built with Tkinter and ttkthemes in Python serves as an interface for predicting Chronic Kidney Disease (CKD) based on user-input medical data. The system takes 23 features as input and predicts the probability of a patient having CKD. The main window features an aesthetically pleasing blue-themed design with a background image. Users are prompted to enter information such as age, blood pressure, and various medical parameters through labelled entry fields. The "Predict" button triggers the highest accuracy of machine learning random forest model (clf_rf) to make a prediction based on the entered data, displaying the result in a labelled output section. The GUI employs a clean layout, thematic styling, and interactive functionality, providing an intuitive platform for users to input medical data and receive CKD predictions.

4.0 RESULT AND DISCUSSION

This project is done by using Jupyter Notebook. The data preparation process is conducted at the beginning to have a better understanding of the dataset and smother the analysis process as well. In the data preprocessing phase, the Python script utilizes the Pandas library to read the "ckd.csv" dataset, replacing missing values, mapping nominal variables to binary values, and handling missing data by filling with mean values. The cleaned dataset is saved as "cleaned_data.csv." The exploration phase involves analyzing dataset characteristics, such as length, shape, and statistical summaries.

The data is then split into training and testing sets. Five machine learning models, namely Decision Tree, Random Forest, Support Vector Machine, Logistic Regression, and Naive Bayes are employed and compared based on accuracy, performance metrics, and error calculations. The Random Forest model emerges as the most accurate and reliable. The graphical user interface (GUI) built with Tkinter allows users to input medical data for predicting Chronic Kidney Disease using the Random Forest model, offering an intuitive platform with a visually appealing design.

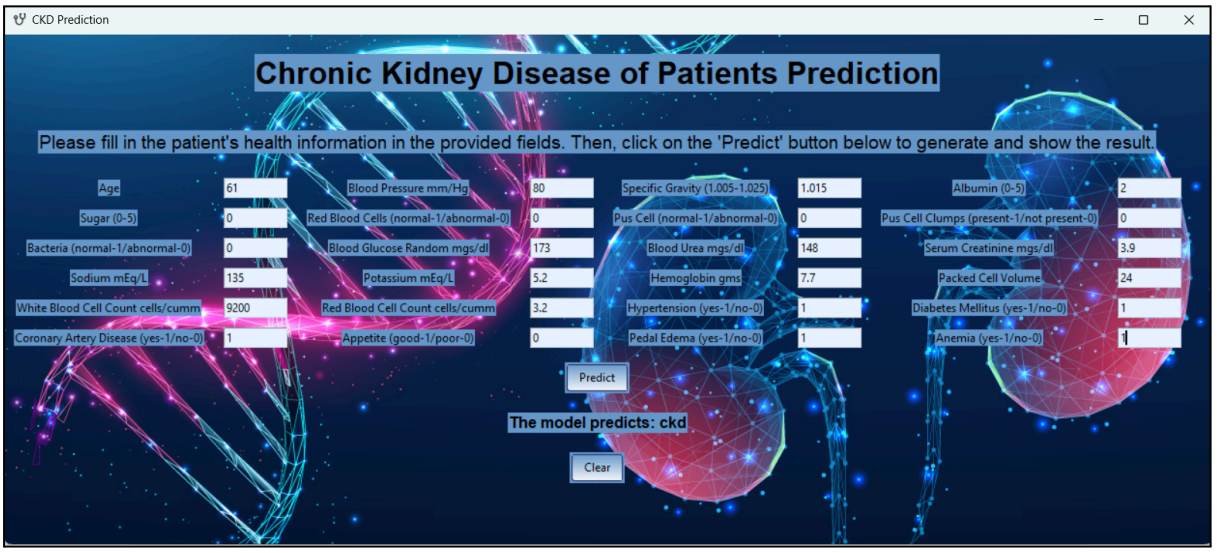


Figure 4.3 CKD Prediction Output

During our collaborative study, we inputted data encompassing a range of patients with Chronic Kidney Disease (CKD). The machine learning model, specifically the Random Forest classifier, demonstrated accurate predictions for CKD based on the entered patient

data. This successful outcome underscores the reliability and effectiveness of the model in assessing and predicting the presence of CKD in diverse patient scenarios.

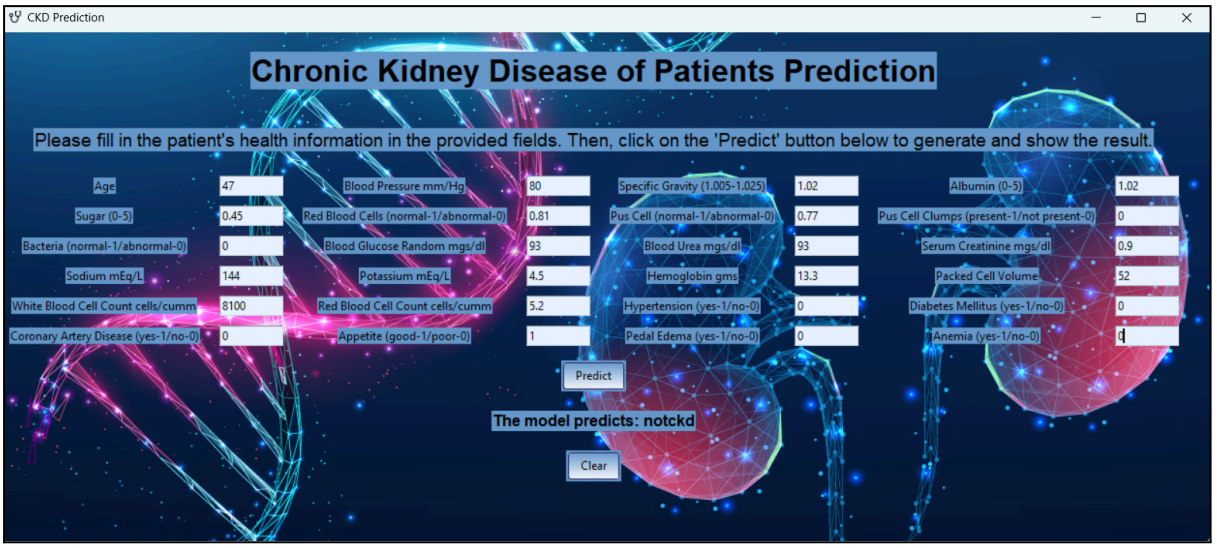


Figure 4.4 NOTCKD Prediction Output

In contrast, our findings revealed that the predictive accuracy of the model extends beyond just identifying Chronic Kidney Disease (CKD); it also demonstrates precision in predicting cases classified as not having CKD. The model's ability to provide accurate predictions for both CKD and non-CKD instances underscores its robustness and reliability across diverse health conditions.

5.0 CONCLUSION

In conclusion, the developed CKD prediction system offers a valuable tool for healthcare professionals to assess the risk of Chronic Kidney Disease (CKD) in patients. The system leverages machine learning algorithms including decision tree, support vector machine, random forest, logistic regression, and Naive Bayes, trained on a comprehensive dataset of patient data, to provide accurate and reliable predictions. The model's ability to provide accurate predictions for both CKD and non-CKD instances underscores its robustness and reliability across diverse health conditions. This dual proficiency enhances the model's utility as a versatile diagnostic tool, showcasing its effectiveness in offering accurate assessments for a broader spectrum of patients, both with and without CKD. The comprehensive and accurate predictive capabilities make this model a valuable asset in clinical settings, contributing to more informed medical decision-making.

The user-friendly interface allows for easy input of patient information, making it accessible to healthcare professionals of all levels. By utilising this system, healthcare providers can enhance the early detection and management of CKD, leading to improved patient outcomes and reduced healthcare costs. Furthermore, the system's ability to provide personalised risk assessments can empower individuals to take proactive steps in managing their health and preventing the progression of CKD. Overall, this project demonstrates the potential of machine learning in revolutionising healthcare by providing accurate, accessible, and timely diagnostic tools.

6.0 REFERENCES

1. Almustafa, K.M. (2021) ‘Prediction of chronic kidney disease using different classification algorithms’, *Informatics in Medicine Unlocked*, 24, p. 100631. doi:10.1016/j.imu.2021.100631.
2. Aqlan, H.A., Ahmed, S. and Danti, A. (2017) ‘Death prediction and analysis using web mining techniques’, *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)* [Preprint]. doi:10.1109/icaccs.2017.8014715.
3. Debal, D.A. and Sitote, T.M. (2022) ‘Chronic kidney disease prediction using machine learning techniques’, Journal of Big Data, 9(1). doi:10.1186/s40537-022-00657-5.
4. Nishat, M. et al. (2018) ‘A comprehensive analysis on detecting chronic kidney disease by employing machine learning algorithms’, EAI Endorsed Transactions on Pervasive Health and Technology, p. 170671. doi:10.4108/eai.13-8-2021.170671.
5. Pinto, A. et al. (2020) ‘Data mining to predict early stage chronic kidney disease’, Procedia Computer Science, 177, pp. 562–567. doi:10.1016/j.procs.2020.10.079.
6. Saleh, Ehab and Bin Abd Kadir, Mohd Fadzil, Prediction of Chronic Kidney Disease Using Data Mining Techniques. http://dx.doi.org/10.2139/ssrn.4022160.
7. Snegha, J. *et al.* (2020) ‘Chronic kidney disease prediction using data mining’, *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)* [Preprint]. doi:10.1109/ic-etite47903.2020.482.
8. V. Kunwar, K. Chandel, A. S. Sabitha and A. Bansal, "Chronic Kidney Disease analysis using data mining classification techniques," 2016 6th International Conference - Cloud System and Big Data Engineering (Confluence), Noida, India, 2016, pp. 300-305, doi: 10.1109/CONFLUENCE.2016.7508132.

APPENDIX_GROUP B_CKD PREDICTION

DM Project Full Coding

In [1]:
```
# Importing the required packages
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report, roc_curv
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import seaborn as sns
import tkinter as tk
from tkinter import ttk
from ttkthemes import ThemedStyle
import joblib
```

In [2]:
```
# Data Exploration
balance_data = pd.read_csv('cleaned_data.csv', sep=',', header=0)
print("Dataset Length: ", len(balance_data))
print("Dataset Shape: ", balance_data.shape)
print("Data Exploration:")
print(balance_data.head())
print(balance_data.describe())
```

```
Dataset Length:  400
Dataset Shape:  (400, 25)
Data Exploration:
    Age    Bp     Sg   Al   Su   Rbc   Pc  Pcc   Ba     Bgr  ...   Pcv  \
0  48.0  80.0  1.020  1.0  0.0  0.81  1.0  0.0  0.0  121.00  ...  44.0
1   7.0  50.0  1.020  4.0  0.0  0.81  1.0  0.0  0.0  148.04  ...  38.0
2  62.0  80.0  1.010  2.0  3.0  1.00  1.0  0.0  0.0  423.00  ...  31.0
3  48.0  70.0  1.005  4.0  0.0  1.00  0.0  1.0  0.0  117.00  ...  32.0
4  51.0  80.0  1.010  2.0  0.0  1.00  1.0  0.0  0.0  106.00  ...  35.0

     Wbcc  Rbcc  Htn  Dm  Cad  Appet   pe  Ane  Class
0  7800.0  5.20  1.0   1  0.0    1.0  0.0  0.0    ckd
1  6000.0  4.71  0.0   0  0.0    1.0  0.0  0.0    ckd
2  7500.0  4.71  0.0   1  0.0    0.0  0.0  1.0    ckd
3  6700.0  3.90  1.0   0  0.0    0.0  1.0  1.0    ckd
4  7300.0  4.60  0.0   0  0.0    1.0  0.0  0.0    ckd

[5 rows x 25 columns]
              Age          Bp          Sg          Al          Su          Rbc  \
count  400.000000  400.000000  400.000000  400.000000  400.000000  400.000000
mean    51.483300   76.469100    1.017712    1.017300    0.450125    0.810300
std     16.974966   13.476298    0.005434    1.272318    1.029487    0.308983
min      2.000000   50.000000    1.005000    0.000000    0.000000    0.000000
25%     42.000000   70.000000    1.015000    0.000000    0.000000    0.810000
50%     54.000000   78.235000    1.020000    1.000000    0.000000    1.000000
75%     64.000000   80.000000    1.020000    2.000000    0.450000    1.000000
max     90.000000  180.000000    1.025000    5.000000    5.000000    1.000000

               Pc         Pcc          Ba         Bgr  ...        Hemo  \
count  400.000000  400.000000  400.000000  400.000000  ...  400.000000
mean     0.772625    0.106100    0.055600  148.036900  ...   12.526900
std      0.383751    0.306756    0.228199   74.782634  ...    2.716171
min      0.000000    0.000000    0.000000   22.000000  ...    3.100000
25%      0.770000    0.000000    0.000000  101.000000  ...   10.875000
50%      1.000000    0.000000    0.000000  126.000000  ...   12.530000
75%      1.000000    0.000000    0.000000  150.000000  ...   14.625000
max      1.000000    1.000000    1.000000  490.000000  ...   17.800000

              Pcv          Wbcc        Rbcc         Htn         Dm  \
count  400.000000    400.000000  400.000000  400.000000  400.00000
mean    38.883700   8406.121800    4.708275    0.369350    0.34250
std      8.151082   2523.219976    0.840315    0.482023    0.47514
min      9.000000   2200.000000    2.100000    0.000000    0.00000
25%     34.000000   6975.000000    4.500000    0.000000    0.00000
50%     38.880000   8406.120000    4.710000    0.000000    0.00000
75%     44.000000   9400.000000    5.100000    1.000000    1.00000
max     54.000000  26400.000000    8.000000    1.000000    1.00000

              Cad       Appet          pe         Ane
count  400.000000  400.000000  400.000000  400.000000
mean     0.085450    0.794475    0.190475    0.150375
std      0.279166    0.404077    0.392677    0.357440
min      0.000000    0.000000    0.000000    0.000000
25%      0.000000    1.000000    0.000000    0.000000
50%      0.000000    1.000000    0.000000    0.000000
75%      0.000000    1.000000    0.000000    0.000000
max      1.000000    1.000000    1.000000    1.000000

[8 rows x 24 columns]
```

In [3]:
```
# Building Phase
# Separating the target variable
X = balance_data.values[:, 0:24]
Y = balance_data.values[:, -1]
```

In [4]:
```
# Splitting the dataset into train and test
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=100)
```

Decision Tree

In [5]:
```
# Hyperparameter Tuning for decision tree
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 5, 7],
    'min_samples_leaf': [3, 5, 7]
}

clf = DecisionTreeClassifier(random_state=100)
grid_search = GridSearchCV(estimator=clf, param_grid=param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
print("Best Hyperparameters:", grid_search.best_params_)
tuned_clf = grid_search.best_estimator_
```

```
Best Hyperparameters: {'criterion': 'gini', 'max_depth': 3, 'min_samples_leaf': 7}
```

In [6]:
```
# Training phase
clf_gini = DecisionTreeClassifier(criterion="gini", random_state=100, max_depth=3, min_sample
clf_gini.fit(X_train, y_train)
```

Out[6]:

▾ DecisionTreeClassifier

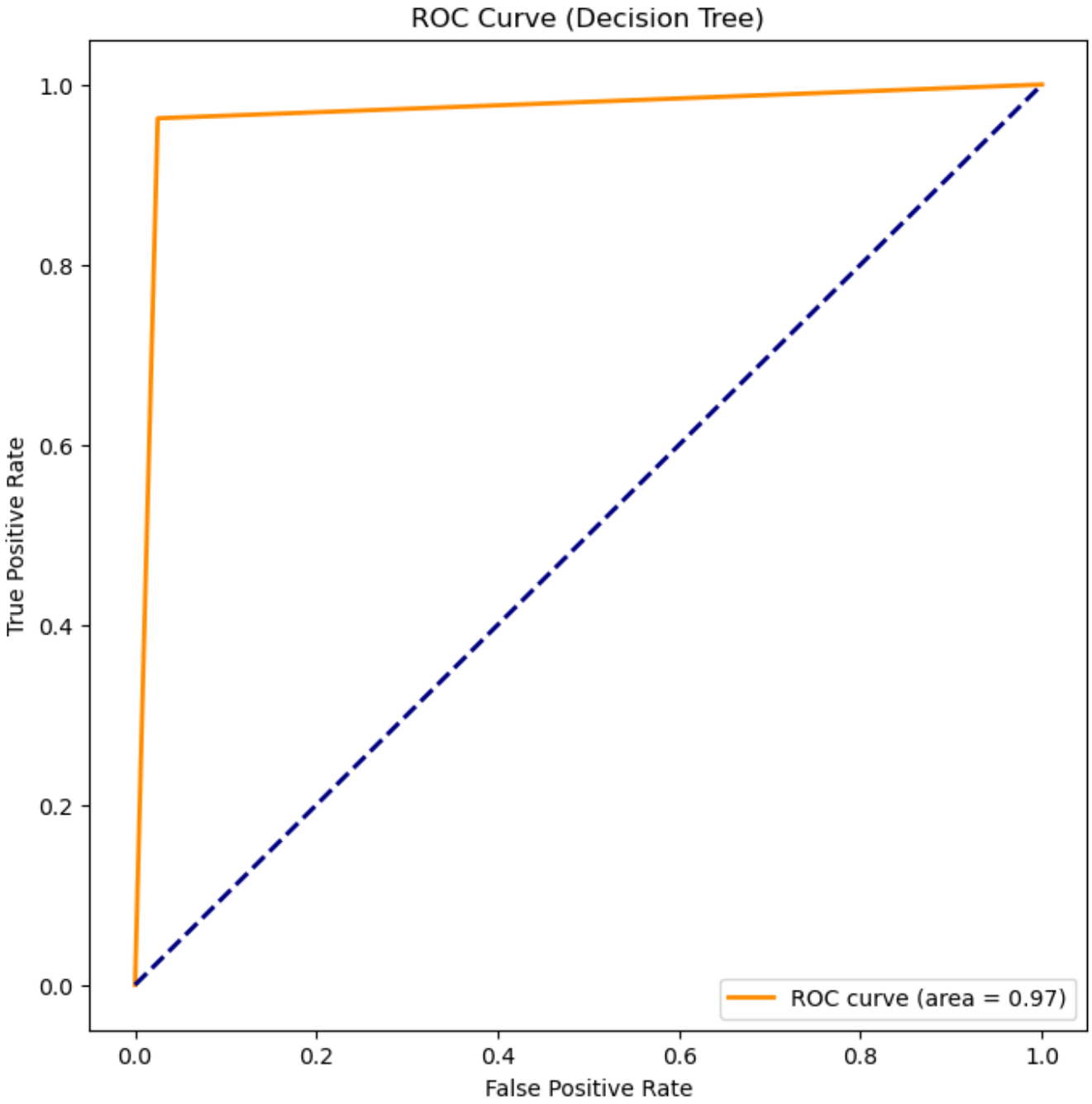
```
DecisionTreeClassifier(max_depth=3, min_samples_leaf=7, random_state=100)
```

In [7]:
```
# Operational Phase
print("\nResults Using Gini Index:")
y_pred_gini = clf_gini.predict(X_test)
y_test_binary = np.where(y_test == 'ckd', 1, 0)
y_pred_gini_binary = np.where(y_pred_gini == 'ckd', 1, 0)
# Metrics for Decision Tree
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_gini))
print("Accuracy:", accuracy_score(y_test,y_pred_gini) * 100)
print("Report:\n", classification_report(y_test, y_pred_gini))
print("F1 Score:", f1_score(y_test_binary, y_pred_gini_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_gini_binary))
```

```
Results Using Gini Index:
Confusion Matrix:
 [[77  3]
 [ 1 39]]
Accuracy: 96.66666666666667
Report:
               precision    recall  f1-score   support

         ckd       0.99      0.96      0.97        80
      notckd       0.93      0.97      0.95        40

    accuracy                           0.97       120
   macro avg       0.96      0.97      0.96       120
weighted avg       0.97      0.97      0.97       120

F1 Score: 0.9746835443037976
AUC Score: 0.96875
```

In [8]:
```
# ROC Curve for Decision Tree
fpr_dt, tpr_dt, _ = roc_curve(y_test_binary,y_pred_gini_binary)
roc_auc_dt = auc(fpr_dt, tpr_dt)

plt.figure(figsize=(8, 8))
plt.plot(fpr_dt, tpr_dt, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(r
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Decision Tree)')
```

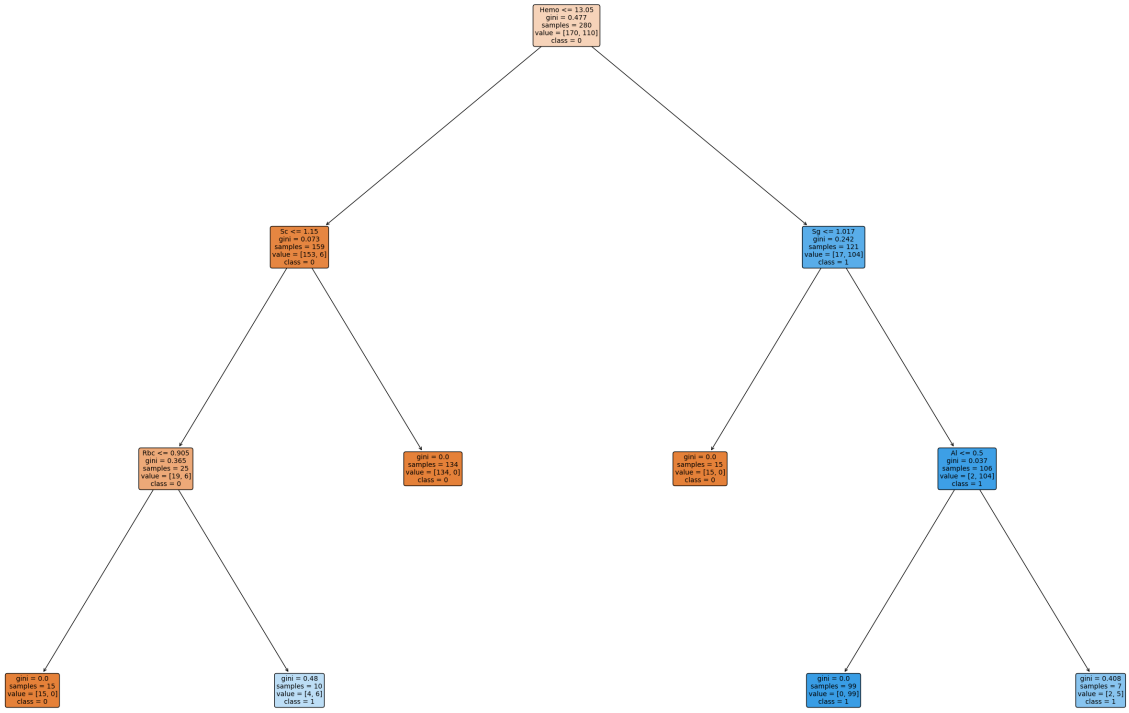
```
plt.legend(loc="lower right")
plt.show()
```



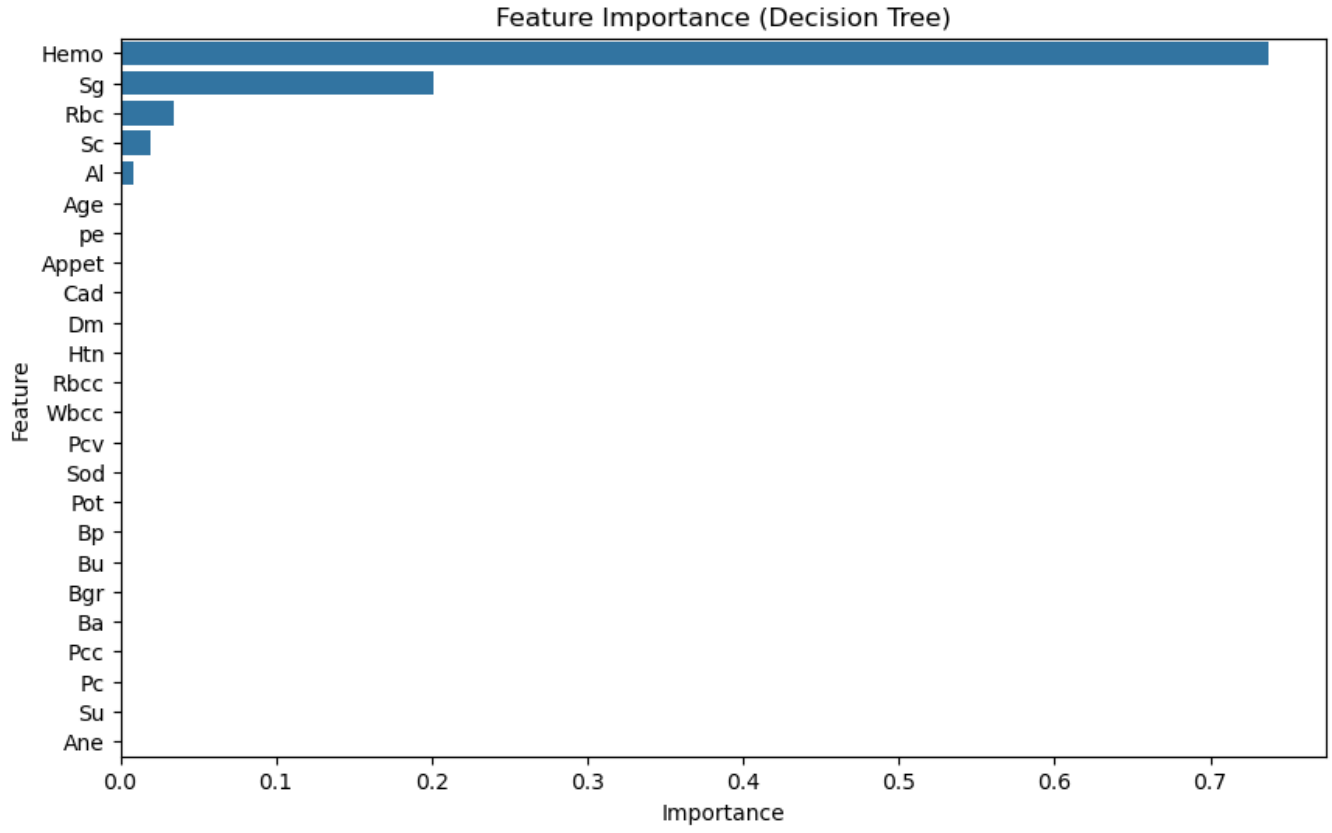
In [9]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Decision Tree
mse_dt = mean_squared_error(y_test_binary,y_pred_gini_binary)
rmse_dt = np.sqrt(mse_dt)
print("Mean Squared Error (MSE):", mse_dt)
print("Root Mean Squared Error (RMSE):", rmse_dt)
```

```
Mean Squared Error (MSE): 0.03333333333333333
Root Mean Squared Error (RMSE): 0.18257418583505536
```

In [10]:
```
# Visualizing Decision Trees using plot_tree
plt.figure(figsize=(36, 24))
plot_tree(clf_gini, filled=True, feature_names=balance_data.columns[:-1], class_names=["0", "
plt.show()
```



In [11]:
```
# Feature Importance for Decision Tree
feature_importance_dt = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Importance': clf
feature_importance_dt = feature_importance_dt.sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance_dt)
plt.title('Feature Importance (Decision Tree)')
plt.show()
```



In [12]:
```
# Cross-Validation for Decision Tree
cv_results_dt = cross_val_score(clf_gini, X, Y, cv=5, scoring='accuracy')
print("Cross-Validation Results (Decision Tree):", cv_results_dt)
print("Mean Accuracy (Decision Tree):", np.mean(cv_results_dt))
```

```
Cross-Validation Results (Decision Tree): [0.95   1.     0.9375 0.975  0.9875]
Mean Accuracy (Decision Tree): 0.9700000000000001
```

Random Forest

In [13]:
```python
# Hyperparameter Tuning for Random Forest
param_grid_rf = {
    'n_estimators': [50, 100, 150],
    'max_depth': [3, 5, 7],
    'min_samples_leaf': [3, 5, 7],
    'max_features': ['sqrt', 'log2']
}

clf_rf = RandomForestClassifier(random_state=100)
grid_search_rf = GridSearchCV(estimator=clf_rf, param_grid=param_grid_rf, cv=5, scoring='accu
grid_search_rf.fit(X_train, y_train)
print("Best Hyperparameters (Random Forest):", grid_search_rf.best_params_)
tuned_clf_rf = grid_search_rf.best_estimator_
```

```
Best Hyperparameters (Random Forest): {'max_depth': 3, 'max_features': 'sqrt', 'min_samples_l
eaf': 3, 'n_estimators': 50}
```

In [14]:
```python
# Training phase for Random Forest
clf_rf = RandomForestClassifier(n_estimators=50, max_depth=3, min_samples_leaf=3, max_feature
clf_rf.fit(X_train, y_train)
```

Out[14]:

RandomForestClassifier

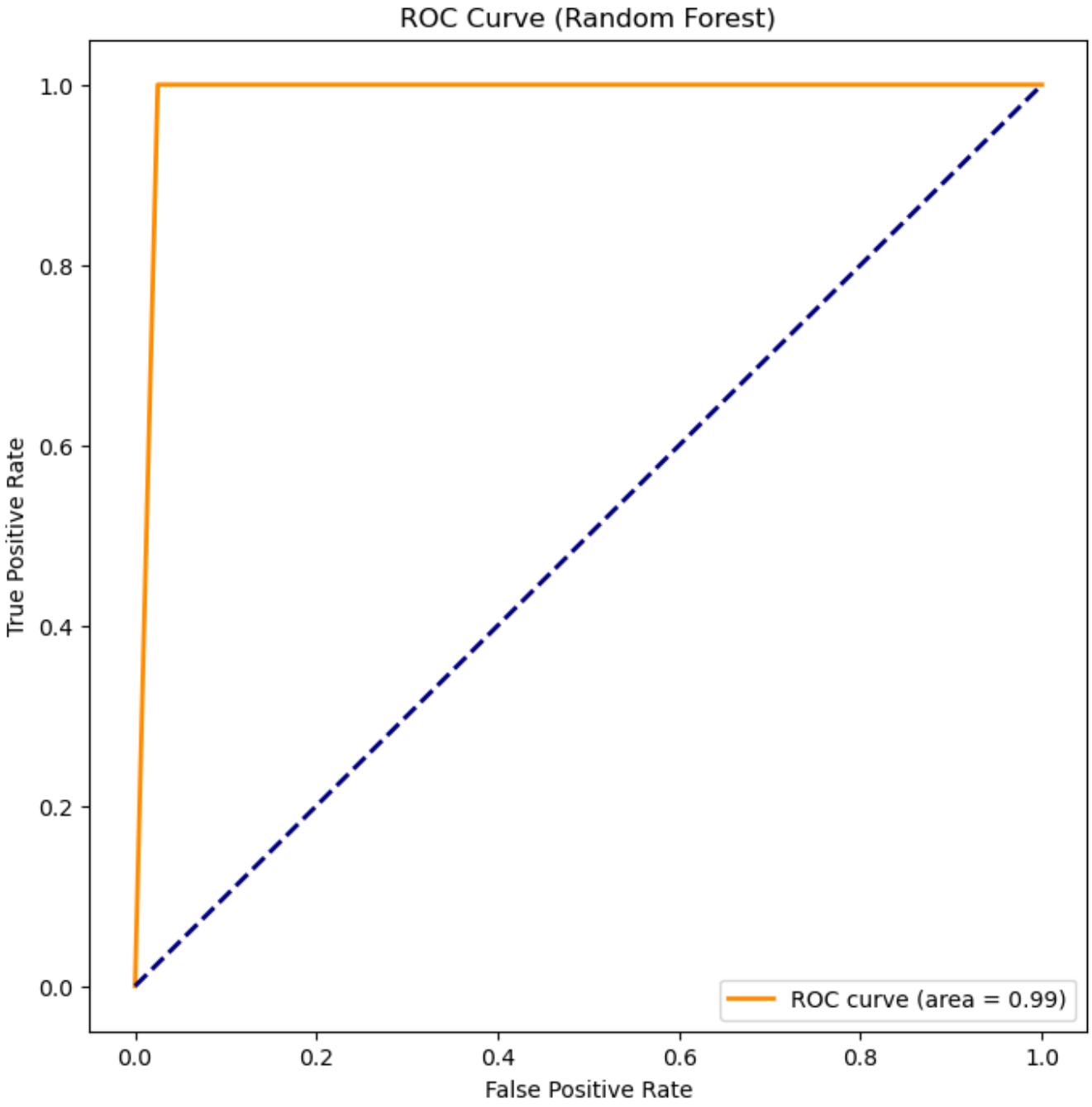
```
RandomForestClassifier(max_depth=3, min_samples_leaf=3, n_estimators=50,
                       random_state=100)
```

In [15]:
```python
# Operational Phase for Random Forest
print("\nResults Using Random Forest:")
y_pred_rf = clf_rf.predict(X_test)
y_test_binary = np.where(y_test == 'ckd', 1, 0)
y_pred_rf_binary = np.where(y_pred_rf == 'ckd', 1, 0)
# Metrics for Random Forest
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
print("Accuracy:", accuracy_score(y_test, y_pred_rf) * 100)
print("Report:\n", classification_report(y_test, y_pred_rf))
print("F1 Score:", f1_score(y_test_binary, y_pred_rf_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_rf_binary))
```

```
Results Using Random Forest:
Confusion Matrix:
 [[80  0]
 [ 1 39]]
Accuracy: 99.16666666666667
Report:
               precision    recall  f1-score   support

         ckd       0.99      1.00      0.99        80
      notckd       1.00      0.97      0.99        40

    accuracy                           0.99       120
   macro avg       0.99      0.99      0.99       120
weighted avg       0.99      0.99      0.99       120

F1 Score: 0.9937888198757764
AUC Score: 0.9874999999999999
```

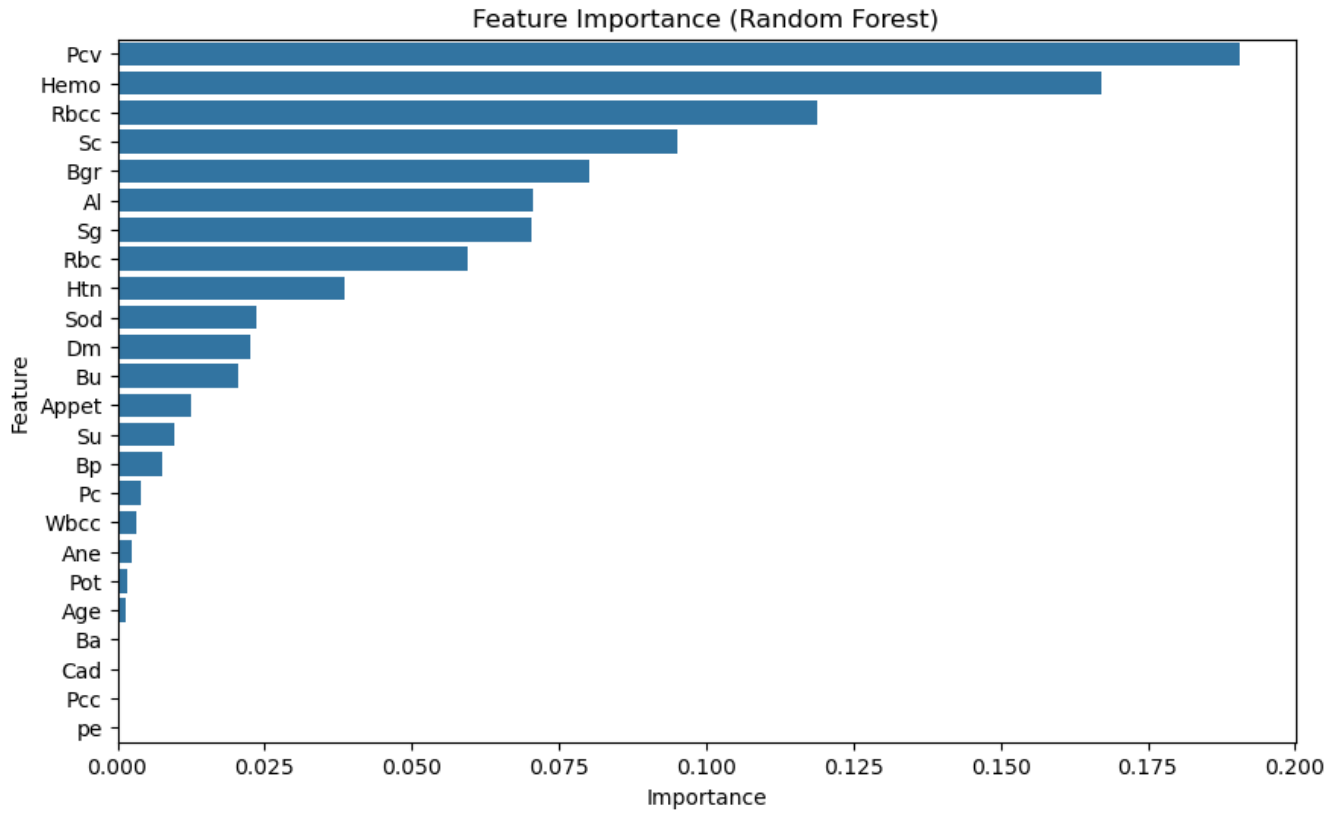
In [16]:
```python
# ROC Curve for Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_test_binary, y_pred_rf_binary)
roc_auc_rf = auc(fpr_rf, tpr_rf)

plt.figure(figsize=(8, 8))
```

```
plt.plot(fpr_rf, tpr_rf, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(r
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Random Forest)')
plt.legend(loc="lower right")
plt.show()
```



In [17]:

```
# Feature Importance for Random Forest
feature_importance_rf = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Importance': clf
feature_importance_rf = feature_importance_rf.sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance_rf)
plt.title('Feature Importance (Random Forest)')
plt.show()
```



In [18]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Random Forest
mse_rf = mean_squared_error(y_test_binary, y_pred_rf_binary)
rmse_rf = np.sqrt(mse_rf)
print("Mean Squared Error (MSE):", mse_rf)
print("Root Mean Squared Error (RMSE):", rmse_rf)
```

```
Mean Squared Error (MSE): 0.008333333333333333
Root Mean Squared Error (RMSE): 0.09128709291752768
```

In [19]:
```
# Cross-Validation for Random Forest
cv_results_rf = cross_val_score(clf_rf, X, Y, cv=5, scoring='accuracy')
print("Cross-Validation Results (Random Forest):", cv_results_rf)
print("Mean Accuracy (Random Forest):", np.mean(cv_results_rf))
```

```
Cross-Validation Results (Random Forest): [0.9875 1.     0.9625 1.     1.    ]
Mean Accuracy (Random Forest): 0.99
```

Support Vector Machine

In [20]:
```
# Training phase for Support Vector Machine (SVM)
clf_svm = SVC(kernel='rbf', random_state=100)
clf_svm.fit(X_train, y_train)
```

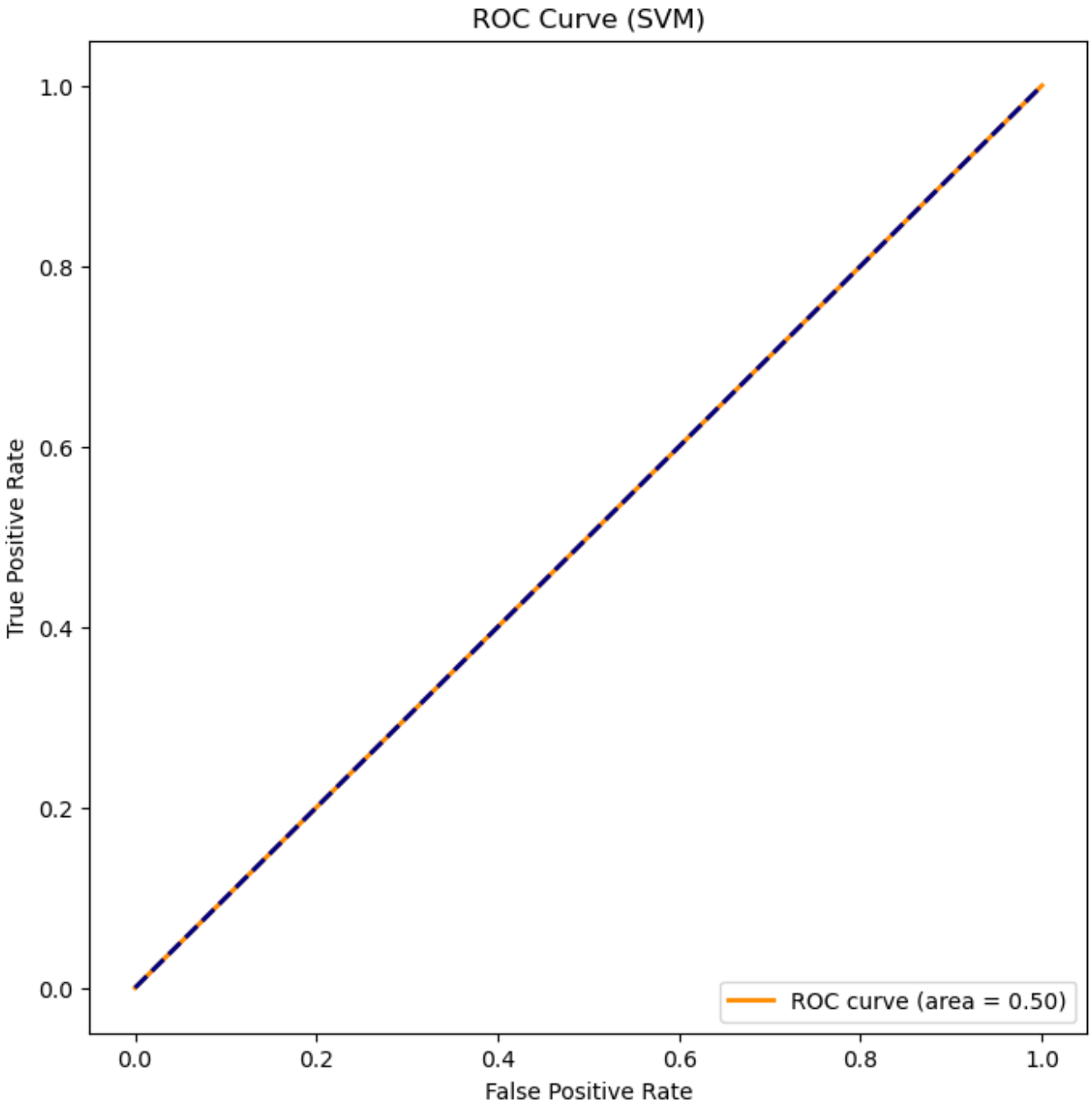
Out[20]:
```
SVC
SVC(random_state=100)
```

In [21]:
```
# Operational Phase for SVM
print("\nResults Using SVM:")
y_pred_svm = clf_svm.predict(X_test)
y_test_binary = np.where(y_test == 'ckd', 1, 0)
y_pred_svm_binary = np.where(y_pred_svm == 'ckd', 1, 0)
# Metrics for SVM
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_svm))
print("Accuracy:", accuracy_score(y_test, y_pred_svm) * 100)
print("Report:\n", classification_report(y_test, y_pred_svm, zero_division=1))
print("F1 Score:", f1_score(y_test_binary, y_pred_svm_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_svm_binary))
```

```
Results Using SVM:
Confusion Matrix:
 [[80  0]
 [40  0]]
Accuracy: 66.66666666666666
Report:
              precision    recall  f1-score   support

         ckd       0.67      1.00      0.80        80
      notckd       1.00      0.00      0.00        40

    accuracy                           0.67       120
   macro avg       0.83      0.50      0.40       120
weighted avg       0.78      0.67      0.53       120

F1 Score: 0.8
AUC Score: 0.5
```

In [22]:

```
# ROC Curve for SVM
fpr_svm, tpr_svm, _ = roc_curve(y_test_binary, y_pred_svm_binary)
roc_auc_svm = auc(fpr_svm, tpr_svm)

plt.figure(figsize=(8, 8))
plt.plot(fpr_svm, tpr_svm, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (SVM)')
plt.legend(loc="lower right")
plt.show()
```



In [23]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for SVM
mse_svm = mean_squared_error(y_test_binary, y_pred_svm_binary)
rmse_svm = np.sqrt(mse_svm)
print("Mean Squared Error (MSE) for SVM:", mse_svm)
print("Root Mean Squared Error (RMSE) for SVM:", rmse_svm)
```

```
Mean Squared Error (MSE) for SVM: 0.3333333333333333
Root Mean Squared Error (RMSE) for SVM: 0.5773502691896257
```

In [24]:
```
# Cross-Validation for SVM
cv_results_svm = cross_val_score(clf_svm, X, Y, cv=5, scoring='accuracy')
print("Cross-Validation Results (SVM):", cv_results_svm)
print("Mean Accuracy (SVM):", np.mean(cv_results_svm))
```

```
Cross-Validation Results (SVM): [0.625 0.625 0.625 0.625 0.625]
Mean Accuracy (SVM): 0.625
```

Logistic Regression

In [25]:
```
# Hyperparameter Tuning for Logistic Regression
param_grid_lr = {
    'penalty': ['l1', 'l2'],
    'C': [0.001, 0.01, 0.1, 1, 10, 100]
}
```

```
clf_lr = LogisticRegression(random_state=100, solver='liblinear')
grid_search_lr = GridSearchCV(estimator=clf_lr, param_grid=param_grid_lr, cv=5, scoring='accu
grid_search_lr.fit(X_train, y_train)
print("Best Hyperparameters (Logistic Regression):", grid_search_lr.best_params_)
tuned_clf_lr = grid_search_lr.best_estimator_
```

```
Best Hyperparameters (Logistic Regression): {'C': 1, 'penalty': 'l2'}
```

In [26]:
```
# Training phase for Logistic Regression
clf_lr = LogisticRegression(C=1, penalty='l2', max_iter=100, random_state=100)
clf_lr.fit(X_train, y_train)
```

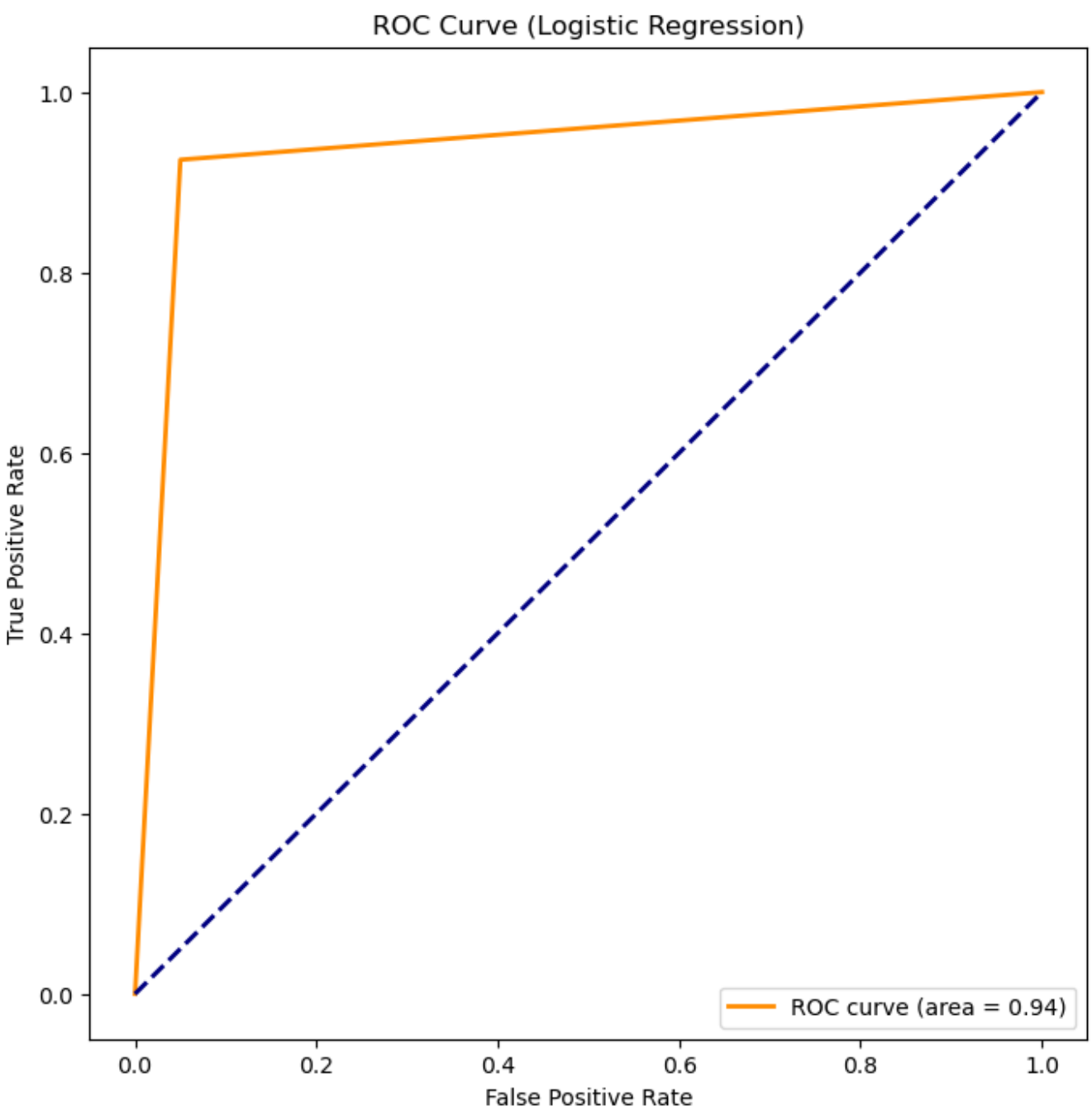
```
C:\Users\PC\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:460: ConvergenceWar
ning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[26]:
```
LogisticRegression
LogisticRegression(C=1, random_state=100)
```

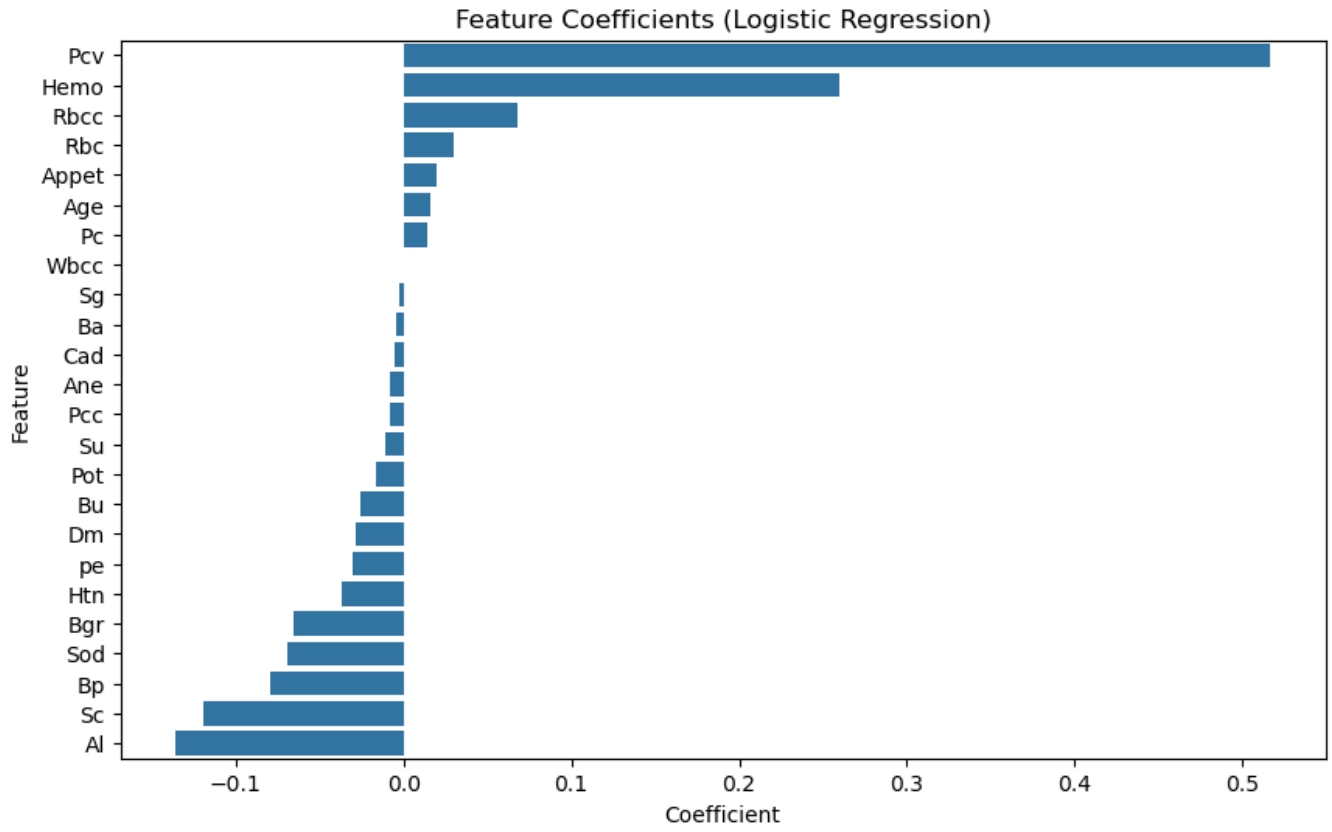
In [27]:
```
# Operational Phase for Logistic Regression
print("\nResults Using Logistic Regression:")
y_pred_lr = clf_lr.predict(X_test)
y_pred_lr_binary = np.where(y_pred_lr == 'ckd', 1, 0)
# Metrics for Logistic Regression
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_lr))
print("Accuracy:", accuracy_score(y_test, y_pred_lr) * 100)
print("Report:\n", classification_report(y_test, y_pred_lr))
print("F1 Score:", f1_score(y_test_binary, y_pred_lr_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_lr_binary))
```

```
Results Using Logistic Regression:
Confusion Matrix:
 [[74  6]
 [ 2 38]]
Accuracy: 93.33333333333333
Report:
               precision    recall  f1-score   support

         ckd       0.97      0.93      0.95        80
      notckd       0.86      0.95      0.90        40

    accuracy                           0.93       120
   macro avg       0.92      0.94      0.93       120
weighted avg       0.94      0.93      0.93       120

F1 Score: 0.9487179487179489
AUC Score: 0.9374999999999999
```

In [28]:
```
# ROC Curve for Logistic Regression
fpr_lr, tpr_lr, _ = roc_curve(y_test_binary, y_pred_lr_binary)
roc_auc_lr = auc(fpr_lr, tpr_lr)

plt.figure(figsize=(8, 8))
plt.plot(fpr_lr, tpr_lr, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(r
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Logistic Regression)')
plt.legend(loc="lower right")
plt.show()
```



In [29]:
```
# Feature Coefficients for Logistic Regression
feature_coefficients_lr = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Coefficient':
feature_coefficients_lr = feature_coefficients_lr.sort_values(by='Coefficient', ascending=Fal

plt.figure(figsize=(10, 6))
sns.barplot(x='Coefficient', y='Feature', data=feature_coefficients_lr)
plt.title('Feature Coefficients (Logistic Regression)')
plt.show()
```



In [30]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Logistic Regression
mse_lr = mean_squared_error(y_test_binary, y_pred_lr_binary)
rmse_lr = np.sqrt(mse_lr)
print("Mean Squared Error (MSE):", mse_lr)
print("Root Mean Squared Error (RMSE):", rmse_lr)
```

```
Mean Squared Error (MSE): 0.06666666666666667
Root Mean Squared Error (RMSE): 0.2581988897471611
```

Naive Bayes

In [31]:
```
# Naive Bayes - Gaussian Naive Bayes does not have many hyperparameters to tune
clf_nb = GaussianNB()
clf_nb.fit(X_train, y_train)
```

Out[31]:
```
▾ GaussianNB
GaussianNB()
```

In [32]:
```
# Operational Phase for Naive Bayes
print("\nResults Using Naive Bayes:")
y_pred_nb = clf_nb.predict(X_test)
y_pred_nb_binary = np.where(y_pred_nb == 'ckd', 1, 0)
```

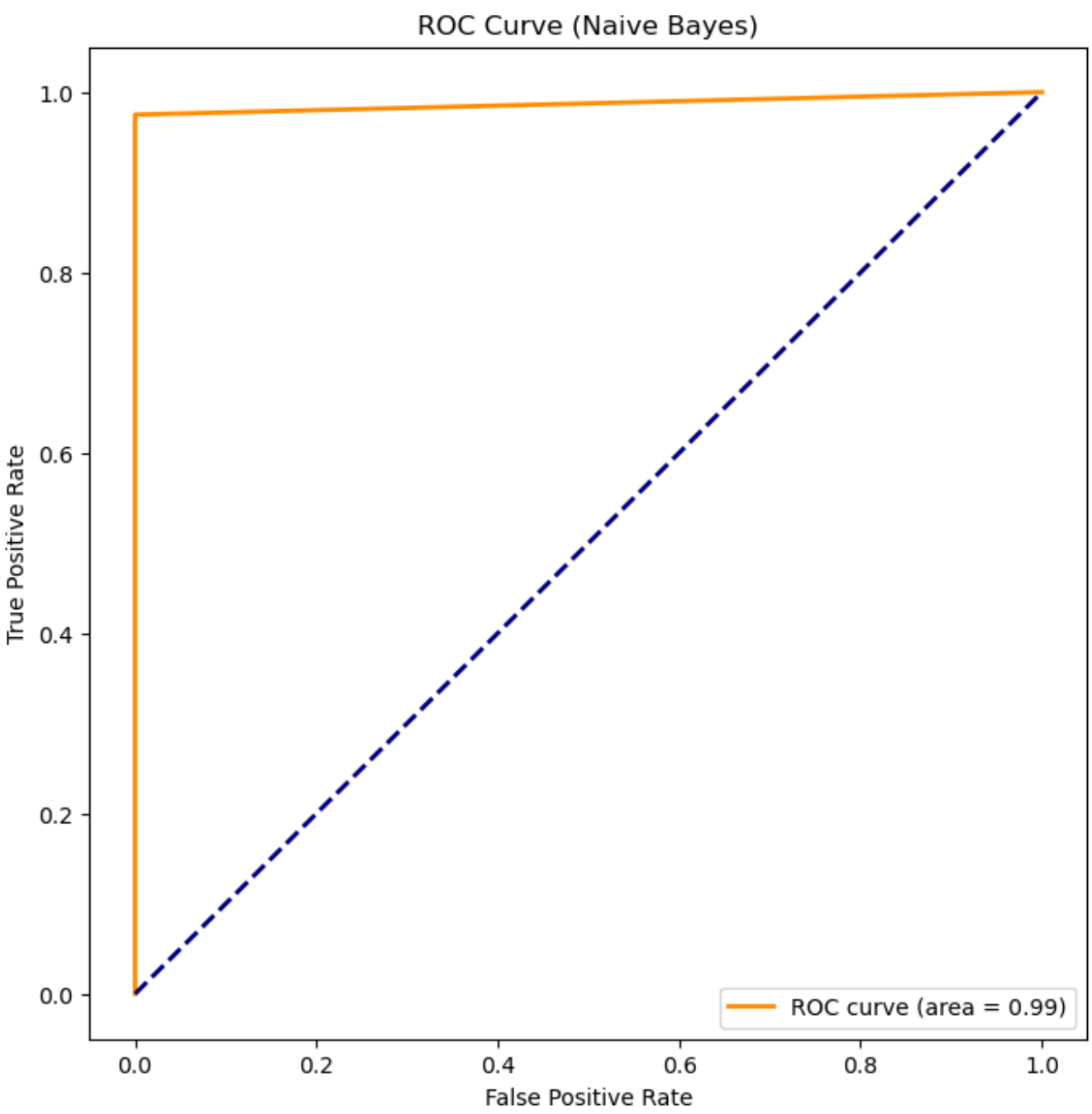
```
Results Using Naive Bayes:
```

In [33]:
```
# Metrics for Naive Bayes
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_nb))
print("Accuracy:", accuracy_score(y_test, y_pred_nb) * 100)
print("Report:\n", classification_report(y_test, y_pred_nb))
print("F1 Score:", f1_score(y_test_binary, y_pred_nb_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_nb_binary))
```

```
Confusion Matrix:
 [[78  2]
 [ 0 40]]
Accuracy: 98.33333333333333
Report:
              precision    recall  f1-score   support

         ckd       1.00      0.97      0.99        80
      notckd       0.95      1.00      0.98        40

    accuracy                           0.98       120
   macro avg       0.98      0.99      0.98       120
weighted avg       0.98      0.98      0.98       120

F1 Score: 0.9873417721518987
AUC Score: 0.9875
```

In [34]:

```
# ROC Curve for Naive Bayes
fpr_nb, tpr_nb, _ = roc_curve(y_test_binary, y_pred_nb_binary)
roc_auc_nb = auc(fpr_nb, tpr_nb)

plt.figure(figsize=(8, 8))
plt.plot(fpr_nb, tpr_nb, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(r
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Naive Bayes)')
plt.legend(loc="lower right")
plt.show()
```



In [35]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Naive Bayes
mse_nb = mean_squared_error(y_test_binary, y_pred_nb_binary)
rmse_nb = np.sqrt(mse_nb)
print("Mean Squared Error (MSE):", mse_nb)
print("Root Mean Squared Error (RMSE):", rmse_nb)
```

```
Mean Squared Error (MSE): 0.016666666666666666
Root Mean Squared Error (RMSE): 0.12909944487358055
```

USER GUI

In [56]:
```
# Create the main window
root = tk.Tk()

# Create the main window with a themed style
style = ThemedStyle(root)
style.set_theme("blue")

# Set the title and icon
root.title("CKD Prediction")
root.iconbitmap("Erix-Subyarko-Medical-Medicine-Medicine-Tool-Doctor-Hospital-Stethoscope.512

# Set the size of the window (width x height)
root.geometry("1250x500")  # Change the size as needed
```

```
# Load the background image
background_image = tk.PhotoImage(file="23-Blood-test_1400x700-Copy.png")  # Replace with the
background_label = tk.Label(root, image=background_image)
background_label.place(relwidth=1, relheight=1)  # Make the label cover the entire window

# Create and place the title label
title_label = ttk.Label(root, text="Chronic Kidney Disease of Patients Prediction", font=('Ar
                        style="TLabel")
title_label.grid(row=0, column=0, columnspan=8, padx=10, pady=20)

# Create and place the guideline label with increased font size
guideline_label = ttk.Label(root, text="Please fill in the patient's health information in th
                                       "Then, click on the 'Predict' button below to generate
                            font=('Arial', 14), style="TLabel")
guideline_label.grid(row=1, column=0, columnspan=8, padx=10, pady=20)



# Create and place entry widgets for each feature in a 6x4 grid
entries = []
feature_labels = [
    "Age", "Blood Pressure mm/Hg", "Specific Gravity (1.005-1.025)", "Albumin (0-5)", "Sugar
    "Red Blood Cells (normal-1/abnormal-0)", "Pus Cell (normal-1/abnormal-0)", "Pus Cell Clum
    "Blood Glucose Random mgs/dl", "Blood Urea mgs/dl", "Serum Creatinine mgs/dl", "Sodium mE
    "Potassium mEq/L", "Hemoglobin gms", "Packed Cell Volume", "White Blood Cell Count cells/
    "Red Blood Cell Count cells/cumm", "Hypertension (yes-1/no-0)", "Diabetes Mellitus (yes-1
    "Coronary Artery Disease (yes-1/no-0)", "Appetite (good-1/poor-0)", "Pedal Edema (yes-1/n
]

row, col = 2, 0 # Start from row 2 (below the guideline label)
for i, label_text in enumerate(feature_labels):
    label = ttk.Label(root, text=label_text, style="TLabel")
    label.grid(row=row, column=col, padx=10, pady=5)
    entry = ttk.Entry(root, style="TEntry",width=10)
    entry.grid(row=row, column=col + 1, padx=10, pady=5)
    entries.append(entry)

    col += 2  # Move to the next column
    if col > 6:
        col = 0
        row += 1  # Move to the next row

# Create and place the predict button with a themed style
predict_button = ttk.Button(root, text="Predict", command=lambda: predict(entries, clf_gini,
                            style='TButton', takefocus=False)
predict_button.grid(row=row + 2, column=0, columnspan=8, pady=10)

# Create and place the result label
result_label = ttk.Label(root, text="Prediction for User Input:", style="PLabel.TLabel")
result_label.grid(row=row + 3, column=0, columnspan=8, pady=10)

# Add styling for labels and entries
style = ThemedStyle(root)
style.configure("TLabel", background=style.lookup("TFrame", "background"))
style.configure("TEntry", background=style.lookup("TFrame", "background"))
style.configure("PLabel.TLabel", font=('Arial', 12, 'bold'))


# Function to get user input for prediction
def get_user_input(entries):
    user_input = [float(entry.get()) for entry in entries]
    user_input_array = np.array(user_input).reshape(1, -1)
    return user_input_array


# Lambda function to predict and update the result label
predict_lambda = lambda: (
    user_input := get_user_input(entries),
```

```
    result_label.config(text=f"The model predicts: {clf_rf.predict(user_input)[0]}")
)

# Create and place the predict button without using the function
predict_button = ttk.Button(root, text="Predict", command=predict_lambda, style='TButton', ta
predict_button.grid(row=row + 2, column=0, columnspan=8, pady=10)

def predict(entries, clf_gini, clf_rf, clf_svm, result_label):
    pass

def get_user_input(entries):
    user_input = [float(entry.get()) for entry in entries]
    user_input_array = np.array(user_input).reshape(1, -1)
    return user_input_array

def clear_input(entries):
    for entry in entries:
        entry.delete(0, tk.END)

def clear_output(result_label):
    result_label.config(text="Prediction for User Input:")

def on_entry_return(event):
    current_entry = event.widget
    current_row, current_col = current_entry.grid_info()["row"], current_entry.grid_info()["c

    # Calculate the next entry's row and column
    next_row = current_row
    next_col = current_col + 2
    if next_col > 6:
        next_col = 0
        next_row += 1

    # Move the focus to the next entry
    entries[next_row][next_col].focus_set()

def clear_input_and_output(entries, result_label):
    clear_input(entries)
    clear_output(result_label)

# Create and place the clear button
clear_button = ttk.Button(root, text="Clear", command=lambda: clear_input_and_output(entries,
                          style='TButton', takefocus=False)
clear_button.grid(row=row + 4, column=0, columnspan=8, pady=10)

# Run the Tkinter event loop
root.mainloop()
```

Model Comparison

Accuracy Score

In [37]:
```
results = pd.DataFrame({
    'Model': ['Support Vector Machine', 'Decision Tree', 'Random Forest', 'Logistic Regressio
    'Accuracy Score': [
        accuracy_score(y_test, y_pred_svm) * 100,
        accuracy_score(y_test, y_pred_gini) * 100,
        accuracy_score(y_test, y_pred_rf) * 100,
        accuracy_score(y_test, y_pred_lr) * 100,  # Logistic Regression
        accuracy_score(y_test, y_pred_nb) * 100   # Naive Bayes
    ]
})

result_df = results.sort_values(by='Accuracy Score', ascending=False)
result_df = result_df.set_index('Accuracy Score')
result_df
```

Out[37]:

Model	Accuracy Score
Random Forest	**99.166667**
Naive Bayes	**98.333333**
Decision Tree	**96.666667**
Logistic Regression	**93.333333**
Support Vector Machine	**66.666667**

Classification Report

In [38]:
```
from sklearn.metrics import classification_report

# Create a DataFrame for Classification Report
classification_reports = []

# SVM
report_svm = classification_report(y_test, y_pred_svm, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'SVM', **report_svm['weighted avg']})

# Decision Tree
report_dt = classification_report(y_test, y_pred_gini, output_dict=True)
classification_reports.append({'Model': 'Decision Tree', **report_dt['weighted avg']})

# Random Forest
report_rf = classification_report(y_test, y_pred_rf, output_dict=True)
classification_reports.append({'Model': 'Random Forest', **report_rf['weighted avg']})

# Logistic Regression
report_lr = classification_report(y_test, y_pred_lr, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'Logistic Regression', **report_lr['weighted avg']})

# Naive Bayes
report_nb = classification_report(y_test, y_pred_nb, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'Naive Bayes', **report_nb['weighted avg']})

# Create DataFrame
classification_df = pd.DataFrame(classification_reports)
classification_df = classification_df.set_index('Model')

# Sort DataFrame by Score in descending order
classification_df = classification_df.sort_values(by='f1-score', ascending=False).round(2)

# Print the DataFrame
print(classification_df)
```

```
                     precision  recall  f1-score  support
Model
Random Forest             0.99    0.99      0.99    120.0
Naive Bayes               0.98    0.98      0.98    120.0
Decision Tree             0.97    0.97      0.97    120.0
Logistic Regression       0.94    0.93      0.93    120.0
SVM                       0.78    0.67      0.53    120.0
```

MSE & RMSE

In [39]:
```
from sklearn.metrics import mean_squared_error
from math import sqrt

# Create a DataFrame for MSE and RMSE
mse_rmse_results = []

# SVM
```

```
mse_svm = mean_squared_error(y_test_binary, y_pred_svm_binary)
rmse_svm = sqrt(mse_svm)
mse_rmse_results.append({'Model': 'SVM', 'MSE': mse_svm, 'RMSE': rmse_svm})

# Decision Tree
mse_dt = mean_squared_error(y_test_binary, y_pred_gini_binary)
rmse_dt = sqrt(mse_dt)
mse_rmse_results.append({'Model': 'Decision Tree', 'MSE': mse_dt, 'RMSE': rmse_dt})

# Random Forest
mse_rf = mean_squared_error(y_test_binary, y_pred_rf_binary)
rmse_rf = sqrt(mse_rf)
mse_rmse_results.append({'Model': 'Random Forest', 'MSE': mse_rf, 'RMSE': rmse_rf})

# Logistic Regression
mse_lr = mean_squared_error(y_test_binary, y_pred_lr_binary)
rmse_lr = sqrt(mse_lr)
mse_rmse_results.append({'Model': 'Logistic Regression', 'MSE': mse_lr, 'RMSE': rmse_lr})

# Naive Bayes
mse_nb = mean_squared_error(y_test_binary, y_pred_nb_binary)
rmse_nb = sqrt(mse_nb)
mse_rmse_results.append({'Model': 'Naive Bayes', 'MSE': mse_nb, 'RMSE': rmse_nb})

# Create DataFrame
mse_rmse_df = pd.DataFrame(mse_rmse_results)
mse_rmse_df = mse_rmse_df.set_index('Model')

# Sort DataFrame by MSE in ascending order
mse_rmse_df = mse_rmse_df.sort_values(by='MSE', ascending=False).round(2)

# Print the DataFrame
print(mse_rmse_df)
```

```
                     MSE  RMSE
Model
SVM                 0.33  0.58
Logistic Regression 0.07  0.26
Decision Tree       0.03  0.18
Naive Bayes         0.02  0.13
Random Forest       0.01  0.09
```

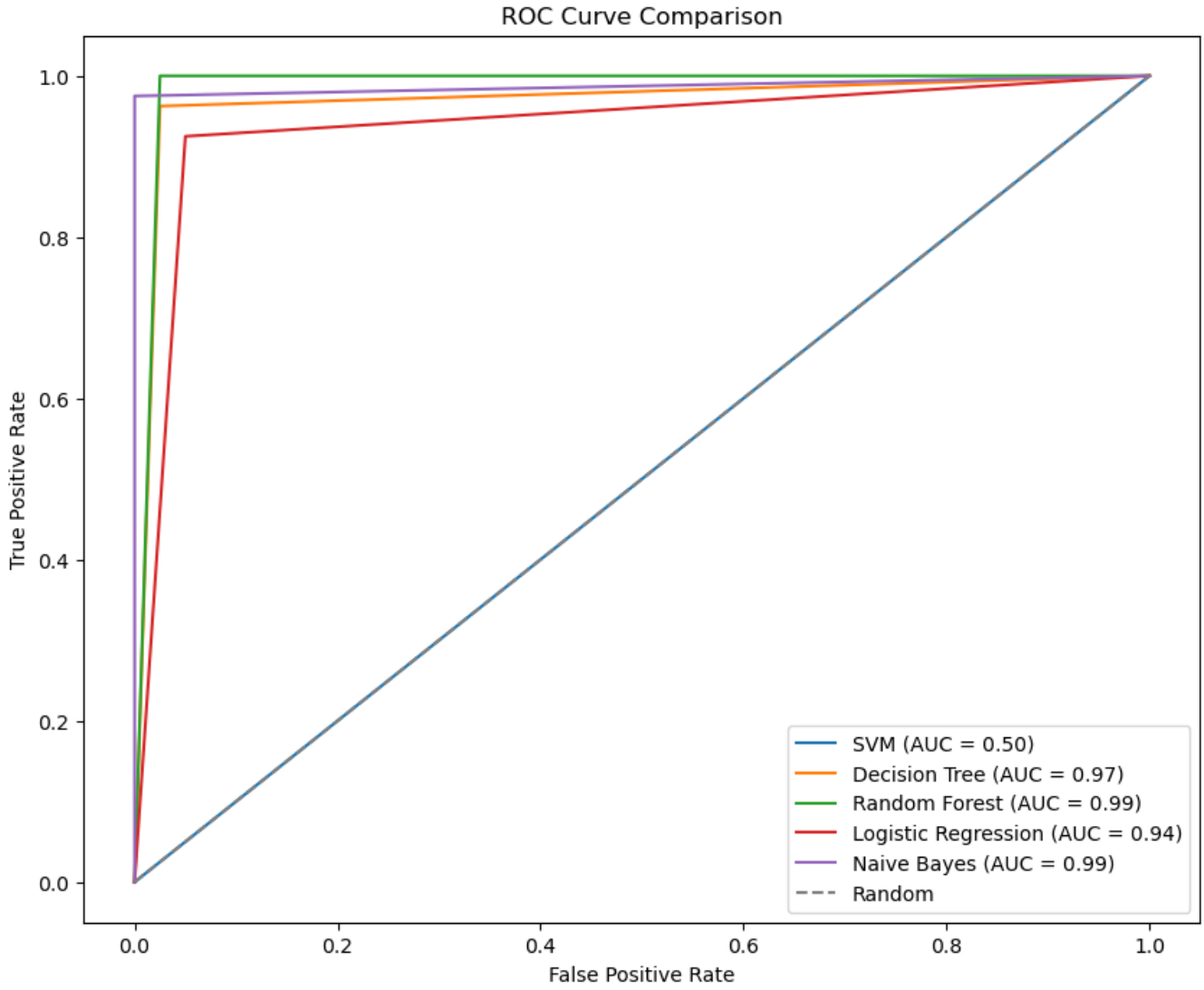
AUC Curve

In [40]:

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Create a DataFrame for AUC
auc_results = []

# SVM
fpr_svm, tpr_svm, _ = roc_curve(y_test_binary, y_pred_svm_binary)
roc_auc_svm = auc(fpr_svm, tpr_svm)
auc_results.append({'Model': 'SVM', 'AUC': roc_auc_svm})

# Decision Tree
fpr_dt, tpr_dt, _ = roc_curve(y_test_binary, y_pred_gini_binary)
roc_auc_dt = auc(fpr_dt, tpr_dt)
auc_results.append({'Model': 'Decision Tree', 'AUC': roc_auc_dt})

# Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_test_binary, y_pred_rf_binary)
roc_auc_rf = auc(fpr_rf, tpr_rf)
auc_results.append({'Model': 'Random Forest', 'AUC': roc_auc_rf})

# Logistic Regression
fpr_lr, tpr_lr, _ = roc_curve(y_test_binary, y_pred_lr_binary)
roc_auc_lr = auc(fpr_lr, tpr_lr)
```

```
auc_results.append({'Model': 'Logistic Regression', 'AUC': roc_auc_lr})

# Naive Bayes
fpr_nb, tpr_nb, _ = roc_curve(y_test_binary, y_pred_nb_binary)
roc_auc_nb = auc(fpr_nb, tpr_nb)
auc_results.append({'Model': 'Naive Bayes', 'AUC': roc_auc_nb})

# Create DataFrame
auc_df = pd.DataFrame(auc_results)
auc_df = auc_df.set_index('Model').round(2)

# Sort DataFrame by AUC in descending order
auc_df = auc_df.sort_values(by='AUC', ascending=False)

# Print the DataFrame
print(auc_df)

# Plot ROC curves
plt.figure(figsize=(10, 8))
plt.plot(fpr_svm, tpr_svm, label=f'SVM (AUC = {roc_auc_svm:.2f})')
plt.plot(fpr_dt, tpr_dt, label=f'Decision Tree (AUC = {roc_auc_dt:.2f})')
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {roc_auc_rf:.2f})')
plt.plot(fpr_lr, tpr_lr, label=f'Logistic Regression (AUC = {roc_auc_lr:.2f})')
plt.plot(fpr_nb, tpr_nb, label=f'Naive Bayes (AUC = {roc_auc_nb:.2f})')

plt.plot([0, 1], [0, 1], linestyle='--', color='grey', label='Random')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend(loc='lower right')
plt.show()
```

```
                     AUC
Model
Random Forest       0.99
Naive Bayes         0.99
Decision Tree       0.97
Logistic Regression 0.94
SVM                 0.50
```


ROC Curve Comparison
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
SVM (AUC = 0.50)
Decision Tree (AUC = 0.97)
Random Forest (AUC = 0.99)
Logistic Regression (AUC = 0.94)
Naive Bayes (AUC = 0.99)
Random

APPENDIX DataPreprocessing

In [1]:
```
import numpy as np
import pandas as pd

# Read dataset file ckd.csv
data = pd.read_csv("ckd.csv",header=0, na_values="?")
data
```

Out[1]:

	Age	Bp	Sg	Al	Su	Rbc	Pc	Pcc	Ba	Bgr	...	Pcv	Wbcc	Rbcc	Htn	
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44.0	7800.0	5.2	yes	
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38.0	6000.0	NaN	no	
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31.0	7500.0	NaN	no	
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...	32.0	6700.0	3.9	yes	
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...	35.0	7300.0	4.6	no	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
395	55.0	80.0	1.020	0.0	0.0	normal	normal	notpresent	notpresent	140.0	...	47.0	6700.0	4.9	no	
396	42.0	70.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	75.0	...	54.0	7800.0	6.2	no	
397	12.0	80.0	1.020	0.0	0.0	normal	normal	notpresent	notpresent	100.0	...	49.0	6600.0	5.4	no	
398	17.0	60.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	114.0	...	51.0	7200.0	5.9	no	
399	58.0	80.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	131.0	...	53.0	6800.0	6.1	no	

400 rows × 25 columns

In [2]:
```
# Replace null values "?" by numpy.NaN
data.replace("?", np.NaN)
```

Out[2]:

	Age	Bp	Sg	Al	Su	Rbc	Pc	Pcc	Ba	Bgr	...	Pcv	Wbcc	Rbcc	Htn	
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44.0	7800.0	5.2	yes	
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38.0	6000.0	NaN	no	
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31.0	7500.0	NaN	no	
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...	32.0	6700.0	3.9	yes	
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...	35.0	7300.0	4.6	no	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
395	55.0	80.0	1.020	0.0	0.0	normal	normal	notpresent	notpresent	140.0	...	47.0	6700.0	4.9	no	
396	42.0	70.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	75.0	...	54.0	7800.0	6.2	no	
397	12.0	80.0	1.020	0.0	0.0	normal	normal	notpresent	notpresent	100.0	...	49.0	6600.0	5.4	no	
398	17.0	60.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	114.0	...	51.0	7200.0	5.9	no	
399	58.0	80.0	1.025	0.0	0.0	normal	normal	notpresent	notpresent	131.0	...	53.0	6800.0	6.1	no	

400 rows × 25 columns

In [3]:
```
# Mapping of nominal values to binary values
nominal_to_binary_mapping = {
    "Rbc": {"normal": 1, "abnormal": 0},
    "Pc": {"normal": 1, "abnormal": 0},
    "Pcc": {"present": 1, "notpresent": 0},
    "Ba": {"present": 1, "notpresent": 0},
    "Htn": {"yes": 1, "no": 0},
    "Dm": {"yes": 1, "no": 0},
    "Cad": {"yes": 1, "no": 0},
    "Appet": {"good": 1, "poor": 0},
    "pe": {"yes": 1, "no": 0},
    "Ane": {"yes": 1, "no": 0}
}

# Replace nominal values with binary values in the dataset
data.replace(nominal_to_binary_mapping, inplace=True)
```

In [4]:
```
data
```

Out[4]:

	Age	Bp	Sg	Al	Su	Rbc	Pc	Pcc	Ba	Bgr	...	Pcv	Wbcc	Rbcc	Htn	Dm	Cad	Appet	pe
0	48.0	80.0	1.020	1.0	0.0	NaN	1.0	0.0	0.0	121.0	...	44.0	7800.0	5.2	1.0	1	0.0	1.0	0.0
1	7.0	50.0	1.020	4.0	0.0	NaN	1.0	0.0	0.0	NaN	...	38.0	6000.0	NaN	0.0	0	0.0	1.0	0.0
2	62.0	80.0	1.010	2.0	3.0	1.0	1.0	0.0	0.0	423.0	...	31.0	7500.0	NaN	0.0	1	0.0	0.0	0.0
3	48.0	70.0	1.005	4.0	0.0	1.0	0.0	1.0	0.0	117.0	...	32.0	6700.0	3.9	1.0	0	0.0	0.0	1.0
4	51.0	80.0	1.010	2.0	0.0	1.0	1.0	0.0	0.0	106.0	...	35.0	7300.0	4.6	0.0	0	0.0	1.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
395	55.0	80.0	1.020	0.0	0.0	1.0	1.0	0.0	0.0	140.0	...	47.0	6700.0	4.9	0.0	0	0.0	1.0	0.0
396	42.0	70.0	1.025	0.0	0.0	1.0	1.0	0.0	0.0	75.0	...	54.0	7800.0	6.2	0.0	0	0.0	1.0	0.0
397	12.0	80.0	1.020	0.0	0.0	1.0	1.0	0.0	0.0	100.0	...	49.0	6600.0	5.4	0.0	0	0.0	1.0	0.0
398	17.0	60.0	1.025	0.0	0.0	1.0	1.0	0.0	0.0	114.0	...	51.0	7200.0	5.9	0.0	0	0.0	1.0	0.0
399	58.0	80.0	1.025	0.0	0.0	1.0	1.0	0.0	0.0	131.0	...	53.0	6800.0	6.1	0.0	0	0.0	1.0	0.0

400 rows × 25 columns

In [6]:
```
# Fill null values with mean value of the respective column

data.fillna(round(data.mean(),2), inplace=True)
data
```

```
C:\Users\acer\AppData\Local\Temp\ipykernel_5712\3608136598.py:3: FutureWarning: Dropping of n
uisance columns in DataFrame reductions (with 'numeric_only=None') is deprecated; in a future
version this will raise TypeError.  Select only valid columns before calling the reduction.
  data.fillna(round(data.mean(),2), inplace=True)
```

Out[6]:

	Age	Bp	Sg	Al	Su	Rbc	Pc	Pcc	Ba	Bgr	...	Pcv	Wbcc	Rbcc	Htn	Dm	Cad	Appet	pe
0	48.0	80.0	1.020	1.0	0.0	0.81	1.0	0.0	0.0	121.00	...	44.0	7800.0	5.20	1.0	1	0.0	1.0	0.0
1	7.0	50.0	1.020	4.0	0.0	0.81	1.0	0.0	0.0	148.04	...	38.0	6000.0	4.71	0.0	0	0.0	1.0	0.0
2	62.0	80.0	1.010	2.0	3.0	1.00	1.0	0.0	0.0	423.00	...	31.0	7500.0	4.71	0.0	1	0.0	0.0	0.0
3	48.0	70.0	1.005	4.0	0.0	1.00	0.0	1.0	0.0	117.00	...	32.0	6700.0	3.90	1.0	0	0.0	0.0	1.0
4	51.0	80.0	1.010	2.0	0.0	1.00	1.0	0.0	0.0	106.00	...	35.0	7300.0	4.60	0.0	0	0.0	1.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
395	55.0	80.0	1.020	0.0	0.0	1.00	1.0	0.0	0.0	140.00	...	47.0	6700.0	4.90	0.0	0	0.0	1.0	0.0
396	42.0	70.0	1.025	0.0	0.0	1.00	1.0	0.0	0.0	75.00	...	54.0	7800.0	6.20	0.0	0	0.0	1.0	0.0
397	12.0	80.0	1.020	0.0	0.0	1.00	1.0	0.0	0.0	100.00	...	49.0	6600.0	5.40	0.0	0	0.0	1.0	0.0
398	17.0	60.0	1.025	0.0	0.0	1.00	1.0	0.0	0.0	114.00	...	51.0	7200.0	5.90	0.0	0	0.0	1.0	0.0
399	58.0	80.0	1.025	0.0	0.0	1.00	1.0	0.0	0.0	131.00	...	53.0	6800.0	6.10	0.0	0	0.0	1.0	0.0

400 rows × 25 columns

In [7]:

```
# Save this dataset as cleaned_data.csv for further prediction
data.to_csv("cleaned_data.csv", sep=',', index=False)
```

APPENDIX_GROUP B_CKD PREDICTION

DM Project Full Coding

In [1]:
```
# Importing the required packages
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report, roc_curv
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import seaborn as sns
import tkinter as tk
from tkinter import ttk
from ttkthemes import ThemedStyle
import joblib
```

In [2]:
```
# Data Exploration
balance_data = pd.read_csv('cleaned_data.csv', sep=',', header=0)
print("Dataset Length: ", len(balance_data))
print("Dataset Shape: ", balance_data.shape)
print("Data Exploration:")
print(balance_data.head())
print(balance_data.describe())
```

```
Dataset Length:  400
Dataset Shape:  (400, 25)
Data Exploration:
    Age    Bp     Sg   Al   Su   Rbc   Pc  Pcc   Ba     Bgr  ...   Pcv  \
0  48.0  80.0  1.020  1.0  0.0  0.81  1.0  0.0  0.0  121.00  ...  44.0
1   7.0  50.0  1.020  4.0  0.0  0.81  1.0  0.0  0.0  148.04  ...  38.0
2  62.0  80.0  1.010  2.0  3.0  1.00  1.0  0.0  0.0  423.00  ...  31.0
3  48.0  70.0  1.005  4.0  0.0  1.00  0.0  1.0  0.0  117.00  ...  32.0
4  51.0  80.0  1.010  2.0  0.0  1.00  1.0  0.0  0.0  106.00  ...  35.0

     Wbcc  Rbcc  Htn  Dm  Cad  Appet   pe  Ane  Class
0  7800.0  5.20  1.0   1  0.0    1.0  0.0  0.0    ckd
1  6000.0  4.71  0.0   0  0.0    1.0  0.0  0.0    ckd
2  7500.0  4.71  0.0   1  0.0    0.0  0.0  1.0    ckd
3  6700.0  3.90  1.0   0  0.0    0.0  1.0  1.0    ckd
4  7300.0  4.60  0.0   0  0.0    1.0  0.0  0.0    ckd

[5 rows x 25 columns]
              Age          Bp          Sg          Al          Su          Rbc  \
count  400.000000  400.000000  400.000000  400.000000  400.000000  400.000000
mean    51.483300   76.469100    1.017712    1.017300    0.450125    0.810300
std     16.974966   13.476298    0.005434    1.272318    1.029487    0.308983
min      2.000000   50.000000    1.005000    0.000000    0.000000    0.000000
25%     42.000000   70.000000    1.015000    0.000000    0.000000    0.810000
50%     54.000000   78.235000    1.020000    1.000000    0.000000    1.000000
75%     64.000000   80.000000    1.020000    2.000000    0.450000    1.000000
max     90.000000  180.000000    1.025000    5.000000    5.000000    1.000000

               Pc         Pcc          Ba         Bgr  ...        Hemo  \
count  400.000000  400.000000  400.000000  400.000000  ...  400.000000
mean     0.772625    0.106100    0.055600  148.036900  ...   12.526900
std      0.383751    0.306756    0.228199   74.782634  ...    2.716171
min      0.000000    0.000000    0.000000   22.000000  ...    3.100000
25%      0.770000    0.000000    0.000000  101.000000  ...   10.875000
50%      1.000000    0.000000    0.000000  126.000000  ...   12.530000
75%      1.000000    0.000000    0.000000  150.000000  ...   14.625000
max      1.000000    1.000000    1.000000  490.000000  ...   17.800000

              Pcv          Wbcc        Rbcc         Htn         Dm  \
count  400.000000    400.000000  400.000000  400.000000  400.00000
mean    38.883700   8406.121800    4.708275    0.369350    0.34250
std      8.151082   2523.219976    0.840315    0.482023    0.47514
min      9.000000   2200.000000    2.100000    0.000000    0.00000
25%     34.000000   6975.000000    4.500000    0.000000    0.00000
50%     38.880000   8406.120000    4.710000    0.000000    0.00000
75%     44.000000   9400.000000    5.100000    1.000000    1.00000
max     54.000000  26400.000000    8.000000    1.000000    1.00000

              Cad       Appet          pe         Ane
count  400.000000  400.000000  400.000000  400.000000
mean     0.085450    0.794475    0.190475    0.150375
std      0.279166    0.404077    0.392677    0.357440
min      0.000000    0.000000    0.000000    0.000000
25%      0.000000    1.000000    0.000000    0.000000
50%      0.000000    1.000000    0.000000    0.000000
75%      0.000000    1.000000    0.000000    0.000000
max      1.000000    1.000000    1.000000    1.000000

[8 rows x 24 columns]
```

In [3]:
```
# Building Phase
# Separating the target variable
X = balance_data.values[:, 0:24]
Y = balance_data.values[:, -1]
```

In [4]:
```
# Splitting the dataset into train and test
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=100)
```

Decision Tree

In [5]:
```
# Hyperparameter Tuning for decision tree
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 5, 7],
    'min_samples_leaf': [3, 5, 7]
}

clf = DecisionTreeClassifier(random_state=100)
grid_search = GridSearchCV(estimator=clf, param_grid=param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
print("Best Hyperparameters:", grid_search.best_params_)
tuned_clf = grid_search.best_estimator_
```

```
Best Hyperparameters: {'criterion': 'gini', 'max_depth': 3, 'min_samples_leaf': 7}
```

In [6]:
```
# Training phase
clf_gini = DecisionTreeClassifier(criterion="gini", random_state=100, max_depth=3, min_sample
clf_gini.fit(X_train, y_train)
```

Out[6]:

▾ DecisionTreeClassifier

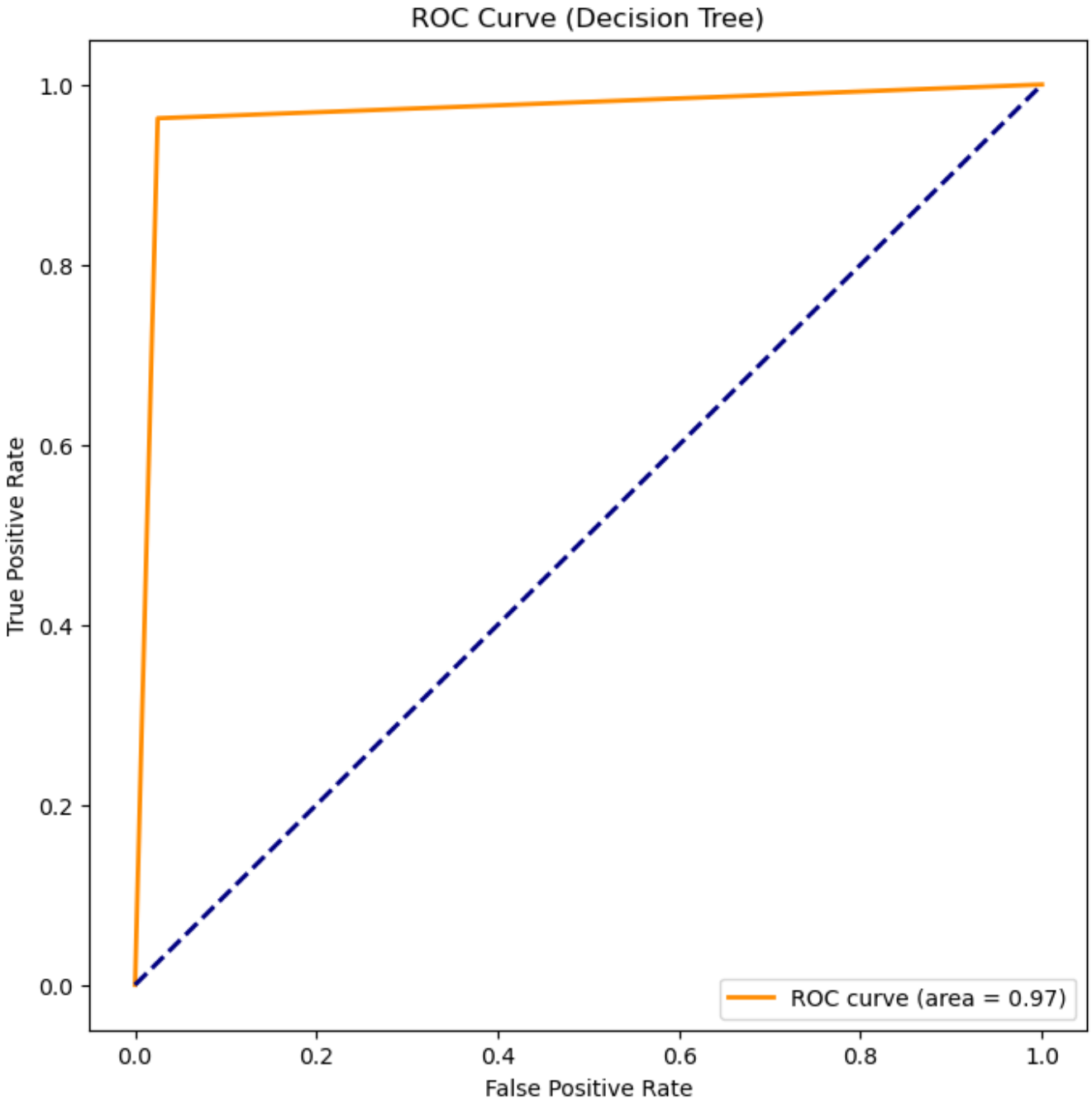
```
DecisionTreeClassifier(max_depth=3, min_samples_leaf=7, random_state=100)
```

In [7]:
```
# Operational Phase
print("\nResults Using Gini Index:")
y_pred_gini = clf_gini.predict(X_test)
y_test_binary = np.where(y_test == 'ckd', 1, 0)
y_pred_gini_binary = np.where(y_pred_gini == 'ckd', 1, 0)
# Metrics for Decision Tree
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_gini))
print("Accuracy:", accuracy_score(y_test,y_pred_gini) * 100)
print("Report:\n", classification_report(y_test, y_pred_gini))
print("F1 Score:", f1_score(y_test_binary, y_pred_gini_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_gini_binary))
```

```
Results Using Gini Index:
Confusion Matrix:
 [[77  3]
 [ 1 39]]
Accuracy: 96.66666666666667
Report:
               precision    recall  f1-score   support

         ckd       0.99      0.96      0.97        80
      notckd       0.93      0.97      0.95        40

    accuracy                           0.97       120
   macro avg       0.96      0.97      0.96       120
weighted avg       0.97      0.97      0.97       120

F1 Score: 0.9746835443037976
AUC Score: 0.96875
```

In [8]:
```
# ROC Curve for Decision Tree
fpr_dt, tpr_dt, _ = roc_curve(y_test_binary,y_pred_gini_binary)
roc_auc_dt = auc(fpr_dt, tpr_dt)

plt.figure(figsize=(8, 8))
plt.plot(fpr_dt, tpr_dt, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(r
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Decision Tree)')
```

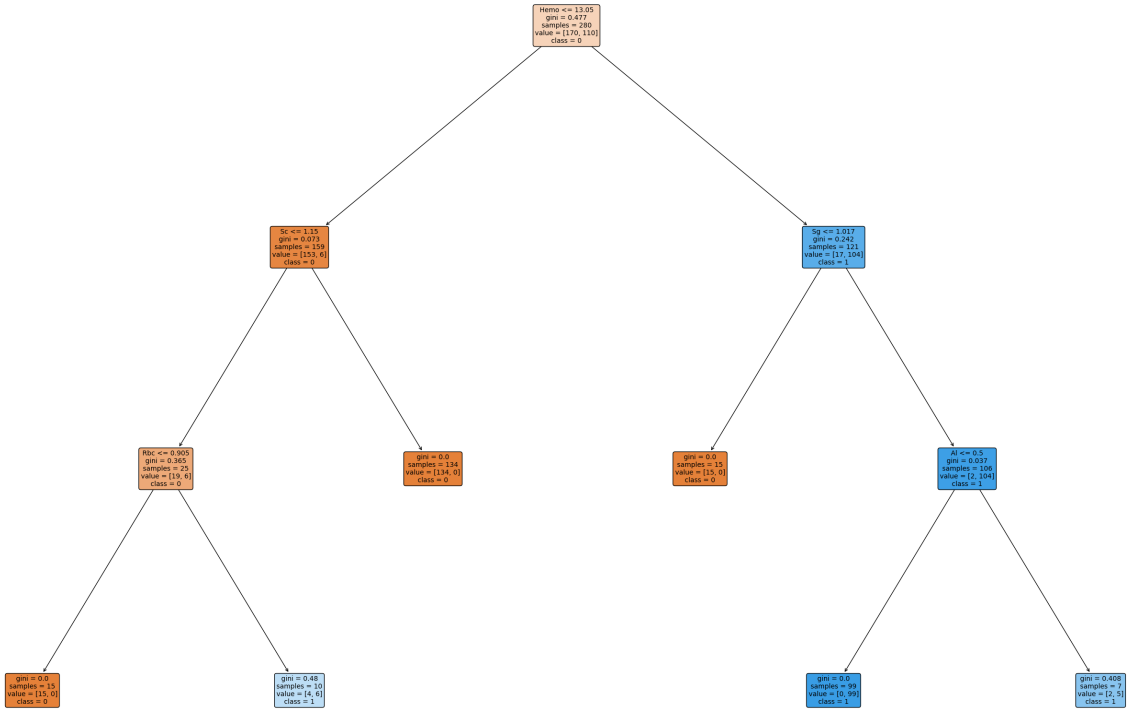
```
plt.legend(loc="lower right")
plt.show()
```



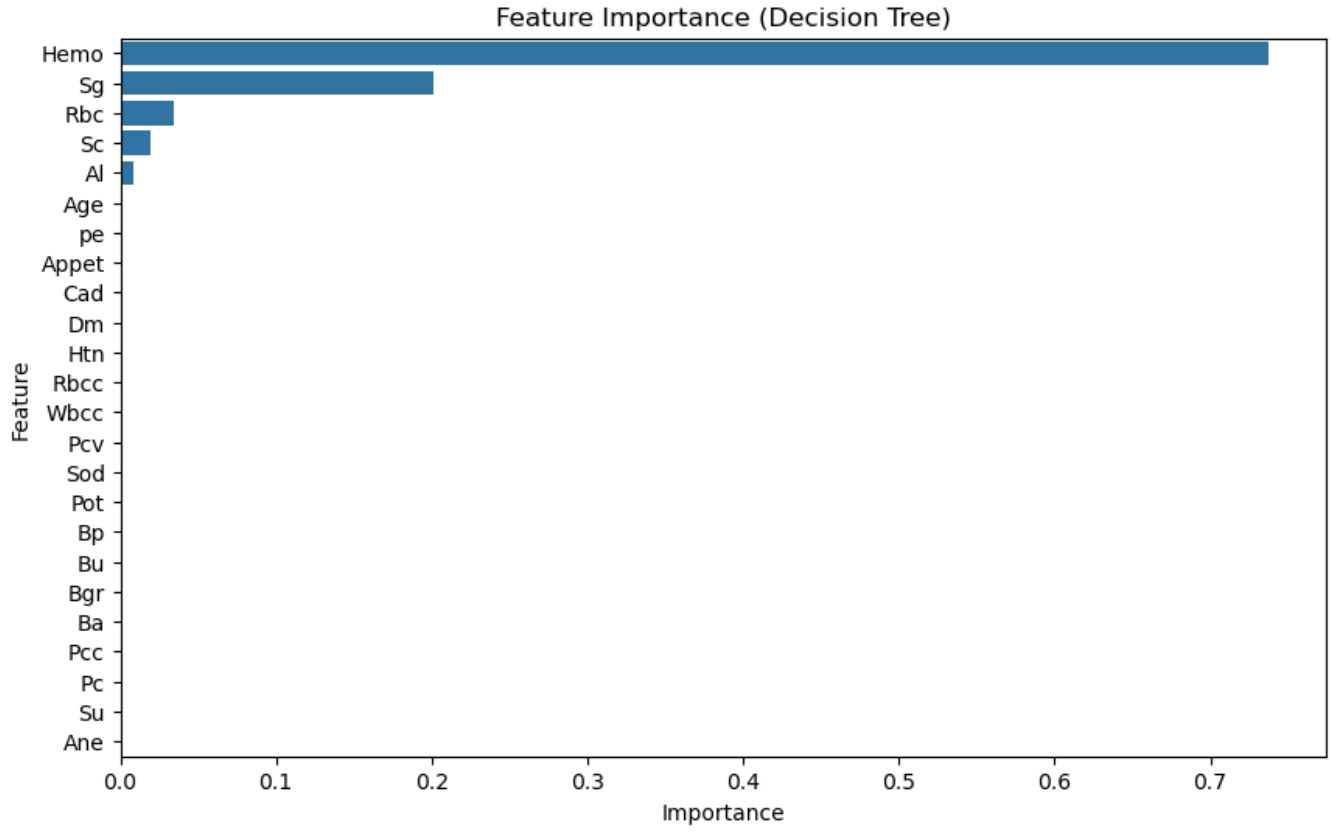
In [9]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Decision Tree
mse_dt = mean_squared_error(y_test_binary,y_pred_gini_binary)
rmse_dt = np.sqrt(mse_dt)
print("Mean Squared Error (MSE):", mse_dt)
print("Root Mean Squared Error (RMSE):", rmse_dt)
```

```
Mean Squared Error (MSE): 0.03333333333333333
Root Mean Squared Error (RMSE): 0.18257418583505536
```

In [10]:
```
# Visualizing Decision Trees using plot_tree
plt.figure(figsize=(36, 24))
plot_tree(clf_gini, filled=True, feature_names=balance_data.columns[:-1], class_names=["0", "
plt.show()
```



In [11]:
```
# Feature Importance for Decision Tree
feature_importance_dt = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Importance': clf
feature_importance_dt = feature_importance_dt.sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance_dt)
plt.title('Feature Importance (Decision Tree)')
plt.show()
```



In [12]:
```
# Cross-Validation for Decision Tree
cv_results_dt = cross_val_score(clf_gini, X, Y, cv=5, scoring='accuracy')
print("Cross-Validation Results (Decision Tree):", cv_results_dt)
print("Mean Accuracy (Decision Tree):", np.mean(cv_results_dt))
```

```
Cross-Validation Results (Decision Tree): [0.95   1.     0.9375 0.975  0.9875]
Mean Accuracy (Decision Tree): 0.9700000000000001
```

Random Forest

In [13]:
```python
# Hyperparameter Tuning for Random Forest
param_grid_rf = {
    'n_estimators': [50, 100, 150],
    'max_depth': [3, 5, 7],
    'min_samples_leaf': [3, 5, 7],
    'max_features': ['sqrt', 'log2']
}

clf_rf = RandomForestClassifier(random_state=100)
grid_search_rf = GridSearchCV(estimator=clf_rf, param_grid=param_grid_rf, cv=5, scoring='accu
grid_search_rf.fit(X_train, y_train)
print("Best Hyperparameters (Random Forest):", grid_search_rf.best_params_)
tuned_clf_rf = grid_search_rf.best_estimator_
```

```
Best Hyperparameters (Random Forest): {'max_depth': 3, 'max_features': 'sqrt', 'min_samples_l
eaf': 3, 'n_estimators': 50}
```

In [14]:
```python
# Training phase for Random Forest
clf_rf = RandomForestClassifier(n_estimators=50, max_depth=3, min_samples_leaf=3, max_feature
clf_rf.fit(X_train, y_train)
```

Out[14]:

RandomForestClassifier

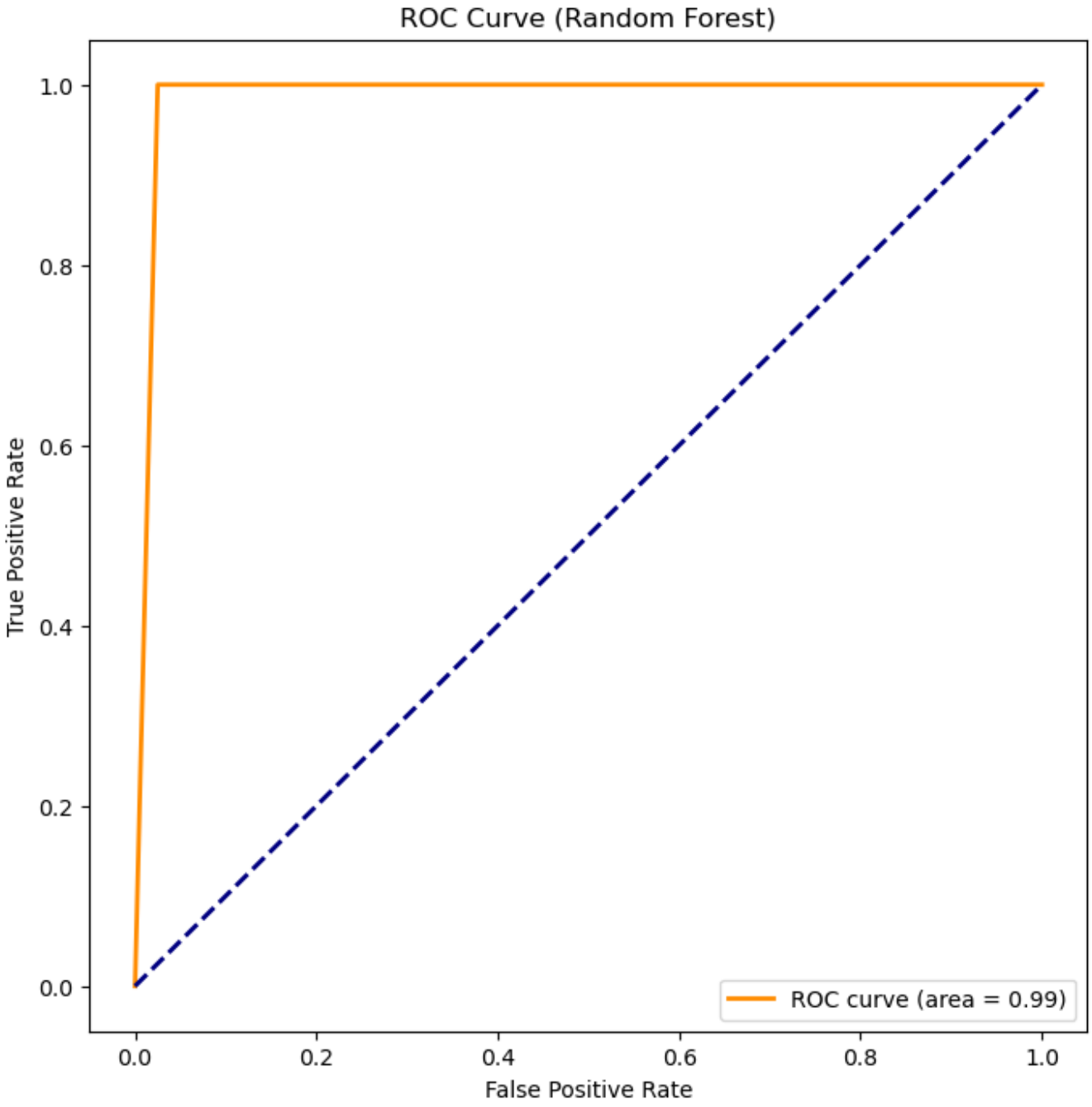
```
RandomForestClassifier(max_depth=3, min_samples_leaf=3, n_estimators=50,
                       random_state=100)
```

In [15]:
```python
# Operational Phase for Random Forest
print("\nResults Using Random Forest:")
y_pred_rf = clf_rf.predict(X_test)
y_test_binary = np.where(y_test == 'ckd', 1, 0)
y_pred_rf_binary = np.where(y_pred_rf == 'ckd', 1, 0)
# Metrics for Random Forest
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
print("Accuracy:", accuracy_score(y_test, y_pred_rf) * 100)
print("Report:\n", classification_report(y_test, y_pred_rf))
print("F1 Score:", f1_score(y_test_binary, y_pred_rf_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_rf_binary))
```

```
Results Using Random Forest:
Confusion Matrix:
 [[80  0]
 [ 1 39]]
Accuracy: 99.16666666666667
Report:
               precision    recall  f1-score   support

         ckd       0.99      1.00      0.99        80
      notckd       1.00      0.97      0.99        40

    accuracy                           0.99       120
   macro avg       0.99      0.99      0.99       120
weighted avg       0.99      0.99      0.99       120

F1 Score: 0.9937888198757764
AUC Score: 0.9874999999999999
```

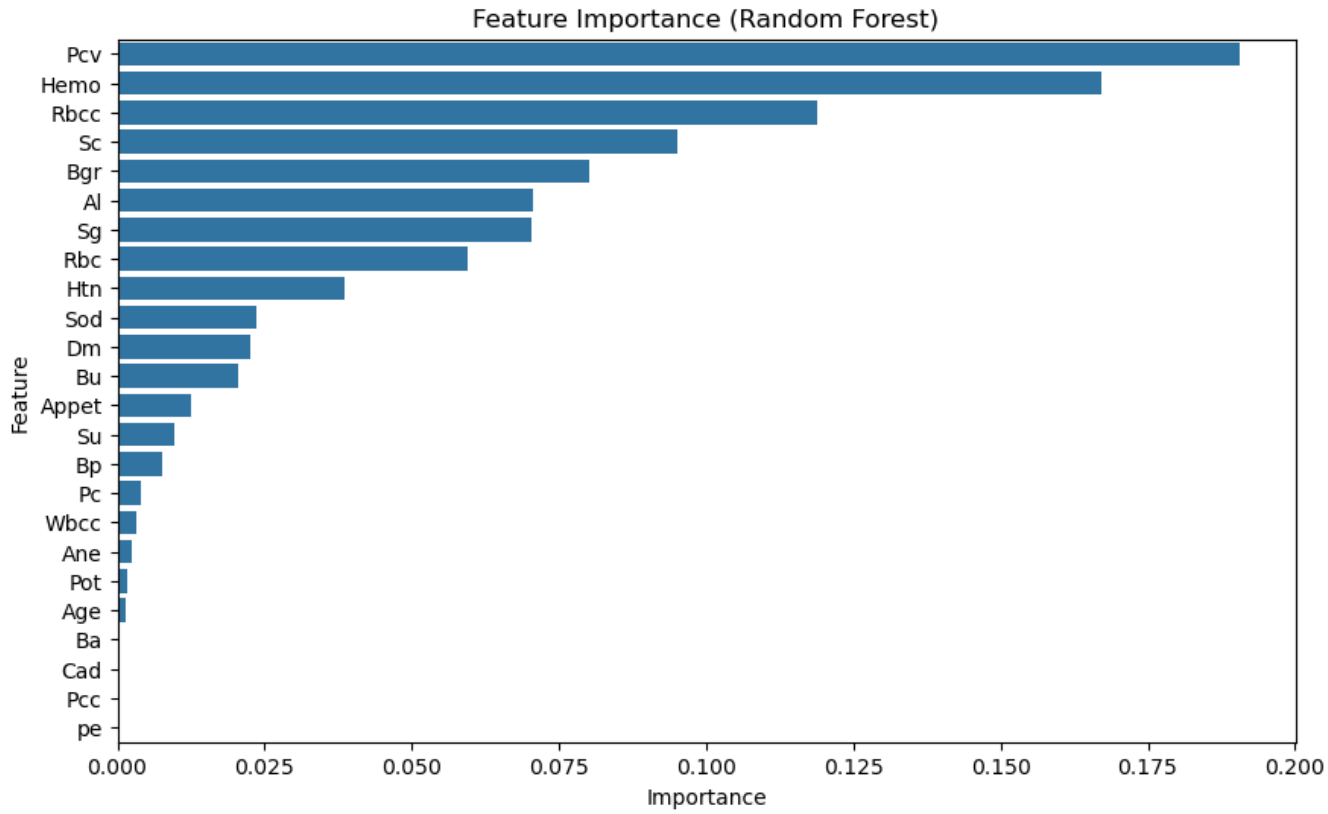
In [16]:
```python
# ROC Curve for Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_test_binary, y_pred_rf_binary)
roc_auc_rf = auc(fpr_rf, tpr_rf)

plt.figure(figsize=(8, 8))
```

```
plt.plot(fpr_rf, tpr_rf, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(r
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Random Forest)')
plt.legend(loc="lower right")
plt.show()
```



In [17]:

```
# Feature Importance for Random Forest
feature_importance_rf = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Importance': clf
feature_importance_rf = feature_importance_rf.sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance_rf)
plt.title('Feature Importance (Random Forest)')
plt.show()
```



In [18]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Random Forest
mse_rf = mean_squared_error(y_test_binary, y_pred_rf_binary)
rmse_rf = np.sqrt(mse_rf)
print("Mean Squared Error (MSE):", mse_rf)
print("Root Mean Squared Error (RMSE):", rmse_rf)
```

```
Mean Squared Error (MSE): 0.008333333333333333
Root Mean Squared Error (RMSE): 0.09128709291752768
```

In [19]:
```
# Cross-Validation for Random Forest
cv_results_rf = cross_val_score(clf_rf, X, Y, cv=5, scoring='accuracy')
print("Cross-Validation Results (Random Forest):", cv_results_rf)
print("Mean Accuracy (Random Forest):", np.mean(cv_results_rf))
```

```
Cross-Validation Results (Random Forest): [0.9875 1.     0.9625 1.     1.    ]
Mean Accuracy (Random Forest): 0.99
```

Support Vector Machine

In [20]:
```
# Training phase for Support Vector Machine (SVM)
clf_svm = SVC(kernel='rbf', random_state=100)
clf_svm.fit(X_train, y_train)
```

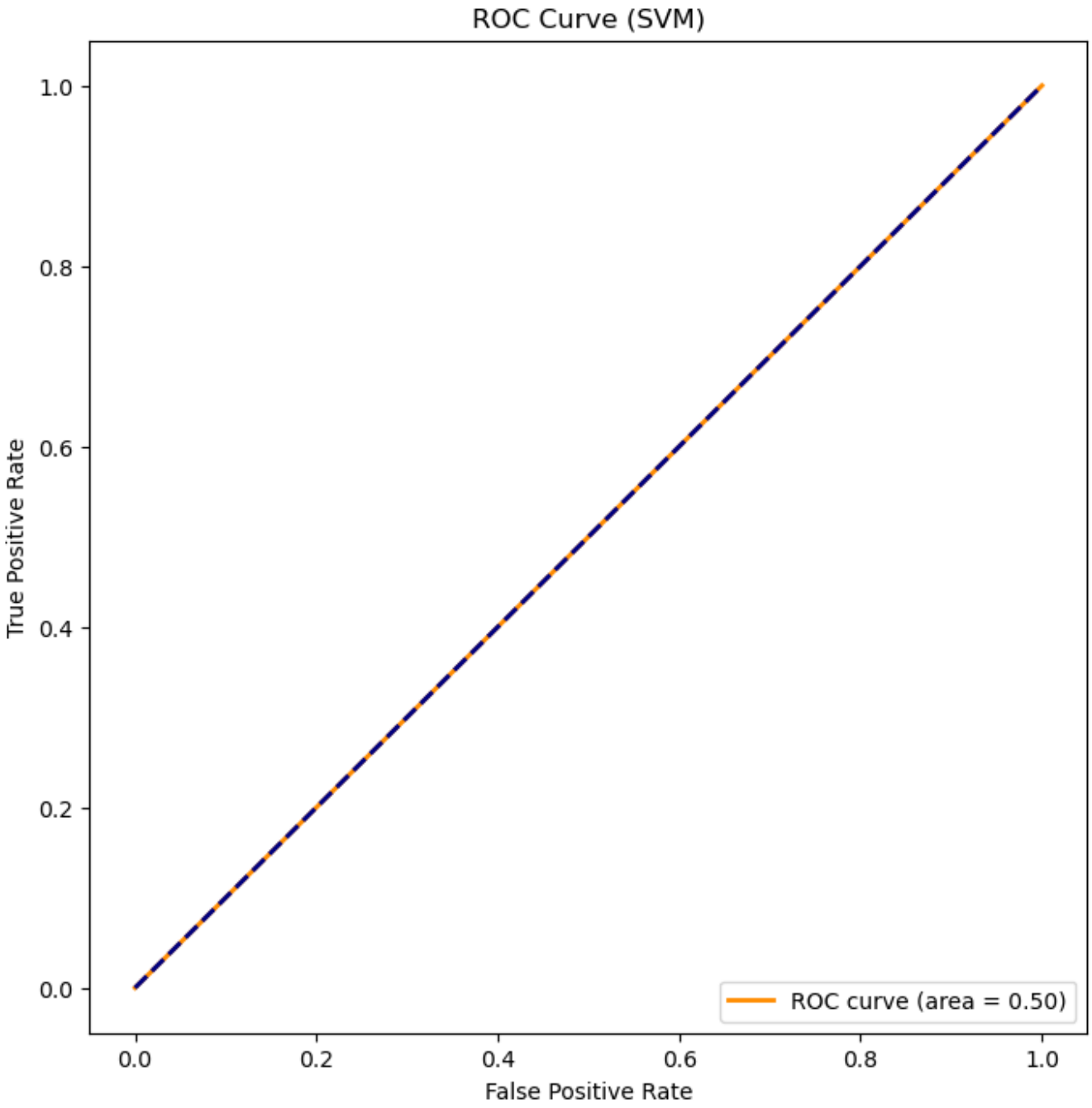
Out[20]:
```
SVC
SVC(random_state=100)
```

In [21]:
```
# Operational Phase for SVM
print("\nResults Using SVM:")
y_pred_svm = clf_svm.predict(X_test)
y_test_binary = np.where(y_test == 'ckd', 1, 0)
y_pred_svm_binary = np.where(y_pred_svm == 'ckd', 1, 0)
# Metrics for SVM
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_svm))
print("Accuracy:", accuracy_score(y_test, y_pred_svm) * 100)
print("Report:\n", classification_report(y_test, y_pred_svm, zero_division=1))
print("F1 Score:", f1_score(y_test_binary, y_pred_svm_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_svm_binary))
```

```
Results Using SVM:
Confusion Matrix:
 [[80  0]
 [40  0]]
Accuracy: 66.66666666666666
Report:
              precision    recall  f1-score   support

         ckd       0.67      1.00      0.80        80
      notckd       1.00      0.00      0.00        40

    accuracy                           0.67       120
   macro avg       0.83      0.50      0.40       120
weighted avg       0.78      0.67      0.53       120

F1 Score: 0.8
AUC Score: 0.5
```

In [22]:

```
# ROC Curve for SVM
fpr_svm, tpr_svm, _ = roc_curve(y_test_binary, y_pred_svm_binary)
roc_auc_svm = auc(fpr_svm, tpr_svm)

plt.figure(figsize=(8, 8))
plt.plot(fpr_svm, tpr_svm, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (SVM)')
plt.legend(loc="lower right")
plt.show()
```



In [23]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for SVM
mse_svm = mean_squared_error(y_test_binary, y_pred_svm_binary)
rmse_svm = np.sqrt(mse_svm)
print("Mean Squared Error (MSE) for SVM:", mse_svm)
print("Root Mean Squared Error (RMSE) for SVM:", rmse_svm)
```

```
Mean Squared Error (MSE) for SVM: 0.3333333333333333
Root Mean Squared Error (RMSE) for SVM: 0.5773502691896257
```

In [24]:
```
# Cross-Validation for SVM
cv_results_svm = cross_val_score(clf_svm, X, Y, cv=5, scoring='accuracy')
print("Cross-Validation Results (SVM):", cv_results_svm)
print("Mean Accuracy (SVM):", np.mean(cv_results_svm))
```

```
Cross-Validation Results (SVM): [0.625 0.625 0.625 0.625 0.625]
Mean Accuracy (SVM): 0.625
```

Logistic Regression

In [25]:
```
# Hyperparameter Tuning for Logistic Regression
param_grid_lr = {
    'penalty': ['l1', 'l2'],
    'C': [0.001, 0.01, 0.1, 1, 10, 100]
}
```

```
clf_lr = LogisticRegression(random_state=100, solver='liblinear')
grid_search_lr = GridSearchCV(estimator=clf_lr, param_grid=param_grid_lr, cv=5, scoring='accu
grid_search_lr.fit(X_train, y_train)
print("Best Hyperparameters (Logistic Regression):", grid_search_lr.best_params_)
tuned_clf_lr = grid_search_lr.best_estimator_
```

```
Best Hyperparameters (Logistic Regression): {'C': 1, 'penalty': 'l2'}
```

In [26]:
```
# Training phase for Logistic Regression
clf_lr = LogisticRegression(C=1, penalty='l2', max_iter=100, random_state=100)
clf_lr.fit(X_train, y_train)
```

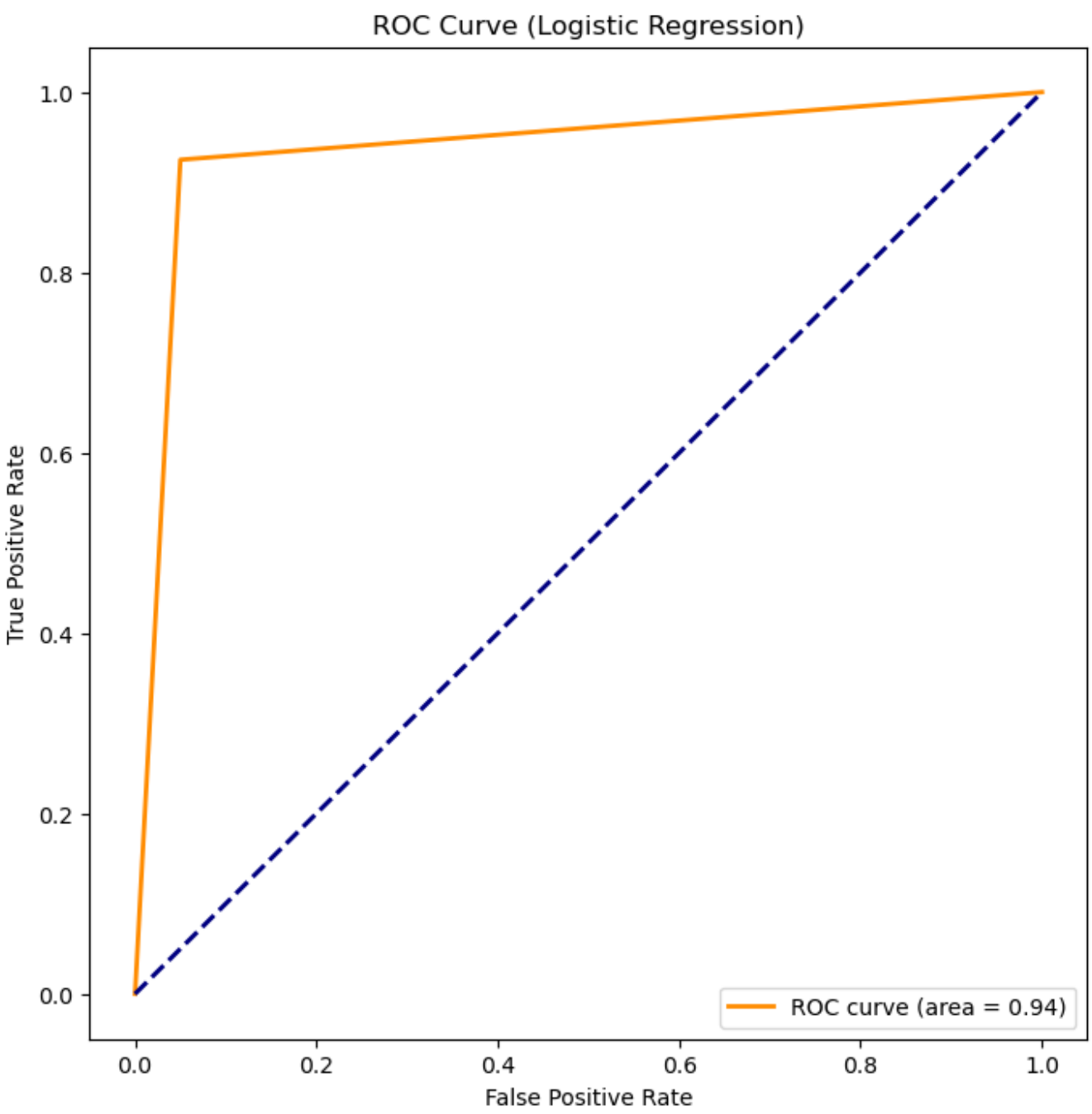
```
C:\Users\PC\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:460: ConvergenceWar
ning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[26]:
```
LogisticRegression
LogisticRegression(C=1, random_state=100)
```

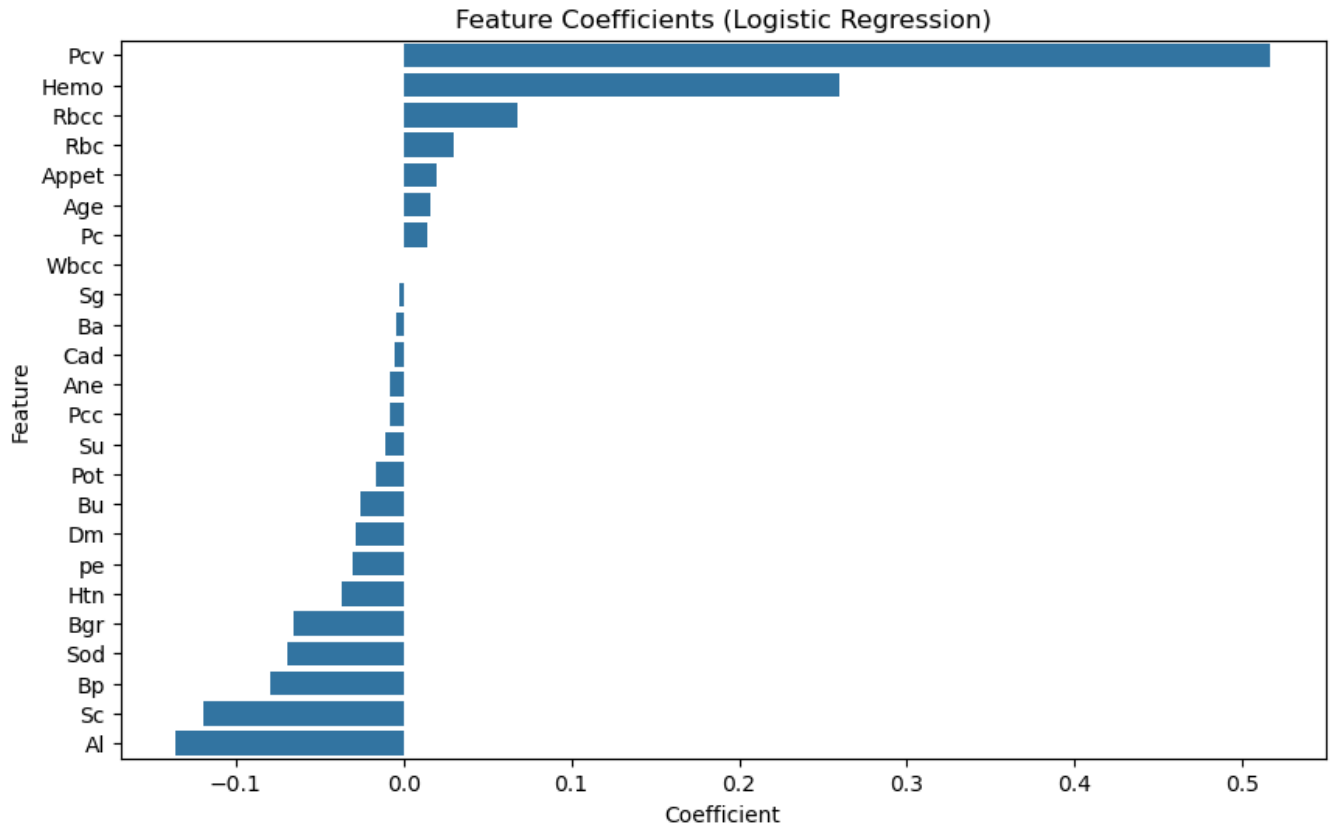
In [27]:
```
# Operational Phase for Logistic Regression
print("\nResults Using Logistic Regression:")
y_pred_lr = clf_lr.predict(X_test)
y_pred_lr_binary = np.where(y_pred_lr == 'ckd', 1, 0)
# Metrics for Logistic Regression
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_lr))
print("Accuracy:", accuracy_score(y_test, y_pred_lr) * 100)
print("Report:\n", classification_report(y_test, y_pred_lr))
print("F1 Score:", f1_score(y_test_binary, y_pred_lr_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_lr_binary))
```

```
Results Using Logistic Regression:
Confusion Matrix:
 [[74  6]
 [ 2 38]]
Accuracy: 93.33333333333333
Report:
               precision    recall  f1-score   support

         ckd       0.97      0.93      0.95        80
      notckd       0.86      0.95      0.90        40

    accuracy                           0.93       120
   macro avg       0.92      0.94      0.93       120
weighted avg       0.94      0.93      0.93       120

F1 Score: 0.9487179487179489
AUC Score: 0.9374999999999999
```

In [28]:
```
# ROC Curve for Logistic Regression
fpr_lr, tpr_lr, _ = roc_curve(y_test_binary, y_pred_lr_binary)
roc_auc_lr = auc(fpr_lr, tpr_lr)

plt.figure(figsize=(8, 8))
plt.plot(fpr_lr, tpr_lr, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(r
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Logistic Regression)')
plt.legend(loc="lower right")
plt.show()
```



In [29]:
```
# Feature Coefficients for Logistic Regression
feature_coefficients_lr = pd.DataFrame({'Feature': balance_data.columns[:-1], 'Coefficient':
feature_coefficients_lr = feature_coefficients_lr.sort_values(by='Coefficient', ascending=Fal

plt.figure(figsize=(10, 6))
sns.barplot(x='Coefficient', y='Feature', data=feature_coefficients_lr)
plt.title('Feature Coefficients (Logistic Regression)')
plt.show()
```



In [30]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Logistic Regression
mse_lr = mean_squared_error(y_test_binary, y_pred_lr_binary)
rmse_lr = np.sqrt(mse_lr)
print("Mean Squared Error (MSE):", mse_lr)
print("Root Mean Squared Error (RMSE):", rmse_lr)
```

```
Mean Squared Error (MSE): 0.06666666666666667
Root Mean Squared Error (RMSE): 0.2581988897471611
```

Naive Bayes

In [31]:
```
# Naive Bayes - Gaussian Naive Bayes does not have many hyperparameters to tune
clf_nb = GaussianNB()
clf_nb.fit(X_train, y_train)
```

Out[31]:
```
▾ GaussianNB
GaussianNB()
```

In [32]:
```
# Operational Phase for Naive Bayes
print("\nResults Using Naive Bayes:")
y_pred_nb = clf_nb.predict(X_test)
y_pred_nb_binary = np.where(y_pred_nb == 'ckd', 1, 0)
```

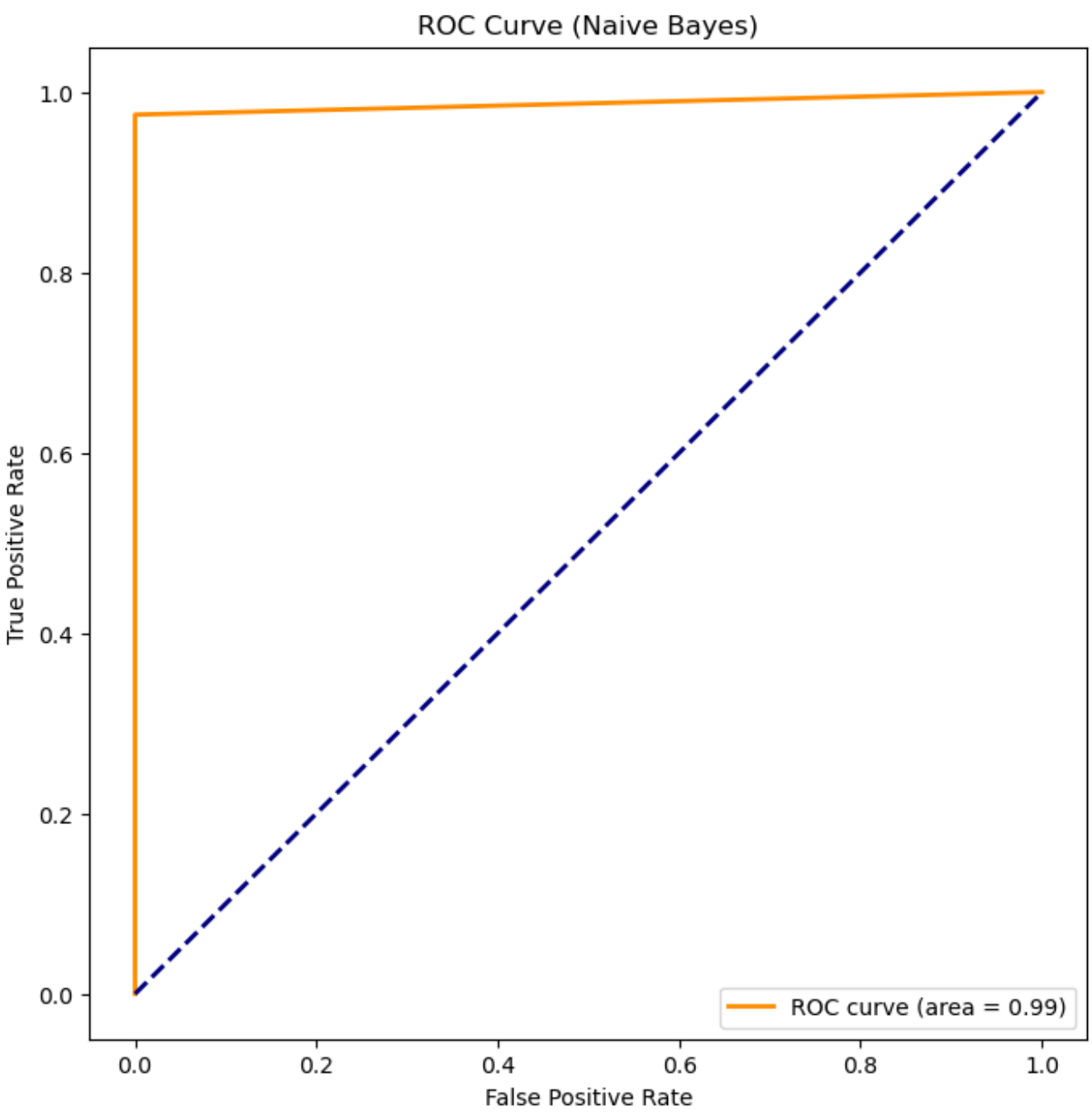
```
Results Using Naive Bayes:
```

In [33]:
```
# Metrics for Naive Bayes
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_nb))
print("Accuracy:", accuracy_score(y_test, y_pred_nb) * 100)
print("Report:\n", classification_report(y_test, y_pred_nb))
print("F1 Score:", f1_score(y_test_binary, y_pred_nb_binary))
print("AUC Score:", roc_auc_score(y_test_binary, y_pred_nb_binary))
```

```
Confusion Matrix:
 [[78  2]
 [ 0 40]]
Accuracy: 98.33333333333333
Report:
              precision    recall  f1-score   support

         ckd       1.00      0.97      0.99        80
      notckd       0.95      1.00      0.98        40

    accuracy                           0.98       120
   macro avg       0.98      0.99      0.98       120
weighted avg       0.98      0.98      0.98       120

F1 Score: 0.9873417721518987
AUC Score: 0.9875
```

In [34]:

```
# ROC Curve for Naive Bayes
fpr_nb, tpr_nb, _ = roc_curve(y_test_binary, y_pred_nb_binary)
roc_auc_nb = auc(fpr_nb, tpr_nb)

plt.figure(figsize=(8, 8))
plt.plot(fpr_nb, tpr_nb, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(r
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Naive Bayes)')
plt.legend(loc="lower right")
plt.show()
```



In [35]:
```
# Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) for Naive Bayes
mse_nb = mean_squared_error(y_test_binary, y_pred_nb_binary)
rmse_nb = np.sqrt(mse_nb)
print("Mean Squared Error (MSE):", mse_nb)
print("Root Mean Squared Error (RMSE):", rmse_nb)
```

```
Mean Squared Error (MSE): 0.016666666666666666
Root Mean Squared Error (RMSE): 0.12909944487358055
```

USER GUI

In [56]:
```
# Create the main window
root = tk.Tk()

# Create the main window with a themed style
style = ThemedStyle(root)
style.set_theme("blue")

# Set the title and icon
root.title("CKD Prediction")
root.iconbitmap("Erix-Subyarko-Medical-Medicine-Medicine-Tool-Doctor-Hospital-Stethoscope.512

# Set the size of the window (width x height)
root.geometry("1250x500")  # Change the size as needed
```

```
# Load the background image
background_image = tk.PhotoImage(file="23-Blood-test_1400x700-Copy.png")  # Replace with the
background_label = tk.Label(root, image=background_image)
background_label.place(relwidth=1, relheight=1)  # Make the label cover the entire window

# Create and place the title label
title_label = ttk.Label(root, text="Chronic Kidney Disease of Patients Prediction", font=('Ar
                        style="TLabel")
title_label.grid(row=0, column=0, columnspan=8, padx=10, pady=20)

# Create and place the guideline label with increased font size
guideline_label = ttk.Label(root, text="Please fill in the patient's health information in th
                                       "Then, click on the 'Predict' button below to generate
                            font=('Arial', 14), style="TLabel")
guideline_label.grid(row=1, column=0, columnspan=8, padx=10, pady=20)



# Create and place entry widgets for each feature in a 6x4 grid
entries = []
feature_labels = [
    "Age", "Blood Pressure mm/Hg", "Specific Gravity (1.005-1.025)", "Albumin (0-5)", "Sugar
    "Red Blood Cells (normal-1/abnormal-0)", "Pus Cell (normal-1/abnormal-0)", "Pus Cell Clum
    "Blood Glucose Random mgs/dl", "Blood Urea mgs/dl", "Serum Creatinine mgs/dl", "Sodium mE
    "Potassium mEq/L", "Hemoglobin gms", "Packed Cell Volume", "White Blood Cell Count cells/
    "Red Blood Cell Count cells/cumm", "Hypertension (yes-1/no-0)", "Diabetes Mellitus (yes-1
    "Coronary Artery Disease (yes-1/no-0)", "Appetite (good-1/poor-0)", "Pedal Edema (yes-1/n
]

row, col = 2, 0 # Start from row 2 (below the guideline label)
for i, label_text in enumerate(feature_labels):
    label = ttk.Label(root, text=label_text, style="TLabel")
    label.grid(row=row, column=col, padx=10, pady=5)
    entry = ttk.Entry(root, style="TEntry",width=10)
    entry.grid(row=row, column=col + 1, padx=10, pady=5)
    entries.append(entry)

    col += 2  # Move to the next column
    if col > 6:
        col = 0
        row += 1  # Move to the next row

# Create and place the predict button with a themed style
predict_button = ttk.Button(root, text="Predict", command=lambda: predict(entries, clf_gini,
                            style='TButton', takefocus=False)
predict_button.grid(row=row + 2, column=0, columnspan=8, pady=10)

# Create and place the result label
result_label = ttk.Label(root, text="Prediction for User Input:", style="PLabel.TLabel")
result_label.grid(row=row + 3, column=0, columnspan=8, pady=10)

# Add styling for labels and entries
style = ThemedStyle(root)
style.configure("TLabel", background=style.lookup("TFrame", "background"))
style.configure("TEntry", background=style.lookup("TFrame", "background"))
style.configure("PLabel.TLabel", font=('Arial', 12, 'bold'))


# Function to get user input for prediction
def get_user_input(entries):
    user_input = [float(entry.get()) for entry in entries]
    user_input_array = np.array(user_input).reshape(1, -1)
    return user_input_array


# Lambda function to predict and update the result label
predict_lambda = lambda: (
    user_input := get_user_input(entries),
```

```
    result_label.config(text=f"The model predicts: {clf_rf.predict(user_input)[0]}")
)

# Create and place the predict button without using the function
predict_button = ttk.Button(root, text="Predict", command=predict_lambda, style='TButton', ta
predict_button.grid(row=row + 2, column=0, columnspan=8, pady=10)

def predict(entries, clf_gini, clf_rf, clf_svm, result_label):
    pass

def get_user_input(entries):
    user_input = [float(entry.get()) for entry in entries]
    user_input_array = np.array(user_input).reshape(1, -1)
    return user_input_array

def clear_input(entries):
    for entry in entries:
        entry.delete(0, tk.END)

def clear_output(result_label):
    result_label.config(text="Prediction for User Input:")

def on_entry_return(event):
    current_entry = event.widget
    current_row, current_col = current_entry.grid_info()["row"], current_entry.grid_info()["c

    # Calculate the next entry's row and column
    next_row = current_row
    next_col = current_col + 2
    if next_col > 6:
        next_col = 0
        next_row += 1

    # Move the focus to the next entry
    entries[next_row][next_col].focus_set()

def clear_input_and_output(entries, result_label):
    clear_input(entries)
    clear_output(result_label)

# Create and place the clear button
clear_button = ttk.Button(root, text="Clear", command=lambda: clear_input_and_output(entries,
                          style='TButton', takefocus=False)
clear_button.grid(row=row + 4, column=0, columnspan=8, pady=10)

# Run the Tkinter event loop
root.mainloop()
```

Model Comparison

Accuracy Score

In [37]:
```
results = pd.DataFrame({
    'Model': ['Support Vector Machine', 'Decision Tree', 'Random Forest', 'Logistic Regressio
    'Accuracy Score': [
        accuracy_score(y_test, y_pred_svm) * 100,
        accuracy_score(y_test, y_pred_gini) * 100,
        accuracy_score(y_test, y_pred_rf) * 100,
        accuracy_score(y_test, y_pred_lr) * 100,  # Logistic Regression
        accuracy_score(y_test, y_pred_nb) * 100   # Naive Bayes
    ]
})

result_df = results.sort_values(by='Accuracy Score', ascending=False)
result_df = result_df.set_index('Accuracy Score')
result_df
```

Out[37]:

Model	Accuracy Score
Random Forest	**99.166667**
Naive Bayes	**98.333333**
Decision Tree	**96.666667**
Logistic Regression	**93.333333**
Support Vector Machine	**66.666667**

Classification Report

In [38]:
```
from sklearn.metrics import classification_report

# Create a DataFrame for Classification Report
classification_reports = []

# SVM
report_svm = classification_report(y_test, y_pred_svm, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'SVM', **report_svm['weighted avg']})

# Decision Tree
report_dt = classification_report(y_test, y_pred_gini, output_dict=True)
classification_reports.append({'Model': 'Decision Tree', **report_dt['weighted avg']})

# Random Forest
report_rf = classification_report(y_test, y_pred_rf, output_dict=True)
classification_reports.append({'Model': 'Random Forest', **report_rf['weighted avg']})

# Logistic Regression
report_lr = classification_report(y_test, y_pred_lr, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'Logistic Regression', **report_lr['weighted avg']})

# Naive Bayes
report_nb = classification_report(y_test, y_pred_nb, zero_division=1, output_dict=True)
classification_reports.append({'Model': 'Naive Bayes', **report_nb['weighted avg']})

# Create DataFrame
classification_df = pd.DataFrame(classification_reports)
classification_df = classification_df.set_index('Model')

# Sort DataFrame by Score in descending order
classification_df = classification_df.sort_values(by='f1-score', ascending=False).round(2)

# Print the DataFrame
print(classification_df)
```

```
                     precision  recall  f1-score  support
Model
Random Forest             0.99    0.99      0.99    120.0
Naive Bayes               0.98    0.98      0.98    120.0
Decision Tree             0.97    0.97      0.97    120.0
Logistic Regression       0.94    0.93      0.93    120.0
SVM                       0.78    0.67      0.53    120.0
```

MSE & RMSE

In [39]:
```
from sklearn.metrics import mean_squared_error
from math import sqrt

# Create a DataFrame for MSE and RMSE
mse_rmse_results = []

# SVM
```

```
mse_svm = mean_squared_error(y_test_binary, y_pred_svm_binary)
rmse_svm = sqrt(mse_svm)
mse_rmse_results.append({'Model': 'SVM', 'MSE': mse_svm, 'RMSE': rmse_svm})

# Decision Tree
mse_dt = mean_squared_error(y_test_binary, y_pred_gini_binary)
rmse_dt = sqrt(mse_dt)
mse_rmse_results.append({'Model': 'Decision Tree', 'MSE': mse_dt, 'RMSE': rmse_dt})

# Random Forest
mse_rf = mean_squared_error(y_test_binary, y_pred_rf_binary)
rmse_rf = sqrt(mse_rf)
mse_rmse_results.append({'Model': 'Random Forest', 'MSE': mse_rf, 'RMSE': rmse_rf})

# Logistic Regression
mse_lr = mean_squared_error(y_test_binary, y_pred_lr_binary)
rmse_lr = sqrt(mse_lr)
mse_rmse_results.append({'Model': 'Logistic Regression', 'MSE': mse_lr, 'RMSE': rmse_lr})

# Naive Bayes
mse_nb = mean_squared_error(y_test_binary, y_pred_nb_binary)
rmse_nb = sqrt(mse_nb)
mse_rmse_results.append({'Model': 'Naive Bayes', 'MSE': mse_nb, 'RMSE': rmse_nb})

# Create DataFrame
mse_rmse_df = pd.DataFrame(mse_rmse_results)
mse_rmse_df = mse_rmse_df.set_index('Model')

# Sort DataFrame by MSE in ascending order
mse_rmse_df = mse_rmse_df.sort_values(by='MSE', ascending=False).round(2)

# Print the DataFrame
print(mse_rmse_df)
```

```
                     MSE  RMSE
Model
SVM                 0.33  0.58
Logistic Regression 0.07  0.26
Decision Tree       0.03  0.18
Naive Bayes         0.02  0.13
Random Forest       0.01  0.09
```

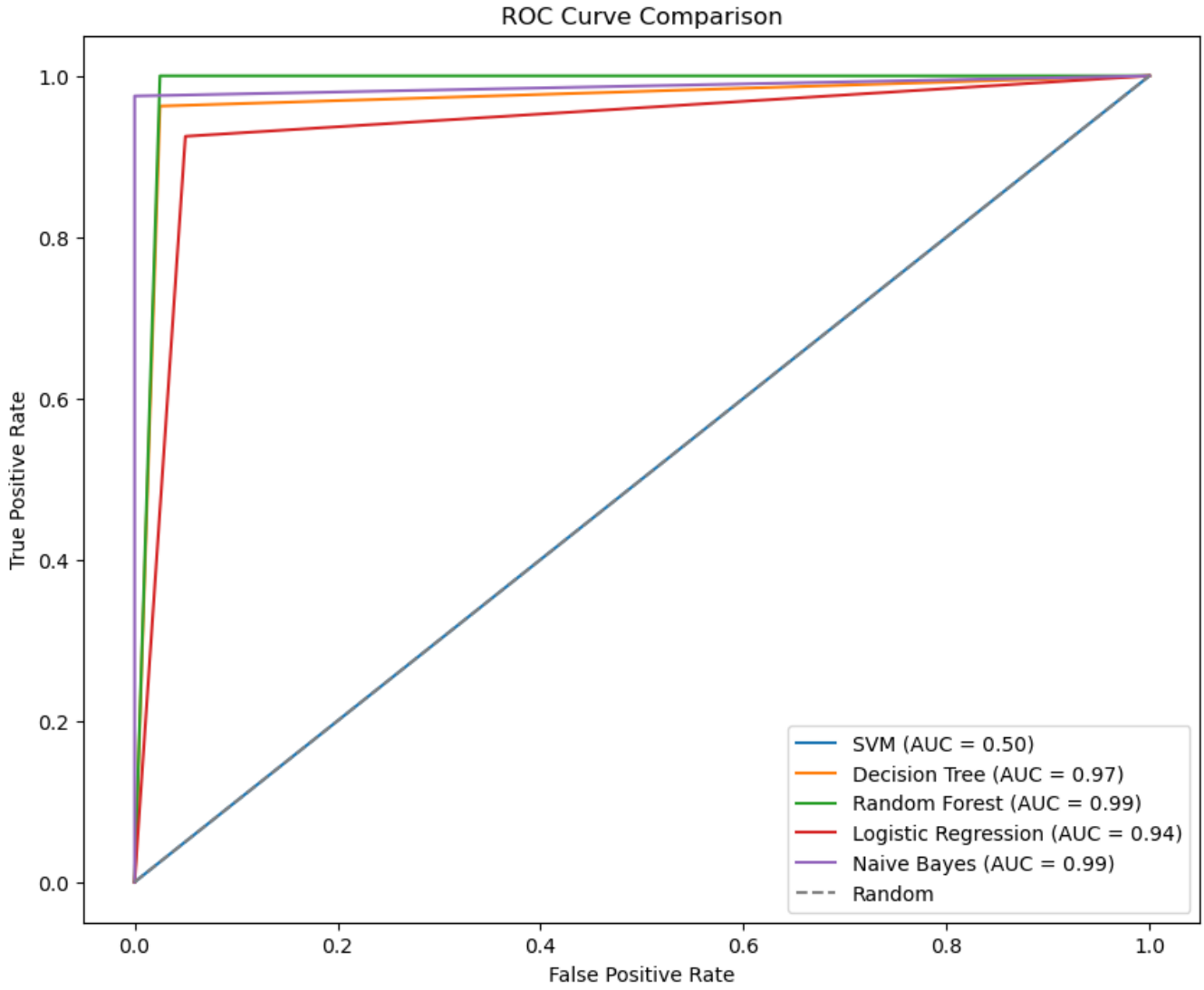
AUC Curve

In [40]:

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Create a DataFrame for AUC
auc_results = []

# SVM
fpr_svm, tpr_svm, _ = roc_curve(y_test_binary, y_pred_svm_binary)
roc_auc_svm = auc(fpr_svm, tpr_svm)
auc_results.append({'Model': 'SVM', 'AUC': roc_auc_svm})

# Decision Tree
fpr_dt, tpr_dt, _ = roc_curve(y_test_binary, y_pred_gini_binary)
roc_auc_dt = auc(fpr_dt, tpr_dt)
auc_results.append({'Model': 'Decision Tree', 'AUC': roc_auc_dt})

# Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_test_binary, y_pred_rf_binary)
roc_auc_rf = auc(fpr_rf, tpr_rf)
auc_results.append({'Model': 'Random Forest', 'AUC': roc_auc_rf})

# Logistic Regression
fpr_lr, tpr_lr, _ = roc_curve(y_test_binary, y_pred_lr_binary)
roc_auc_lr = auc(fpr_lr, tpr_lr)
```

```
auc_results.append({'Model': 'Logistic Regression', 'AUC': roc_auc_lr})

# Naive Bayes
fpr_nb, tpr_nb, _ = roc_curve(y_test_binary, y_pred_nb_binary)
roc_auc_nb = auc(fpr_nb, tpr_nb)
auc_results.append({'Model': 'Naive Bayes', 'AUC': roc_auc_nb})

# Create DataFrame
auc_df = pd.DataFrame(auc_results)
auc_df = auc_df.set_index('Model').round(2)

# Sort DataFrame by AUC in descending order
auc_df = auc_df.sort_values(by='AUC', ascending=False)

# Print the DataFrame
print(auc_df)

# Plot ROC curves
plt.figure(figsize=(10, 8))
plt.plot(fpr_svm, tpr_svm, label=f'SVM (AUC = {roc_auc_svm:.2f})')
plt.plot(fpr_dt, tpr_dt, label=f'Decision Tree (AUC = {roc_auc_dt:.2f})')
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {roc_auc_rf:.2f})')
plt.plot(fpr_lr, tpr_lr, label=f'Logistic Regression (AUC = {roc_auc_lr:.2f})')
plt.plot(fpr_nb, tpr_nb, label=f'Naive Bayes (AUC = {roc_auc_nb:.2f})')

plt.plot([0, 1], [0, 1], linestyle='--', color='grey', label='Random')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend(loc='lower right')
plt.show()
```

```
                     AUC
Model
Random Forest       0.99
Naive Bayes         0.99
Decision Tree       0.97
Logistic Regression 0.94
SVM                 0.50
```


ROC Curve Comparison
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
SVM (AUC = 0.50)
Decision Tree (AUC = 0.97)
Random Forest (AUC = 0.99)
Logistic Regression (AUC = 0.94)
Naive Bayes (AUC = 0.99)
Random

BSD3533 DATA MINING
GROUP PROJECT: MARKING SCHEME

Rubric for CO2

CO2: Design data mining model prototypes/models and demonstrate critical thinking ideas in data mining knowledge and problem-solving.								
CRITERIA	**LEVEL OF ACHIEVEMENT**						**WEIGHTAGE**	**SCORE A**
	0 Grossly Inadequate	**1 Inadequate**	**2 Emerging**	**3 Developing**	**4 Good**	**5 Excellent**		
Description and explanation of the selected project and problem to be solved.	No description or explanation about the project selected and the problem to be solved in the report.	Barely describe and explain the project selected and the problem to be solved in the report.	Partly describe and explain the project selected and the problem to be solved in the report.	Sufficiently describe and explain the project selected and the problem to be solved in the report but unclear.	Clearly describe and explain the project selected and the problem to be solved in the report.	Excellently describe and explain the project selected and the problem to be solved in the report.	0.5	
Explanation regarding the summary of the project context and objectives.	Failed to explain the summary of the project context and objectives.	Barely explain the summary of the project context and objectives.	Partly explain the summary of the project context and objectives.	Sufficiently explain the summary of the project context and objectives but unclear.	Clearly explain the summary of the project context and objectives.	Excellently explain the summary of the project context and objectives.	1	
Explanation of the project methodology.	Failed to explain the project methodology.	Barely explain the project methodology.	Partly explain the project methodology.	Sufficiently explain the project methodology but unclear.	Clearly explain the project methodology.	Excellently explain the project methodology.	1	
Analysis explanations and interpretations of the results and findings	The analysis is insufficiently explained. The interpretation of the results is not discussed.	The analysis is haphazardly explained. The interpretation of the results is not clearly discussed.	The analysis lacks sufficient explanation. The interpretation of the results is sufficiently discussed.	The analysis is sufficiently and logically explained. The interpretation of the results is sufficiently discussed.	The analysis is clearly and logically explained. The interpretation of the results is clearly discussed.	The analysis is excellently and logically explained. The interpretation of the results is well discussed.	1	
Concluding remarks.	No concluding remarks were provided.	Barely concluding remarks provided and inaccurate.	Concluding remarks were provided but unclear and inaccurate.	Concluding remarks were provided but partly inaccurate.	Clear and good concluding remarks were provided.	Very clear and excellent concluding remarks were provided.	0.5	
						TOTAL		**20**

Rubric for CO3

CO3: Construct the programming codes or workflows using appropriate analytics tools.								
CRITERIA	**LEVEL OF ACHIEVEMENT**						**WEIGHTAGE**	**SCORE**
	0 Grossly Inadequate	**1 Inadequate**	**2 Emerging**	**3 Developing**	**4 Good**	**5 Excellent**		
Ability to extract the datasets from sources very well.	Unable to extract the datasets.	Barely able to extract the datasets from the sources.	Partly able to extract the datasets from the sources.	Able to extract the datasets from the sources in partly successful results.	Very good for extracting the datasets from the sources.	Excellently extract the datasets from the sources excellently.	0.5	
Ability to apply data pre-processing/ ETL or ELT/ algorithms.	Unable to apply data pre-processing/ ETL or ELT/ algorithms.	Barely able to apply data pre-processing ETL or ELT/ algorithms.	Partly able to apply data pre-processing/ ETL or ELT/ algorithms.	Able to apply data pre-processing/ ETL or ELT/ algorithms in partly successful results.	Very good for applying data pre-processing/ ETL or ELT/ algorithms.	Excellently apply data pre-processing/ ETL or ELT/ algorithms.	1	
Ability to construct data mining algorithms for the modelling process.	Unable to construct data mining algorithms for the modelling process.	Barely able to construct data mining algorithms for the modelling process.	Partly able to construct data mining algorithms for the modelling process.	Able to construct data mining algorithms for the modelling process in partly successful results.	Very good for constructing data mining algorithms for the modelling process.	Excellently construct data mining algorithms for the modelling process.	2	
Ability to evaluate and deploy the model.	Unable to evaluate and deploy the model.	Barely able to evaluate and deploy the model.	Partly able to evaluate and deploy the model.	Able to evaluate and deploy the model with partly successful results.	Very good for evaluating and deploying the model with successful results.	Excellently evaluate and deploy the model.	2	
Ability to visualise (table, graph, etc) the programming codes.	Unable to visualise (table, graph, etc) the programming codes.	Barely able to visualise (table, graph, etc) the programming codes.	Partly able to visualise (table, graph, etc) the programming codes.	Ability to visualise (table, graph, etc) the programming codes in successful results.	Very good for visualising (table, graph, etc) the programming codes.	Excellently visualise (table, graph, etc) the programming codes.	1	
The code can be executed correctly and easy to understand the codes constructed.	No code is constructed.	Only a few codes can be executed correctly and difficult to follow the structure and flow of the codes.	Some of the codes can be executed correctly and fairly difficult to follow the structure and flow of the codes.	The code can be executed correctly and fairly easy to follow the structure and flow of the codes.	The code can be executed correctly and easily to follow the structure and flow of the codes.	The code can be executed correctly and well easily follows the structure and flow of the codes.	0.5	
Level of programming difficulty and effort.	No codes developed/ modified.	Codes developed/ modified are not challenging, less ambitious and take little effort.	Codes developed/ modified are not challenging, less ambitious and take little effort.	Codes developed/ modified is less challenging, moderately ambitious and take some effort.	Codes developed/ modified are quite challenging, moderately ambitious and take effort.	Codes developed/ modified is very challenging, ambitious and take extra effort.	1	
						TOTAL		**40**

Rubric for CO4

CO4: Demonstrate verbal and written communication skills.								
CRITERIA	**LEVEL OF ACHIEVEMENT**						**WEIGHTAGE**	**SCORE**
	0 Grossly Inadequate	**1 Inadequate**	**2 Emerging**	**3 Developing**	**4 Good**	**5 Excellent**		
Verbal: Able to communicate with team and lecturer.	Do not commit to any discussion session presentation.	Poorly able to communicate/ discuss the data mining knowledge in presentation.	Fairly able to communicate/ discuss the data mining knowledge in presentation.	Able to communicate/ discuss the data mining knowledge in presentation.	Able to communicate/ discuss in a good way regarding the data mining knowledge in the presentation.	Able to communicate/ discuss excellently regarding the data mining knowledge in presentation.	1	
Verbal: Able to provide general and background information which corresponds with the purpose. Show understanding of the topic.	Unable to provide general and background information which corresponds with the purpose. Refrain from showing understanding of the topic.	General and background information are inadequate and do not correspond with the purpose. Show poor understanding of the topic.	General and background information is slightly inadequate and fairly corresponds with the purpose. Show fair understanding of the topic.	General and background information are adequate but moderately correspond with the topic. Show average understanding of the topic.	General and background information are adequate and correspond with the topic. Show a good understanding of the topic.	General and background information are adequate and correspond with the topic. Show excellent understanding of the topic.	1	
Written: Able to demonstrate report clearly, coherent and systematically.	Do not submit a group project report.	Poorly able to demonstrate report clearly, coherent and systematically.	Fairly able to demonstrate report clearly, coherent and systematically.	Able to demonstrate reports clearly, coherently and systematically.	Clearly demonstrate a good report, coherent and systematically.	Demonstrate an excellent report clearly, coherently and systematically.	1	
Written: Able to provide adequate, relevant and significant information about the chapters. Provide links between ideas.	Unable to provide adequate, relevant and significant information on the chapters. Do not provide links between ideas	Information is inadequate; The relevance and significance of information are poorly made; Links and connections between ideas is poorly made.	Information is present, but supporting details are inadequate; The relevance and significance of information are fairly made; Links and connections between ideas is fairly made.	Information is adequate; The relevance and significance of information are moderately made; Links and connections between ideas is moderately made.	Information is adequate; The relevance and significance of information are good; Links and connections between ideas are good.	Information is adequate; The relevance and significance of information are excellent; Links and connections between ideas is excellent.	1	
						TOTAL		**20**

Rubric for CO5

CO5: Integrate data mining knowledge to the project and future problems.								
CRITERIA	**LEVEL OF ACHIEVEMENT**						**WEIGHTAGE**	**SCORE**
	0 Grossly Inadequate	**1 Inadequate**	**2 Emerging**	**3 Developing**	**4 Good**	**5 Excellent**		
Knowledge application or transfer.	No reference from previous learning and do not apply the knowledge in methodology, result and discussion.	Makes vague reference to previous learning but does not apply the data mining knowledge to performance in methodology, result and discussion.	Makes defined reference to previously gained knowledge but does not apply to methodology, result and discussion.	Makes defined reference to gathered previous knowledge, and demonstrates the limited capacity to utilize methodology, result and discussion.	Demonstrates reference to the previously gained knowledge, and demonstrates strong application to methodology, result and discussion.	Makes explicit reference to previous knowledge, and applies these skills to methodology, results and discussion. Creative methods/ manners.	1	
Able to pursue data mining knowledge beyond expectation in completing the project.	Unable to pursue data mining knowledge beyond expectation in completing the project.	Rarely pursue data mining knowledge beyond expectation in completing the project.	Fairly pursue data mining knowledge beyond expectation in completing the project.	Sometimes pursue data mining knowledge beyond expectation in completing the project.	Often pursue data mining knowledge beyond expectation in completing the project.	Always pursue data mining knowledge beyond expectation in completing the project.	1	
Look for relevant information independently.	Too dependent on others to look for relevant and reliable information in the project.	Always needs help from others to look for relevant and reliable information in doing the project.	Continuously needs help from others to look for relevant and reliable information in doing the project.	Often needs help from others to look for relevant and reliable information in doing the project.	Sometimes needs help from others to look for relevant and reliable information in doing the project.	Rarely needs help from others to look for relevant and reliable information in doing the project.	1	
Reflection from prior learning towards life experience.	Do not review any prior learning at any level.	Reviews prior learning at a surface level, but without clarifying meaning or indicating a broader perspective about educational or life events.	Reviews prior learning with limited capacity, giving minor clarification or broad perspective.	Reviews prior learning and shows clarification or broad perspective about educational or life events.	Reviews prior learning in some depth, revealing clear meaning and indicating broad perspectives related to educational events.	Review prior learning in depth to reveal significantly changed perspectives about educational and life experiences.	1	
						TOTAL		**20**