SMARTBRIDGE
Let's Bridge the Gap

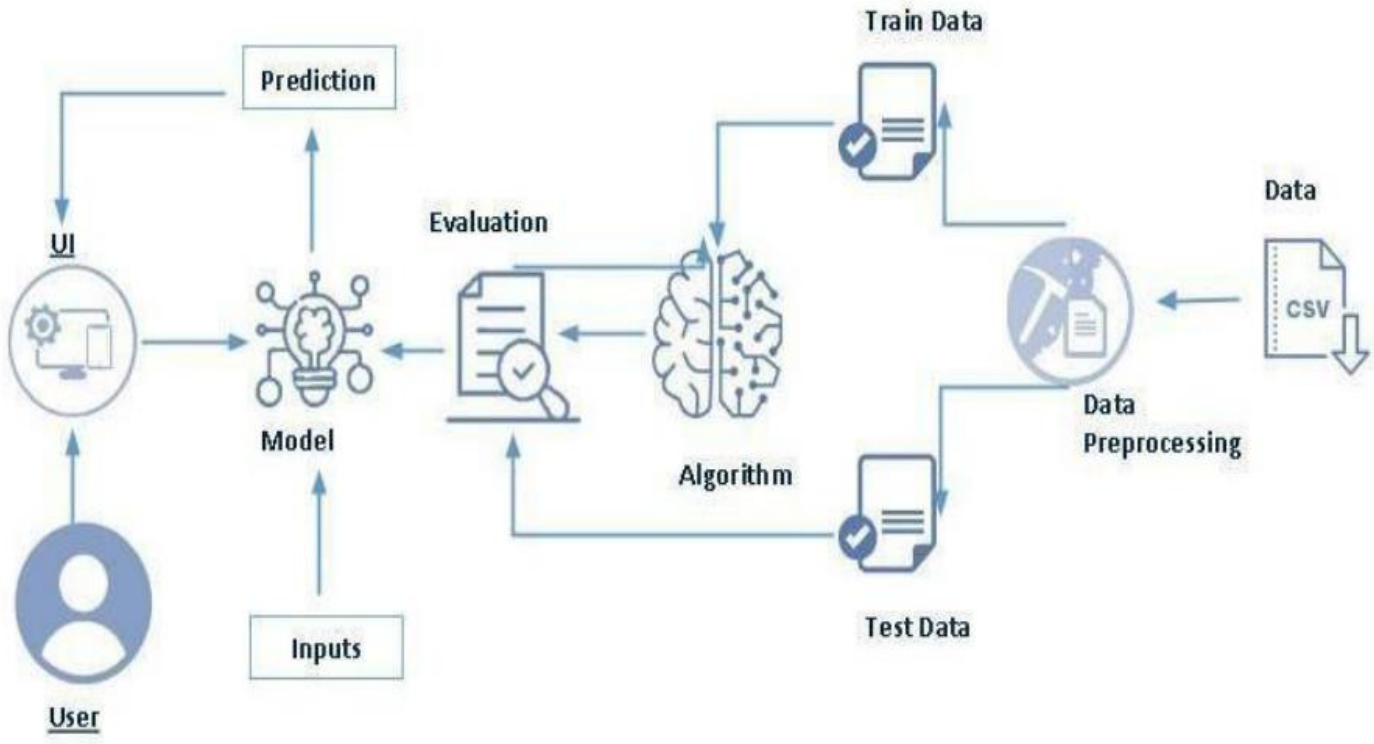
Smart Internz

EARLY PREDICTION FOR CHRONIC KIDNEY DISEASE DETECTION: A PROGRESSIVE APPROACH TO HEALTH MANAGEMENT

Project overview:

The objective of this project is to develop a machine learning model that can accurately predict early-stage Chronic Kidney Disease (CKD) using structured patient medical data. CKD is often diagnosed in later stages, leading to serious complications and higher treatment costs. Early detection remains a challenge due to the lack of predictive tools in initial diagnosis. This project focuses on applying supervised learning techniques to historical patient records to address this gap. Multiple classification algorithms are trained and evaluated after performing thorough data preprocessing, encoding, and exploratory data analysis. Feature selection techniques are applied to enhance model performance, and the best-performing model is chosen based on defined evaluation metrics. The project does not include deployment or real-time integration but emphasizes building a robust predictive system that can support healthcare professionals in identifying at-risk individuals sooner. Key features of this solution include data preprocessing, model training, feature importance analysis, and performance visualization.

Technical Architecture:



Project Flow:

User interacts with the UI to enter the input. Entered input is analysed by the model which is integrated. Once model analyses the input the prediction is showcased on the UI. To accomplish this, we complete all the activities listed below,

- Define Problem / Problem Understanding
  - Specify the business problem
  - Business requirements
  - Literature Survey
  - Social or Business Impact.

- Data Collection & Preparation
  - Collect the dataset
  - Data Preparation

- Exploratory Data Analysis
  - Descriptive statistical
  - Visual Analysis

- Model Building
  - Training the model in multiple algorithms
  - Testing the model

- Performance Testing & Hyperparameter Tuning
  - Testing model with multiple evaluation metrics
  - Comparing model accuracy before & after applying hyperparameter tuning

- Model Deployment
  - Save the best model
  - Integrate with Web Framework

- Project Demonstration & Documentation
  - Record explanation Video for project end to end solution
  - Project Documentation-Step by step project development procedure

Milestone 1: Define Problem / Problem Understanding

Activity 1: Specify the business problem

Chronic Kidney Disease (CKD) progresses silently in its early stages and is often diagnosed only after significant kidney damage has occurred. This late detection limits treatment options and worsens patient outcomes. Despite the availability of patient health data, primary care settings often lack effective tools to predict CKD early. The challenge is to build a reliable, data-driven model that can analyze medical parameters and accurately identify the likelihood of early-stage CKD before symptoms become severe.

Activity 2: Business requirements

For the successful deployment of a CKD early prediction model, several key business requirements must be addressed. The model must achieve high accuracy and reliability to ensure its predictions are clinically valid and support trustworthy diagnoses and treatment decisions. Real-time or near-real-time data processing capabilities are essential, allowing seamless integration with healthcare systems such as Electronic Health Records (EHRs) to enable continuous patient monitoring. The system should also be scalable and flexible, capable of being deployed across multiple hospitals or clinics and easily updated to incorporate new features or adhere to evolving diagnostic standards. Compliance with healthcare regulations, including data privacy laws like HIPAA, is critical to ensure the secure handling of sensitive patient information. Lastly, the user interface should be designed with usability in mind, offering healthcare professionals clear, interpretable outputs that can aid in efficient and informed medical decision-making.

Activity 3: Literature Survey

A literature review for CKD prediction involves examining existing studies, algorithms, and medical findings that contribute to understanding and improving predictive modeling for the disease. Numerous studies have demonstrated that machine learning algorithms such as Random Forest, Support Vector Machines (SVM), and Logistic Regression are capable of accurately predicting CKD using clinical data. These models often rely on features such as age, blood pressure, serum creatinine, and blood urea—variables known to correlate strongly with kidney health. Publicly available datasets, particularly the UCI CKD dataset, are commonly used for research and experimentation due to their accessibility and structured clinical attributes.

Comparative analyses in the literature reveal varying performance levels among different algorithms, with an emphasis on the importance of feature selection, effective preprocessing, and model interpretability—especially in medical applications where transparency is crucial. However, despite promising results, many existing studies face limitations. Common gaps include the lack of real-world deployment, poor handling of imbalanced datasets, limited generalizability across diverse populations, and insufficient focus on creating user-friendly interfaces for clinical use. Addressing these gaps will guide the design, development, and evaluation of the proposed CKD prediction system, ensuring it is both effective and practical for real-world healthcare settings.

Activity 4: Social or Business Impact.

The early prediction of Chronic Kidney Disease (CKD) through machine learning has significant social and business impacts. Socially, it enables early intervention, allowing healthcare providers to identify at-risk individuals before the disease progresses to critical stages. This proactive approach can prevent severe complications, reduce the need for dialysis or kidney transplants, and ultimately enhance the quality of life for patients. Additionally, predictive insights empower patients by increasing awareness of their health risks, encouraging lifestyle changes, and promoting adherence to medical advice - all of which contribute to better long-term outcomes.

From a business perspective, the implementation of a predictive system can lead to substantial cost reductions. Preventive care is far less expensive than treating advanced CKD, translating to savings for both healthcare providers and insurance companies. Furthermore, automating the identification of high-risk patients improves operational efficiency by optimizing the use of medical staff and resources. Finally, the project supports data-driven innovation by integrating artificial intelligence into clinical practice, paving the way for smarter, more responsive, and technology-enhanced healthcare delivery systems.

Milestone 2: Data Collection & Preparation

ML depends heavily on data. It is the most crucial aspect that makes algorithm training possible. So, this section allows you to download the required dataset.

Activity 1: Collect the dataset

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc.
In this project we have used chronickidneydisease.csv data. This data is downloaded from kaggle.com. Please refer to the link given below to download the dataset.
Link: https://www.kaggle.com/datasets/mansoordaku/ckdiseaset

Activity 1.1: Importing the libraries

Import the necessary libraries.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.experimental import enable_iterative_imputer  # noqa
from sklearn.impute import IterativeImputer
from sklearn.preprocessing import StandardScaler
import os
from scipy.stats import zscore
ts.outliers_influence import variance_inflation_factor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import validation_curve, learning_curve, StratifiedKFold, cross_val_score
from sklearn.metrics import (f1_score, accuracy_score, precision_score, recall_score,
                             roc_auc_score, average_precision_score, confusion_matrix,
                             roc_curve, precision_recall_curve, classification_report)
```

```
from sklearn.linear_model import LogisticRegression, RidgeClassifier, SGDClassifier
from sklearn.ensemble import (RandomForestClassifier, GradientBoostingClassifier,
                              ExtraTreesClassifier, AdaBoostClassifier, BaggingClassifier,
                              VotingClassifier)
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
from sklearn.dummy import DummyClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier
import warnings
warnings.filterwarnings('ignore')
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV, StratifiedKFold, cross_val_score,
```

Activity 1.2: Read the Dataset

Our dataset is in csv format. We can read the dataset with the help of pandas. In pandas we have a function called read_csv() to read the dataset. As a parameter we have to give the directory of the csv file.

```
df = pd.read_csv('chronickidneydisease.csv')
df.head()
```

	id	age	bp	sg	al	su	rbc	pc	pcc	ba	...	pcv	wc	rc	htn	dm	cad	appet	pe	ane	classification
0	0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	...	44	7800	5.2	yes	yes	no	good	no	no	ckd
1	1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	...	38	6000	NaN	no	no	no	good	no	no	ckd
2	2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	...	31	7500	NaN	no	yes	no	poor	no	yes	ckd
3	3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	...	32	6700	3.9	yes	no	no	poor	yes	yes	ckd
4	4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	...	35	7300	4.6	no	no	no	good	no	no	ckd

5 rows × 26 columns

Activity 2: Data Preparation

The next step is to pre-process the data. The data set is not suitable for training the machine learning model as it might have so much randomness so we need to clean the dataset properly in order to fetch good results. This includes the following steps.

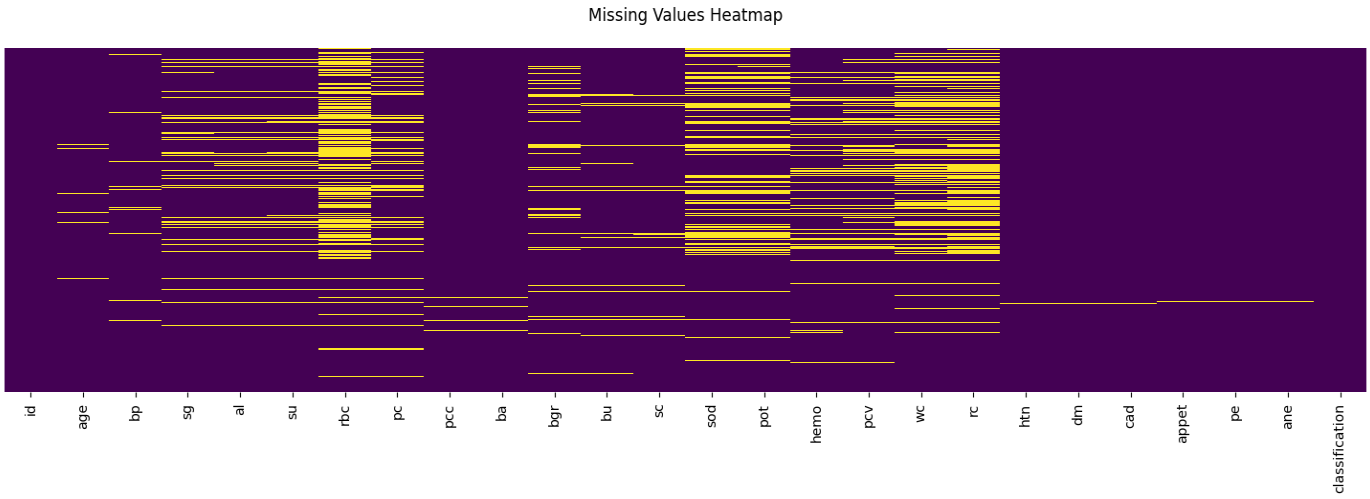
- Handling missing values
- Handling Outliers
- Handling duplicates
- Cleaning columns and changing incorrect datatypes

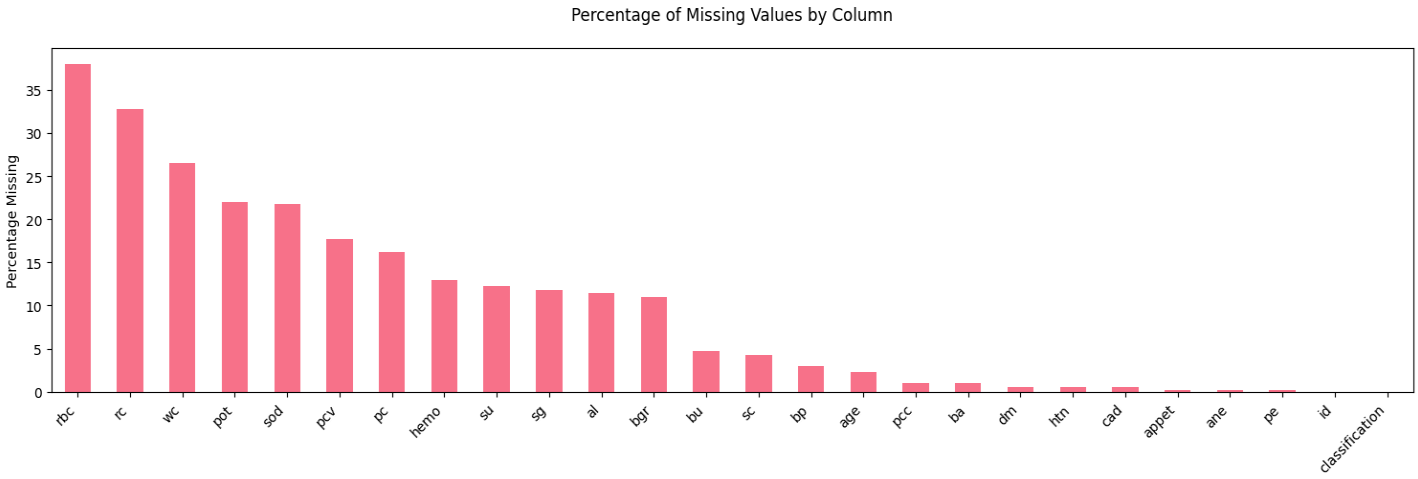
```
unclean_categorical_columns=['classification','cad','dm']

def clean_categorical_column(series):
    series = series.astype(str).str.replace('\t','')
    series = series.astype(str).str.replace(' ', '')
    series = series.replace('?', np.nan)
    series= series.replace('nan',np.nan)
    return series

for column in unclean_categorical_columns:
    df_clean[column]=clean_categorical_column(df_clean[column])
```

Activity 2.1: Handling missing values

```
# Plot 1: Missing value heatmap
sns.heatmap(df.isnull(), yticklabels=False, cbar=False, cmap='viridis', ax=ax1)
ax1.set_title('Missing Values Heatmap', pad=20)
```

```
# Plot 2: Missing values percentage bar plot
missing_percentage.sort_values(ascending=False).plot(kind='bar', ax=ax2)
ax2.set_title('Percentage of Missing Values by Column', pad=20)
ax2.set_ylabel('Percentage Missing')
plt.xticks(rotation=45, ha='right')
```





Handling missing values:

1)

- Low Missing (<5%)>:
  - Categorical
    - Filled with mode
  - Numerical
    - Filled with mean

```
[ ]  low_missing_cats = ['pcc', 'ba', 'htn', 'dm', 'cad', 'appet', 'pe', 'ane']
     for col in low_missing_cats:
         mode_val = df_clean[col].mode()[0]
         df_clean[col].fillna(mode_val, inplace=True)
```

```
[ ]  df_clean['age'].fillna(df_clean['age'].mean(), inplace=True)
     df_clean['bp'].fillna(df_clean['bp'].mean(), inplace=True)
```

2)

- Too Many (>30%):
  - Categorical
    - Make separate category called missing
  - Numerical
    - None found

```
[ ]  df_clean['rbc'] = df_clean['rbc'].fillna('missing')
     df_clean['pc'] = df_clean['pc'].fillna('missing')
```

3)

- Intermediate Amount (5-30%):
  - Mice Imputer using default imputer model

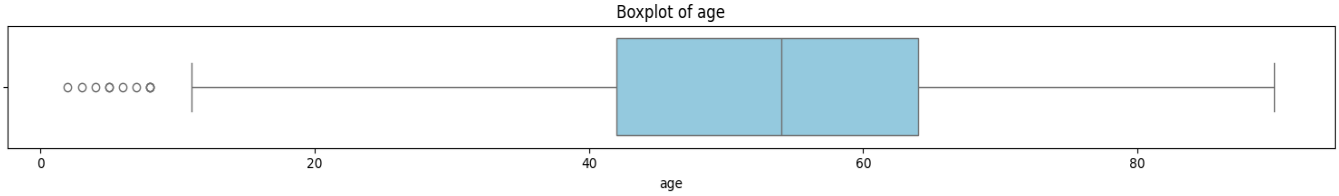
```
def mice_impute_group(df, columns, estimator=None, max_iter=10, random_state=0):
    scaler = StandardScaler()
    scaled = scaler.fit_transform(df[columns])
    imputer = IterativeImputer(estimator=estimator, max_iter=max_iter, random_state=random_state)
    imputed = imputer.fit_transform(scaled)
    df[columns] = scaler.inverse_transform(imputed)
    return df
```

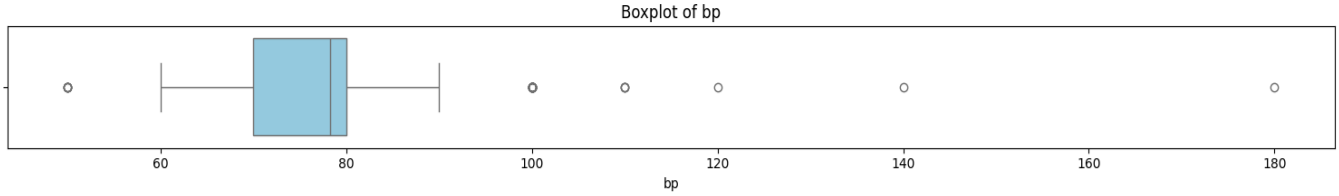
```
[ ]  # Impute each group
     df_clean = mice_impute_group(df_clean, ['sod', 'pot'])
     df_clean = mice_impute_group(df_clean, ['sg', 'al', 'su'])
     df_clean = mice_impute_group(df_clean, ['hemo', 'pcv', 'wc', 'rc'])
     df_clean = mice_impute_group(df_clean, ['bu', 'sc', 'bgr'])  # bgr included for completeness
```

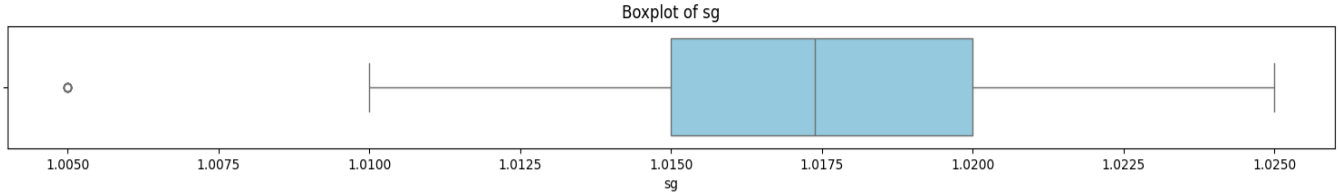
Activity 2.2: Handling Outlier

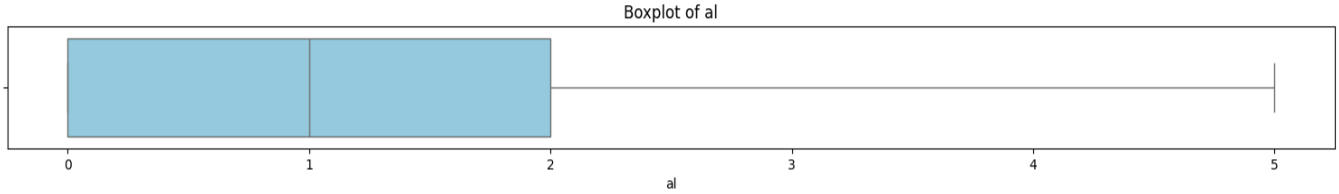
With the help of boxplot, outliers are visualized.

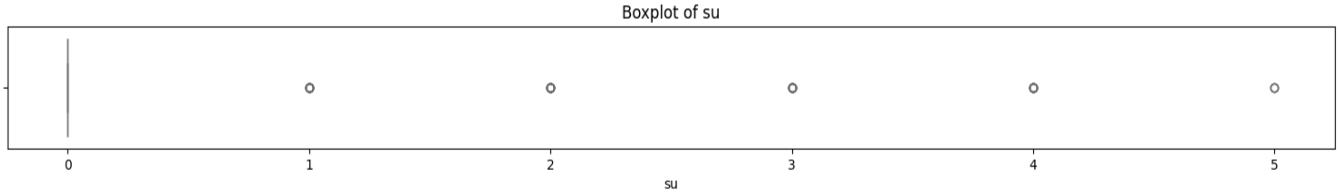
```
[ ]  numerical_cols = df_clean.select_dtypes(include=['float64', 'int64']).columns.tolist()
     numerical_cols = [col for col in numerical_cols if col not in ['id']]  # Exclude id

     plt.figure(figsize=(15, 2 * len(numerical_cols)))
     for i, col in enumerate(numerical_cols, 1):
         plt.subplot(len(numerical_cols), 1, i)
         sns.boxplot(x=df_clean[col], color='skyblue')
         plt.title(f'Boxplot of {col}')
         plt.tight_layout()
     plt.show()
```

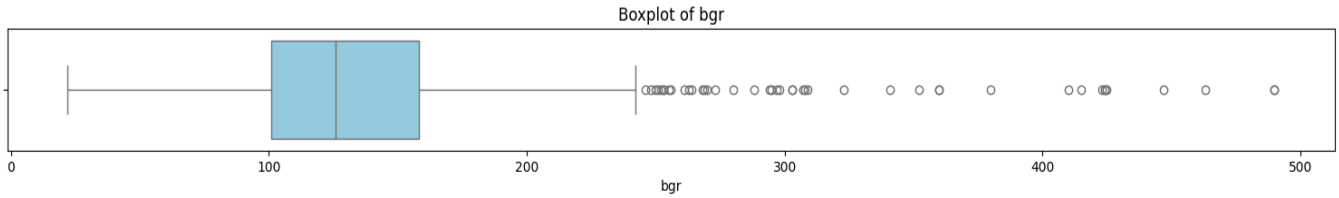

Boxplot of age
0
20
40
60
80
age

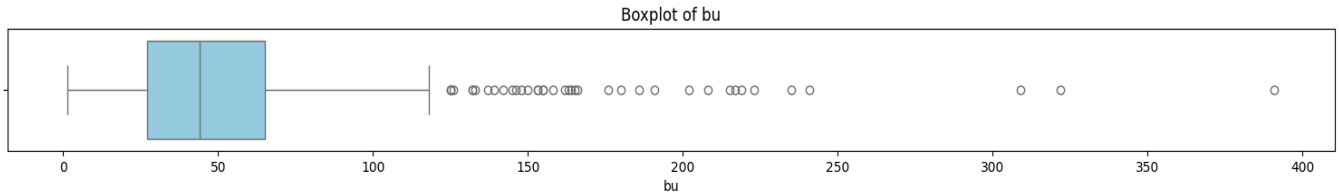

Boxplot of bp
60
80
100
120
140
160
180
bp

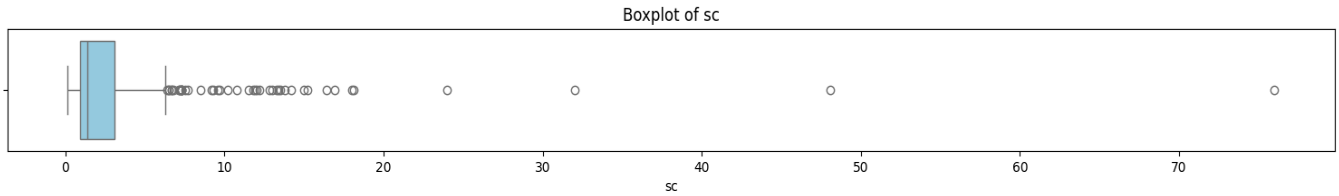

Boxplot of sg
1.0050
1.0075
1.0100
1.0125
1.0150
1.0175
1.0200
1.0225
1.0250
sg


Boxplot of al
0
1
2
3
4
5
al


Boxplot of su
0
1
2
3
4
5
su


Boxplot of bgr
0
100
200
300
400
500
bgr


Boxplot of bu
0
50
100
150
200
250
300
350
400
bu


Boxplot of sc
0
10
20
30
40
50
60
70
sc

To calculate the outlier in each numerical category:

```
outlier_summary = {}
for col in numerical_cols:
    Q1 = df_clean[col].quantile(0.25)
    Q3 = df_clean[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = df_clean[(df_clean[col] < lower_bound) | (df_clean[col] > upper_bound)][col]
    outlier_summary[col] = {
        'lower_bound': lower_bound,
        'upper_bound': upper_bound,
        'num_outliers': outliers.count(),
        'outlier_values': outliers.values
    }
    print(f"{col}: {outliers.count()} outliers")
```

```
age: 10 outliers
bp: 36 outliers
sg: 7 outliers
al: 0 outliers
su: 65 outliers
bgr: 43 outliers
bu: 39 outliers
sc: 44 outliers
sod: 18 outliers
pot: 15 outliers
hemo: 2 outliers
pcv: 4 outliers
wc: 17 outliers
rc: 1 outliers
```

We checked all the medical data ranges for all these numerical columns for which box plots were made and decided to remove only those that were clinically impossible. Other outliers were possible, but as a result of severe disease. Since the other records provided valuable medical information, we decided to keep them.

sod (Sodium):

- Removed rows where sod = 4.5 mEq/L (physiologically impossible).
- Retained all other values, including extreme hyponatremia/hypernatremia, as these are possible in severe kidney disease.

pot (Potassium):

- Removed rows where pot ≥ 39 mmol/L (clinically impossible).
- Retained all other values, including severe hypo/hyperkalaemia, as these are possible in advanced disease.

pcv (Packed Cell Volume):

- Removed rows where pcv = 9% (clinically impossible).
- Retained all other values, including severe anemia, as these are possible in advanced disease.

All other columns:

- Retained all outlier values, as they are medically possible, though some are rare or dangerous. This

preserves the clinical diversity of the dataset for robust modelling

```
[ ] # Count before removal
    removed_sod = np.sum(np.isclose(df_clean['sod'], 4.5, atol=0.01))
    removed_pot = np.sum(df_clean['pot'] >= 39)
    removed_pcv = np.sum(np.isclose(df_clean['pcv'], 9.0, atol=0.01))

    print(f"Rows to be removed for sod: {removed_sod}")
    print(f"Rows to be removed for pot: {removed_pot}")
    print(f"Rows to be removed for pcv: {removed_pcv}")

    # Now remove the impossible values
    df_clean = df_clean[~np.isclose(df_clean['sod'], 4.5, atol=0.01) | df_clean['sod'].isnull()]
    df_clean = df_clean[(df_clean['pot'].isnull()) | (df_clean['pot'] < 39)]
    df_clean = df_clean[~np.isclose(df_clean['pcv'], 9.0, atol=0.01)]
```

```
Rows to be removed for sod: 1
Rows to be removed for pot: 2
Rows to be removed for pcv: 1
```

Converting Number/Object to Category Wherever Needed:

```
[ ] # Convert to categorical with ordered categories
    df_clean['al'] = pd.Categorical(df_clean['al'], categories=[0,1,2,3,4,5], ordered=True)
    df_clean['su'] = pd.Categorical(df_clean['su'], categories=[0,1,2,3,4,5], ordered=True)
```

Milestone 3: Exploratory Data Analysis

Activity 1: Descriptive statistical

```
df.describe()
```

	id	age	bp	sg	al	su	bgr	bu	sc	sod	pot	hemo
count	400.000000	391.000000	388.000000	353.000000	354.000000	351.000000	356.000000	381.000000	383.000000	313.000000	312.000000	348.000000
mean	199.500000	51.483376	76.469072	1.017408	1.016949	0.450142	148.036517	57.425722	3.072454	137.528754	4.627244	12.526437
std	115.614301	17.169714	13.683637	0.005717	1.352679	1.099191	79.281714	50.503006	5.741126	10.408752	3.193904	2.912587
min	0.000000	2.000000	50.000000	1.005000	0.000000	0.000000	22.000000	1.500000	0.400000	4.500000	2.500000	3.100000
25%	99.750000	42.000000	70.000000	1.010000	0.000000	0.000000	99.000000	27.000000	0.900000	135.000000	3.800000	10.300000
50%	199.500000	55.000000	80.000000	1.020000	0.000000	0.000000	121.000000	42.000000	1.300000	138.000000	4.400000	12.650000
75%	299.250000	64.500000	80.000000	1.020000	2.000000	0.000000	163.000000	66.000000	2.800000	142.000000	4.900000	15.000000
max	399.000000	90.000000	180.000000	1.025000	5.000000	5.000000	490.000000	391.000000	76.000000	163.000000	47.000000	17.800000

Activity 2: Visual analysis

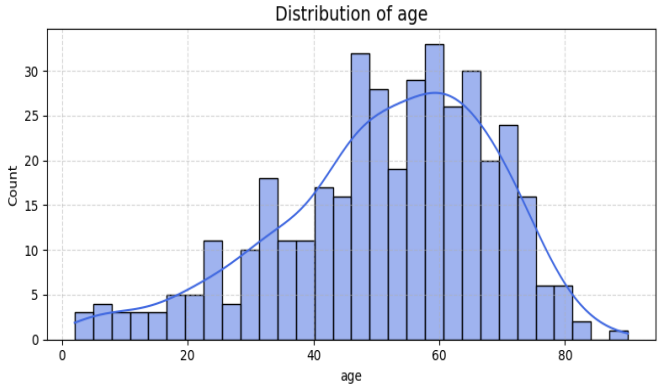
Activity 2.1: Univariate analysis

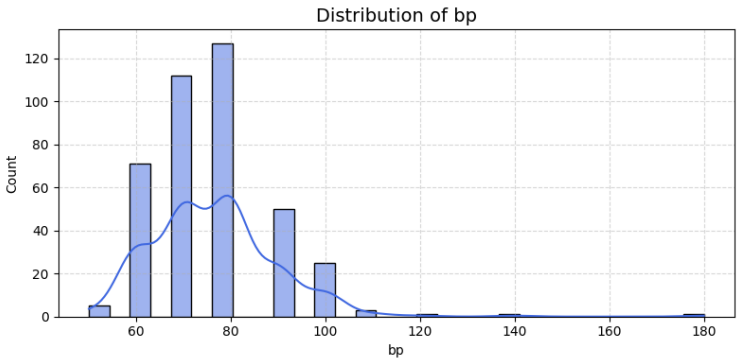
Distribution plot for Numerical columns:

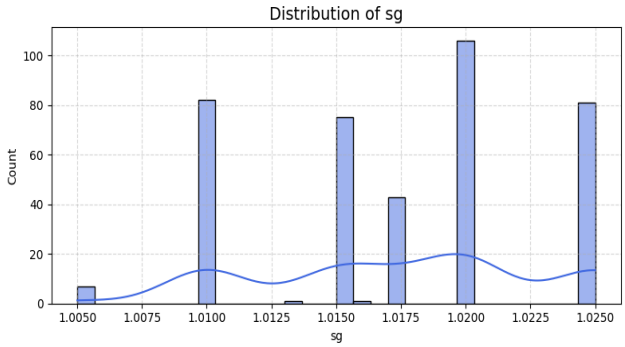
```
import os # Create a folder for plots if it doesn't exist
os.makedirs('plots', exist_ok=True)
```

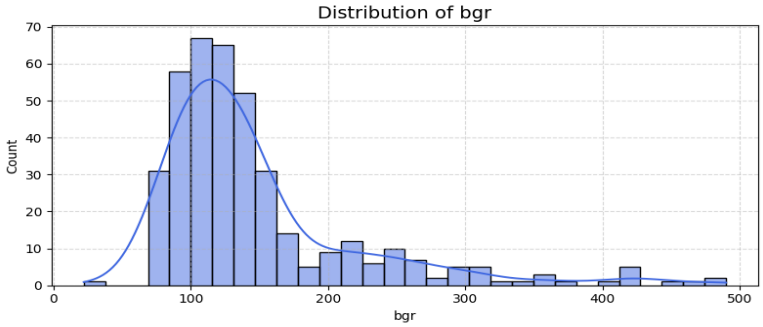
```
# List of numerical columns (excluding id if present)
numerical_cols = df_clean.select_dtypes(include=['float64', 'int64']).columns.tolist()
print(numerical_cols)
if 'id' in numerical_cols:
    numerical_cols.remove('id')

# Plot distributions
for col in numerical_cols:
    plt.figure(figsize=(8, 4))
    sns.histplot(df_clean[col], kde=True, color='royalblue', bins=30)
    plt.title(f'Distribution of {col}', fontsize=14)
    plt.xlabel(col)
    plt.ylabel('Count')
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.savefig(f'plots/hist_{col}.png')
    plt.show()
    # Interpretation: Look for skewness, multimodality, and outliers.
```

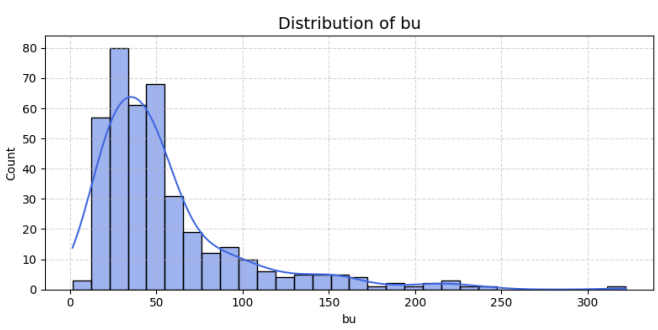
\\

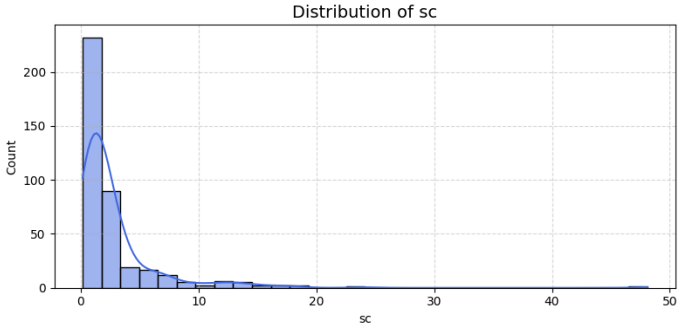


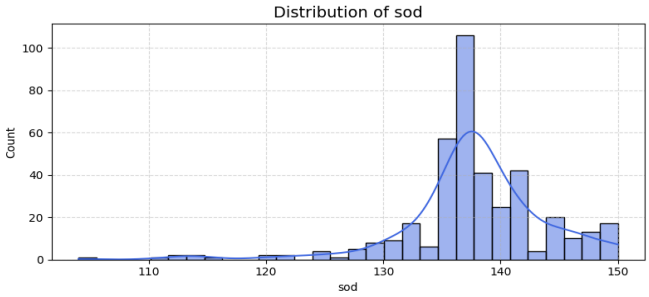


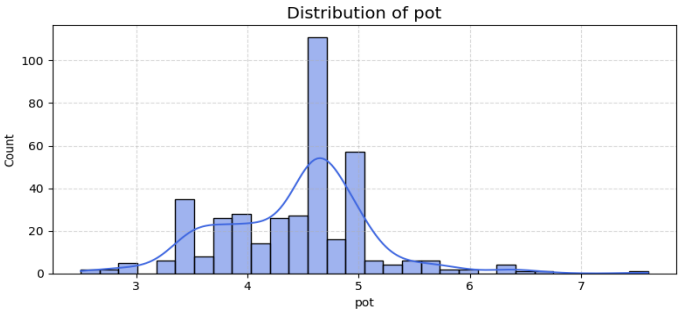


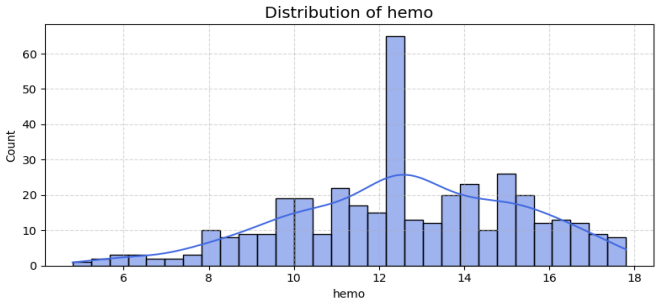


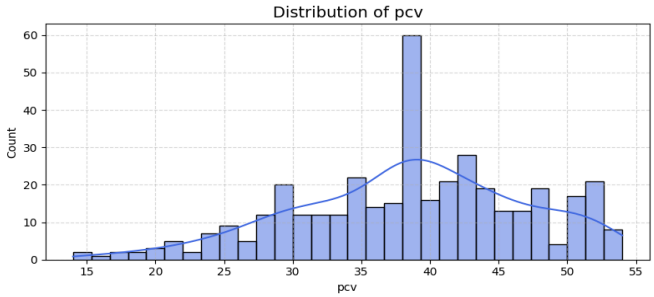

Distribution of bu
Count
bu

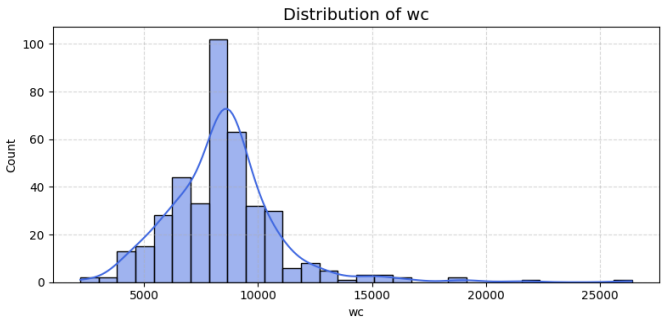

Distribution of sc
Count
sc

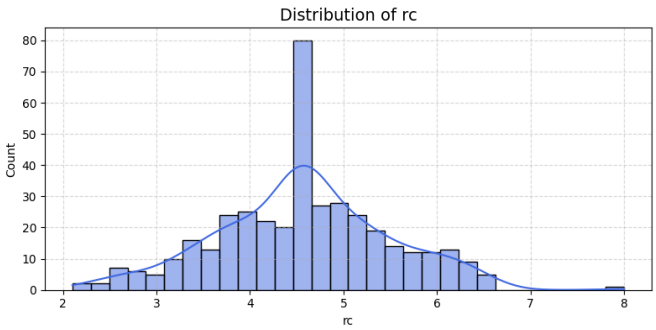

Distribution of sod
Count
sod


Distribution of pot
Count
pot


Distribution of hemo
Count
hemo

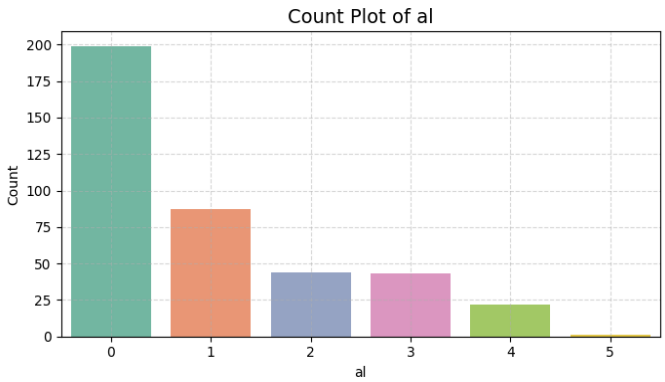

Distribution of pcv
Count
pcv

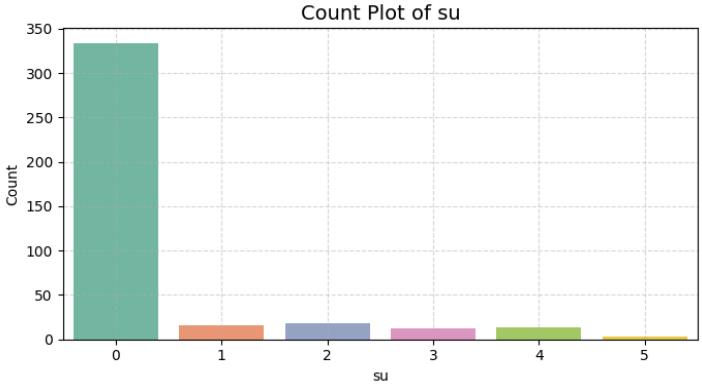

Distribution of wc
Count
wc

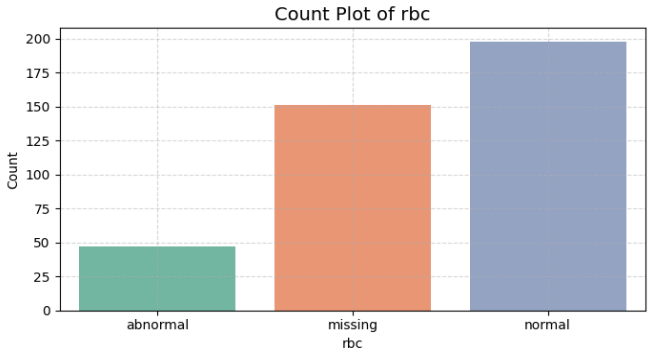

Distribution of rc
Count
rc

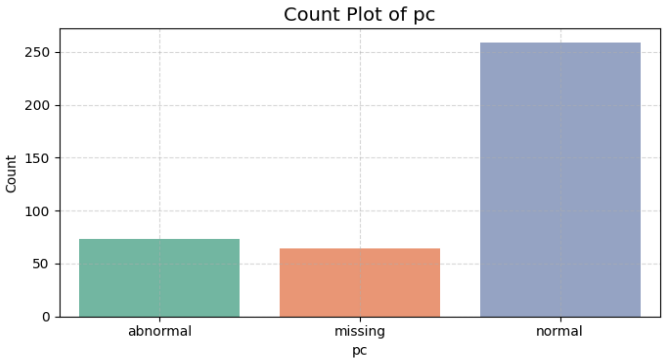
Count Plot for Categorical columns:

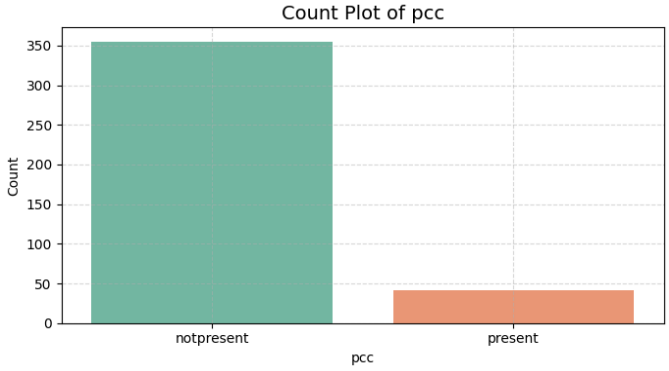
```
# List of categorical columns
cat_cols = df_clean.select_dtypes(include='category').columns.tolist()

for col in cat_cols:
    plt.figure(figsize=(7, 4))
    sns.countplot(x=col, data=df_clean, palette='Set2')
    plt.title(f'Count Plot of {col}', fontsize=14)
    plt.xlabel(col)
    plt.ylabel('Count')
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.savefig(f'plots/count_{col}.png')
    plt.show()
    # Interpretation: See class balance, rare categories, and missing category if present.
```

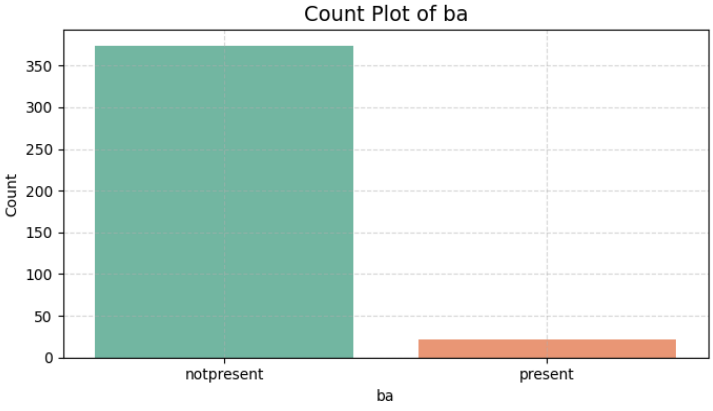


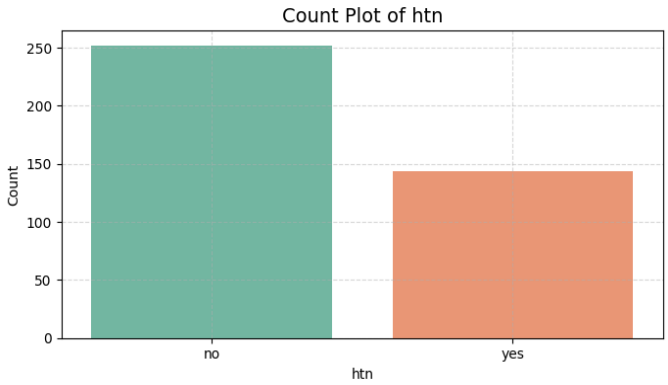


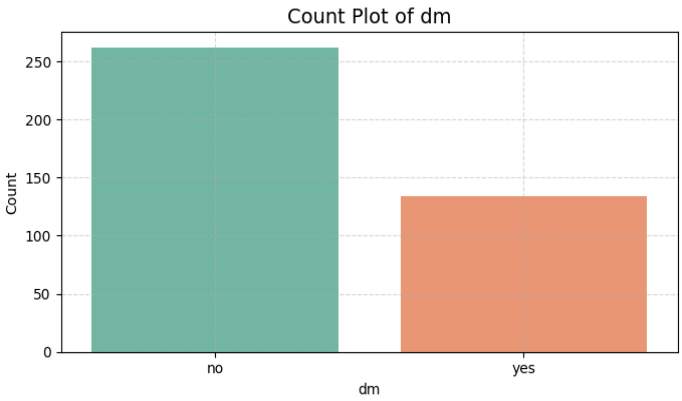


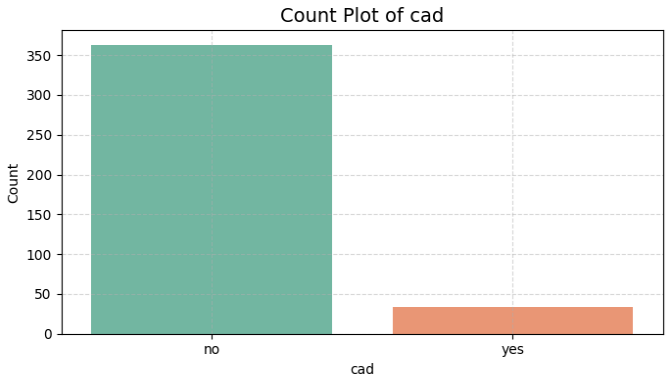


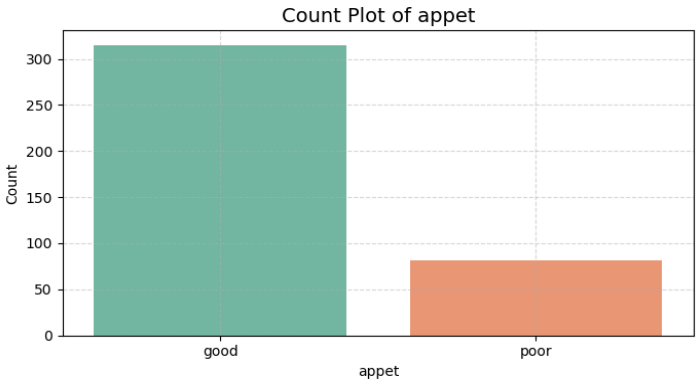


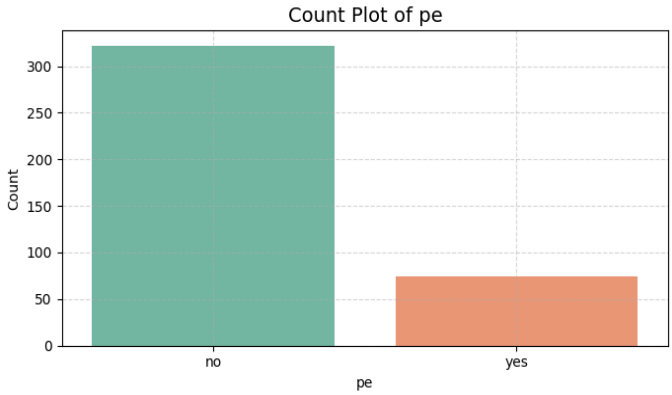


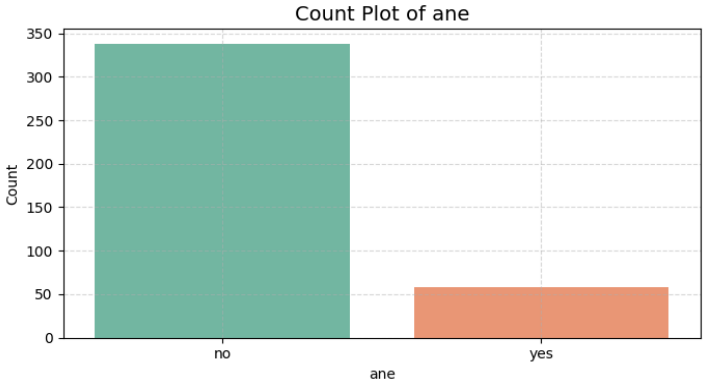








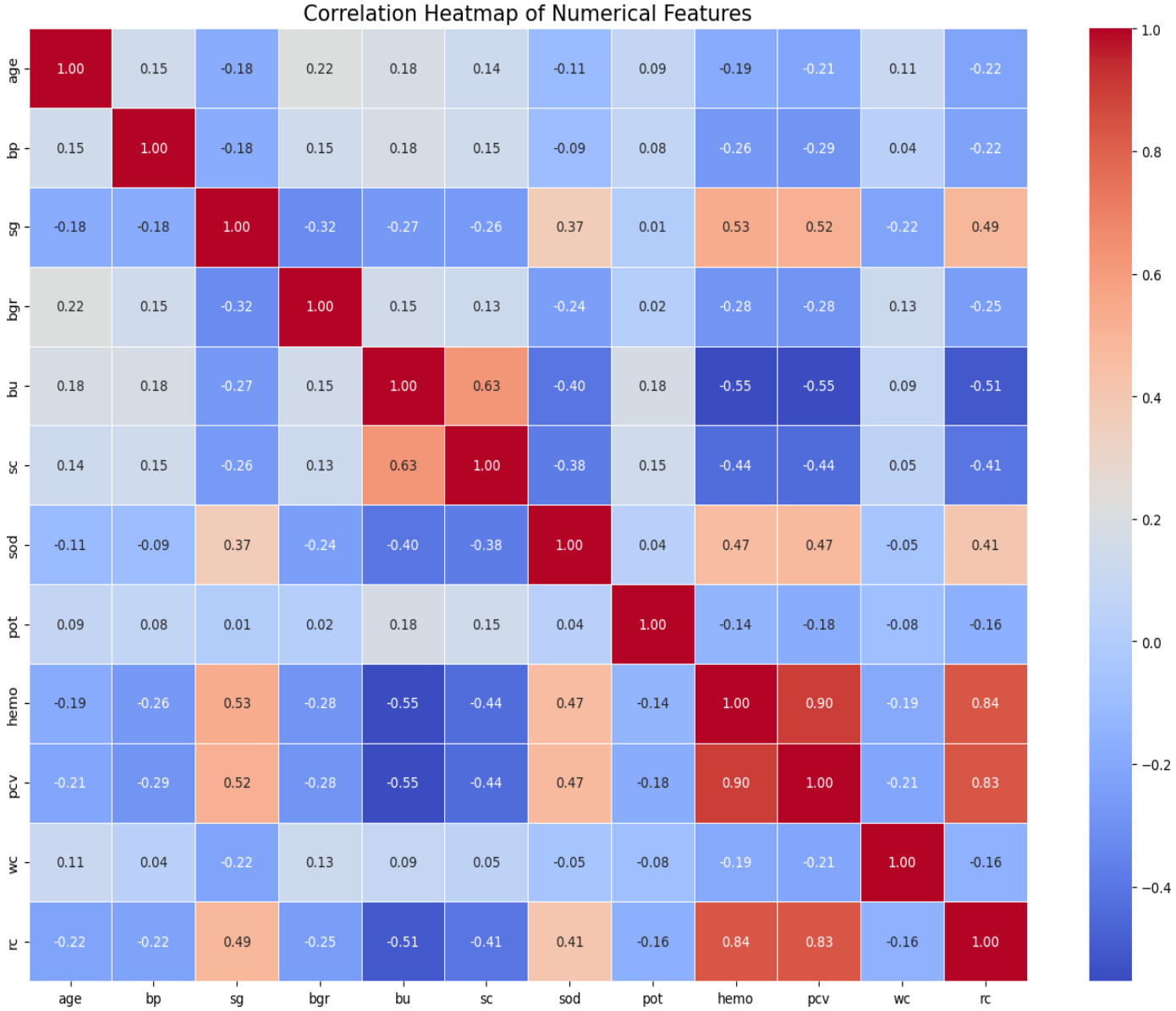




Activity 2.2: Bivariate analysis

Correlation Heatmap for Numerical Variables (Before Feature Engineered Columns Were Added):

```
plt.figure(figsize=(14, 10))
corr = df_clean[numerical_cols].corr()
sns.heatmap(corr, annot=True, fmt=".2f", cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap of Numerical Features', fontsize=16)
plt.tight_layout()
plt.savefig('plots/corr_heatmap.png')
plt.show()
# Interpretation: Look for strong positive/negative correlations, which may indicate redundancy or important relationships.
```

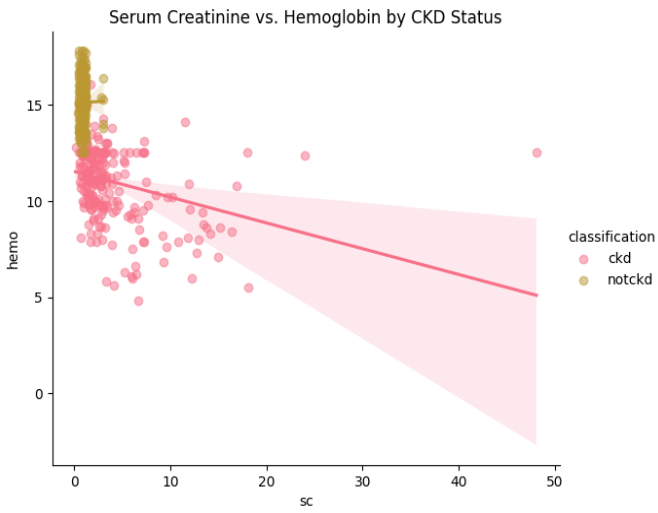

Correlation Heatmap of Numerical Features
age 1.00 0.15 -0.18 0.22 0.18 0.14 -0.11 0.09 -0.19 -0.21 0.11 -0.22
bp 0.15 1.00 -0.18 0.15 0.18 0.15 -0.09 0.08 -0.26 -0.29 0.04 -0.22
sg -0.18 -0.18 1.00 -0.32 -0.27 -0.26 0.37 0.01 0.53 0.52 -0.22 0.49
bgr 0.22 0.15 -0.32 1.00 0.15 0.13 -0.24 0.02 -0.28 -0.28 0.13 -0.25
bu 0.18 0.18 -0.27 0.15 1.00 0.63 -0.40 0.18 -0.55 -0.55 0.09 -0.51
sc 0.14 0.15 -0.26 0.13 0.63 1.00 -0.38 0.15 -0.44 -0.44 0.05 -0.41
sod -0.11 -0.09 0.37 -0.24 -0.40 -0.38 1.00 0.04 0.47 0.47 -0.05 0.41
pot 0.09 0.08 0.01 0.02 0.18 0.15 0.04 1.00 -0.14 -0.18 -0.08 -0.16
hemo -0.19 -0.26 0.53 -0.28 -0.55 -0.44 0.47 -0.14 1.00 0.90 -0.19 0.84
pcv -0.21 -0.29 0.52 -0.28 -0.55 -0.44 0.47 -0.18 0.90 1.00 -0.21 0.83
wc 0.11 0.04 -0.22 0.13 0.09 0.05 -0.05 -0.08 -0.19 -0.21 1.00 -0.16
rc -0.22 -0.22 0.49 -0.25 -0.51 -0.41 0.41 -0.16 0.84 0.83 -0.16 1.00
age bp sg bgr bu sc sod pot hemo pcv wc rc
1.0 0.8 0.6 0.4 0.2 0.0 −0.2 −0.4

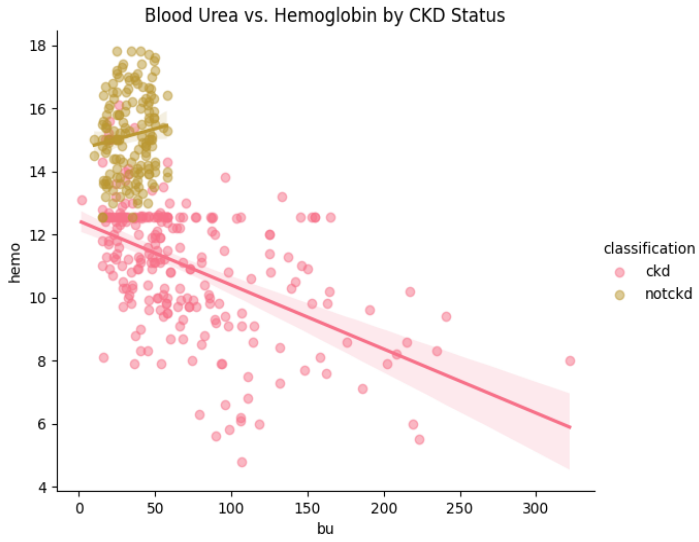
Activity 2.3: Multivariate analysis

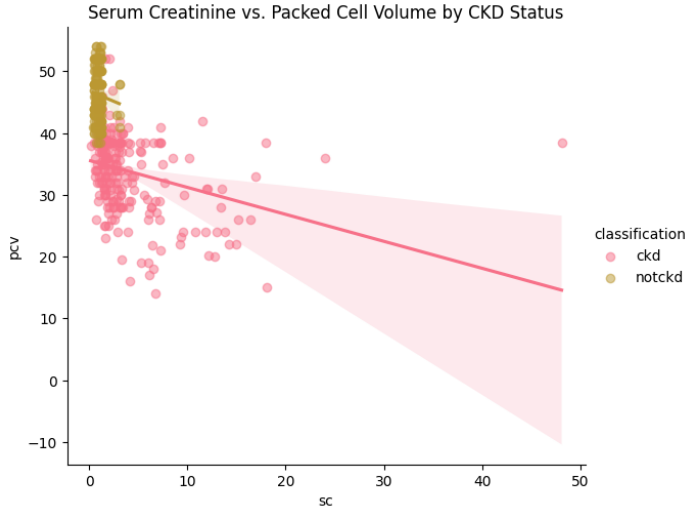
a. Kidney Function and Anemia

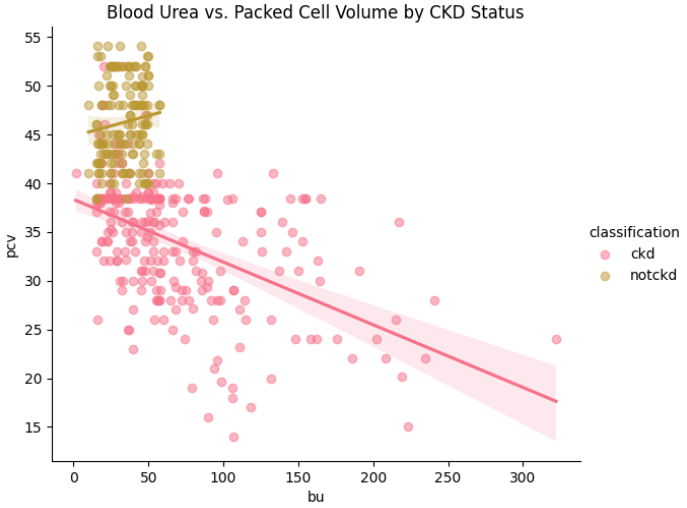
- SC, BU vs. Hemo, PCV, RC:
- These relationships are central to CKD progression (as kidney function declines, anemia worsens).

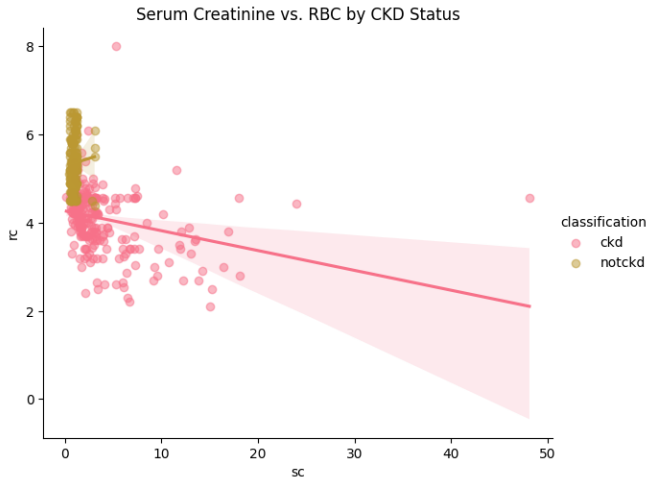
```
# Scatterplots with regression lines
sns.lmplot(x='sc', y='hemo', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Serum Creatinine vs. Hemoglobin by CKD Status')
plt.show()

sns.lmplot(x='bu', y='hemo', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Blood Urea vs. Hemoglobin by CKD Status')
plt.show()

sns.lmplot(x='sc', y='pcv', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Serum Creatinine vs. Packed Cell Volume by CKD Status')
plt.show()

sns.lmplot(x='bu', y='pcv', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Blood Urea vs. Packed Cell Volume by CKD Status')
plt.show()

sns.lmplot(x='sc', y='rc', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Serum Creatinine vs. RBC by CKD Status')
plt.show()

sns.lmplot(x='bu', y='rc', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Blood Urea vs. RBC by CKD Status')
plt.show()

sns.lmplot(x='sc', y='sod', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Serum Creatinine vs. Sodium by CKD Status')
plt.show()

sns.lmplot(x='bu', y='sod', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Blood Urea vs. Sodium by CKD Status')
plt.show()
```

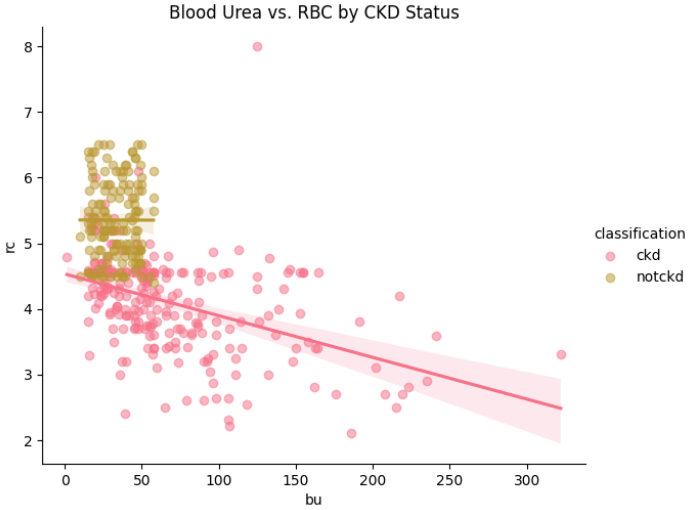


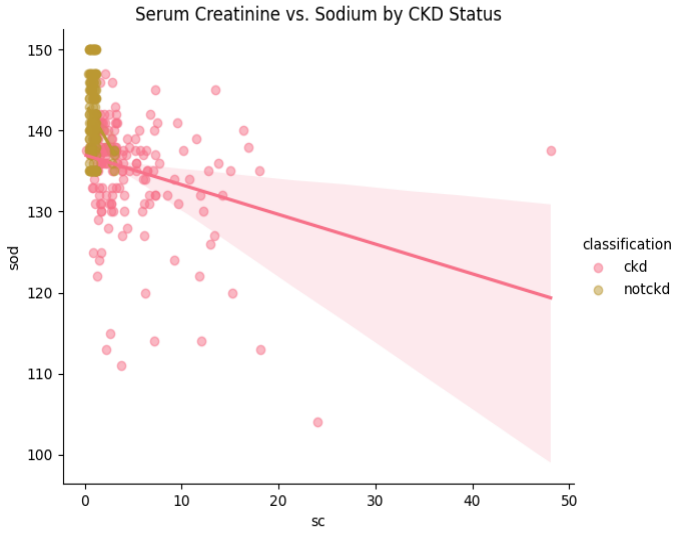


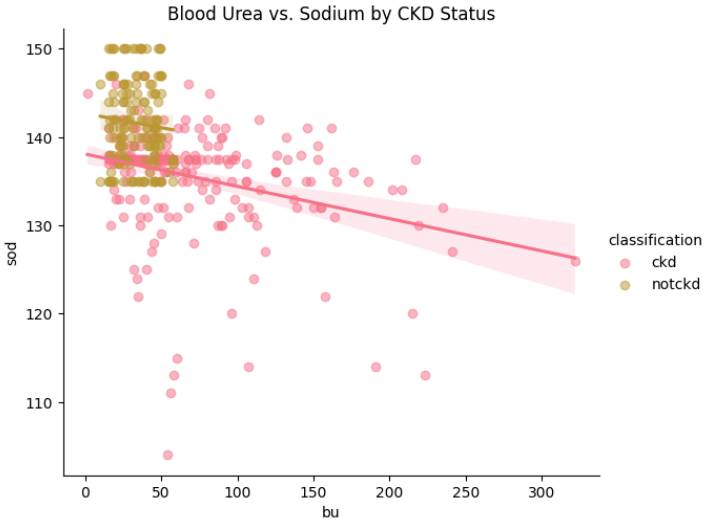








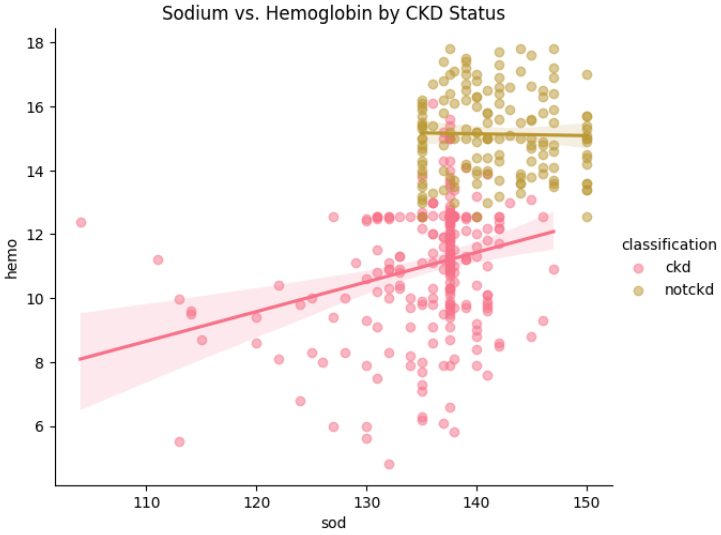




b. Sodium and Blood Health

- Sodium vs. Hemo/PCV/RC:
- Visualize how sodium imbalances relate to anemia.

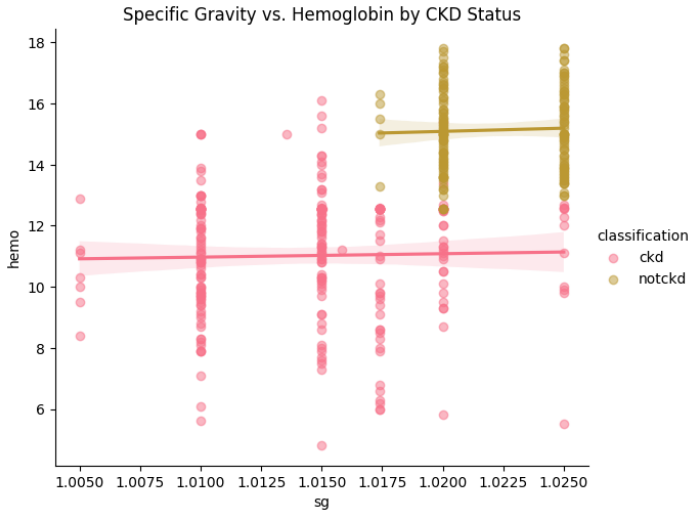
```
sns.lmplot(x='sod', y='hemo', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Sodium vs. Hemoglobin by CKD Status')
plt.show()
```



c. Specific Gravity and Blood Markers

- SG vs. Hemo/PCV/RC:
- Higher SG may indicate better kidney function and blood health.

```
sns.lmplot(x='sg', y='hemo', hue='classification', data=df_clean, aspect=1.2, height=5, scatter_kws={'alpha':0.5})
plt.title('Specific Gravity vs. Hemoglobin by CKD Status')
plt.show()
```



Splitting data into train and test:

```
from sklearn.model_selection import train_test_split

# Exclude target and categorical columns from features
target = 'classification'
feature_cols = [col for col in df_clean.columns if col not in [target, 'sc_bin', 'hemo_bin', 'bu_bin']]  # Exclude binned cols and target

X = df_clean[feature_cols]
y = df_clean[target]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

Feature Encoding:

Data Transformation Steps:

- Log-transformed highly skewed variables: bgr, bu, sc, wc, eGFR.
- Standard scaled all numerical features.
- One-hot encoded all categorical variables.
- All transformations were fit on the training set and applied to the test set to prevent data leakage.

One Hot Encoding:

```
[ ]  cat_cols = X_train.select_dtypes(include='category').columns.tolist()

     X_train = pd.get_dummies(X_train, columns=cat_cols, drop_first=True)
     X_test = pd.get_dummies(X_test, columns=cat_cols, drop_first=True)

     # Ensure columns match in train and test
     X_test = X_test.reindex(columns=X_train.columns, fill_value=0)
```

Scaling

Scaling is a technique used to transform the values of a dataset to a similar scale to improve the performance of machine learning algorithms. Scaling is important because many machine learning algorithms are sensitive to the scale of the input features. Here we are using Standard Scaler. This scales the data to have a mean of 0 and a standard deviation of 1. The formula is given by: X_scaled = (X - X_mean) / X_std

```
[ ] from sklearn.preprocessing import StandardScaler

    # Identify numerical columns (excluding categorical and binned)
    num_cols_imp1 = X_train_imp1.select_dtypes(include=['float64', 'int32']).columns.tolist()

    scaler_imp1 = StandardScaler()
    X_train_imp1[num_cols_imp1] = scaler_imp1.fit_transform(X_train_imp1[num_cols_imp1])
    X_test_imp1[num_cols_imp1] = scaler_imp1.transform(X_test_imp1[num_cols_imp1])
```

```
[ ] cat_cols_imp1 = X_train_imp1.select_dtypes(include='category').columns.tolist()

    X_train_imp1 = pd.get_dummies(X_train_imp1, columns=cat_cols_imp1, drop_first=True)
    X_test_imp1 = pd.get_dummies(X_test_imp1, columns=cat_cols_imp1, drop_first=True)

    # Ensure columns match in train and test
    X_test_imp1 = X_test_imp1.reindex(columns=X_train_imp1.columns, fill_value=0)
```

Label Encoding:

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y_train = le.fit_transform(y_train)
y_test = le.transform(y_test)

print(le.classes_)  # To see which label is 0 and which is 1

y_train = pd.Series(y_train)
y_test = pd.Series(y_test)
```

```
['ckd' 'notckd']
```

```
[ ] from sklearn.preprocessing import LabelEncoder

    le_imp1 = LabelEncoder()
    y_train_imp1 = le_imp1.fit_transform(y_train_imp1)
    y_test_imp1 = le_imp1.transform(y_test_imp1)

    print(le_imp1.classes_)  # To see which label is 0 and which is 1

    y_train_imp1 = pd.Series(y_train_imp1)
    y_test_imp1 = pd.Series(y_test_imp1)
```

```
['ckd' 'notckd']
```

```
log_transform_cols_imp1 = ['bgr','sc', 'eGFR']

for col in log_transform_cols_imp1:
    for df in [X_train_imp1, X_test_imp1]:
        df[col] = np.log(df[col])
```

Handling Imbalanced dataset

Imbalanced datasets are a common challenge in machine learning, where one class significantly outnumbers the other. This imbalance can lead to biased models that perform well on the majority class but poorly on the minority class. To address this, various resampling techniques such as SMOTE (Synthetic Minority Over-sampling Technique) are commonly used to balance the class distribution.

In this case, although the SMOTE technique was considered, the class distribution in the dataset was only slightly imbalanced. After evaluation, it was determined that the imbalance was not significant enough to adversely affect model performance. Therefore, no resampling was applied.

Milestone 4: Model Building

Activity 1: Training the model in multiple algorithms

Now our data is cleaned and it's time to build the model. We applied multiple classification algorithms on 3 different versions of the dataset to see which one would suit our requirements. We decided to keep the one with the topmost important features according to the feature importance graph and then we chose the topmost model focusing on low overfitting and high recall. The best model is saved based on its performance.

```
[ ]  results_imp1 = run_healthcare_ml_pipeline(X_train_imp1, X_test_imp1, y_train_imp1, y_test_imp1)
```

This project emphasizes high recall, as false negatives in medical applications can have serious consequences and must be minimized. At the same time, we control overfitting risk through strategies like performance gap monitoring and extreme probability suppression. The following function is a healthcare-focused machine learning pipeline that automatically preprocesses the target variable, builds and tunes multiple models, and detects overfitting risks. It evaluates models using critical metrics like recall and F1-score, ranks them based on a custom healthcare score, and identifies both the safest and riskiest models for clinical use. With built-in visualizations, hyperparameter insights, and validation curve analysis, it supports informed and trustworthy model selection—especially vital in sensitive domains like early kidney disease prediction.

To check for overfitting:

To check if the model overfits we examined them based on two different tests and based on those results verified whether our model was overfitting or not.

First, we enforced a gap threshold by ensuring that the absolute difference between the training and testing scores remained below 0.05, promoting model generalization and reducing the risk of the model performing well only on the training data.

Second, we implemented extreme probability suppression by limiting overly confident predictions, specifically, we ensured that less than 20% of the predicted probabilities fell below 0.1 or above 0.9. This helped prevent the model from being overly certain and improved the overall calibration of its predictions.

Code:

```
def run_healthcare_ml_pipeline_hyp(X_train, X_test, y_train, y_test):
    """
    Complete healthcare ML pipeline with enhanced hyperparameter tuning and overfitting detection
    """
    print(f"\n{'='*80}")
    print("HEALTHCARE ML ANALYSIS PIPELINE")
    print("Kidney Disease Detection - Enhanced with Hyperparameter Tuning")
    print(f"{'='*80}")
```

```
    # Convert target to binary if needed
    if hasattr(y_train, 'dtype') and y_train.dtype == 'object':
        unique_classes = np.unique(y_train)
        if len(unique_classes) == 2:
            y_train_processed = (y_train == unique_classes[0]).astype(int)
            y_test_processed = (y_test == unique_classes[0]).astype(int)
        else:
            from sklearn.preprocessing import LabelEncoder
            le = LabelEncoder()
            y_train_processed = le.fit_transform(y_train)
            y_test_processed = le.transform(y_test)
    else:

y_train_processed = y_train
y_test_processed = y_test

    print(f"   Dataset Info:")
    print(f"   Training samples: {X_train.shape[0]}")
    print(f"   Test samples: {X_test.shape[0]}")
    print(f"   Features: {X_train.shape[1]}")
    print(f"   Classes: {len(np.unique(y_train_processed))}")
    print(f"   Class distribution: {dict(zip(*np.unique(y_train_processed, return_counts=True)))}")

    # Create hyperparameter-tuned models
    models, tuning_summary = create_tuned_models_dict(X_train, y_train_processed)
    print(f"\nTesting {len(models)} different classification algorithms...")
    print(f"   {len(tuning_summary)} models have been hyperparameter-tuned")
    print(f"   {len(models) - len(tuning_summary)} baseline models included")

    # Print hyperparameter tuning summary
    if tuning_summary:
        print_hyperparameter_tuning_summary(tuning_summary)

    # Run comprehensive analysis with enhanced overfitting detection
    results_list, cv_results, overfitting_summary = detect_overfitting_comprehensive_enhanced(
        X_train, X_test, y_train_processed, y_test_processed, models
    )

    # Convert results list to dictionary for easier access
    results_dict = {}
    for result in results_list:
        model_name = result['Model']
        results_dict[model_name] = result
```

```
    print(f"\n{'='*80}")
    print("INDIVIDUAL MODEL ANALYSIS WITH VISUALIZATIONS")
    print(f"{'='*80}")

    # Dictionary to store trained models for plotting
    trained_models = {}

    # Individual model analysis with plotting
    for model_name, model_results in results_dict.items():
        if model_results is None:
            continue
        print(f"\n ANALYZING: {model_name}")
        print("-" * 60)



try:
            # Get model instance
            if model_name not in models:
                print(f"Model {model_name} not found in models dictionary, skipping...")
                continue

            model = models[model_name]

            # Train the model
            model.fit(X_train, y_train_processed)
            trained_models[model_name] = model

            # Make predictions
            y_pred = model.predict(X_test)

            # Get prediction probabilities if available
            if hasattr(model, 'predict_proba'):
                y_pred_proba = model.predict_proba(X_test)
                if len(np.unique(y_test_processed)) == 2:
                    y_pred_proba = y_pred_proba[:, 1]
                else:
                    y_pred_proba = y_pred_proba.max(axis=1)
            elif hasattr(model, 'decision_function'):
                y_pred_proba = model.decision_function(X_test)
                # Normalize decision function scores to [0,1] for binary classification
                if len(np.unique(y_test_processed)) == 2:
                    y_pred_proba = (y_pred_proba - y_pred_proba.min()) / (y_pred_proba.max() -
y_pred_proba.min())
            else:
                y_pred_proba = None
```

```
print(f"Model Performance:")
# Access metrics from model_results dictionary
print(f"   Test Accuracy: {model_results.get('Test Accuracy', 0):.4f}")
print(f"   Precision: {model_results.get('Precision', 0):.4f}")
print(f"   Recall: {model_results.get('Recall', 0):.4f}")
print(f"   F1 Score: {model_results.get('F1 Score', 0):.4f}")
print(f"   ROC AUC: {model_results.get('ROC AUC', 0):.4f}")

# Display hyperparameter tuning results if available
if model_name in tuning_summary:
    tuning_info = tuning_summary[model_name]
    print(f"\nHyperparameter Tuning Results:")

   print(f"  Best Recall Score (CV): {tuning_info['results']['best_score']:.4f}")
    print(f" Overfitting Risk: {tuning_info['results']['overfitting_risk']}")
    print(f" Overfitting Gap: {tuning_info['results']['overfitting_gap']:.4f}")
    print(f" Search Method: {tuning_info['results']['search_type']}")

    # Show key hyperparameters
    key_params = list(tuning_info['params'].items())[:3]  # Show first 3 params
    if key_params:
        print(f"Key Tuned Parameters:")
        for param, value in key_params:
            print(f"{param}: {value}")

# Plot confusion matrix
print(f"\nGenerating Confusion Matrix...")
plot_confusion_matrix(y_test_processed, y_pred, model_name)

# Plot ROC curve (only for binary classification)
if len(np.unique(y_test_processed)) == 2 and y_pred_proba is not None:
    print(f"Generating ROC Curve...")
    plot_roc_curve(y_test_processed, y_pred_proba, model_name)

    print(f"Generating Precision-Recall Curve...")
    plot_precision_recall_curve(y_test_processed, y_pred_proba, model_name)

# Plot feature importance (if available)
if hasattr(model, 'feature_importances_'):
    print(f"Generating Feature Importance Plot...")
    plot_feature_importance(model, X_train, model_name)
elif hasattr(model, 'coef_') and model.coef_.ndim == 1:
    print(f"Generating Feature Coefficients Plot...")
    # Handle linear model coefficients
```

```
            plt.figure(figsize=(10, 6))
            if hasattr(X_train, 'columns'):
                coef_series = pd.Series(np.abs(model.coef_), index=X_train.columns)
            else:
                coef_series = pd.Series(np.abs(model.coef_), index=[f'Feature_{i}' for i in
range(len(model.coef_))])
            coef_series.sort_values(ascending=False).head(10).plot(kind='bar')
            plt.title(f'Top 10 Feature Coefficients (Absolute) - {model_name}')
            plt.xticks(rotation=45, ha='right')
            plt.tight_layout()
            plt.show()

        # Validation curve analysis for selected models with hyperparameters

         print(f"Generating Validation Curve Analysis...")
        if model_name == 'Random Forest':
            validation_curve_analysis_enhanced(
                X_train, y_train_processed, model,
                'n_estimators', [10, 50, 100, 200, 300]
            )
        elif model_name == 'XGBoost':
            validation_curve_analysis_enhanced(
                X_train, y_train_processed, model,
                'max_depth', [3, 4, 5, 6, 7, 8]
            )
        elif model_name == 'LightGBM':
            validation_curve_analysis_enhanced(
                X_train, y_train_processed, model,
                'num_leaves', [10, 20, 30, 40, 50]
            )
        elif 'SVM' in model_name:
            validation_curve_analysis_enhanced(
                X_train, y_train_processed, model,
                'C', [0.1, 1, 10, 100, 1000]
            )
        elif 'Logistic Regression' in model_name:
            validation_curve_analysis_enhanced(
                X_train, y_train_processed, model,
                'C', [0.01, 0.1, 1, 10, 100]
            )

        print(f"Completed analysis for {model_name}\n")

    except Exception as e:
```

```
        print(f"Error analyzing {model_name}: {str(e)}")
        continue

    # Print overfitting summary
risk_groups = print_overfitting_summary(overfitting_summary)

# Convert results list to DataFrame for healthcare model selection
results_df = pd.DataFrame(results_list)

# Healthcare-specific model selection
best_model_name, ranked_models = healthcare_model_selection_algorithm(results_df)

# Generate comprehensive model comparison plots
print(f"\n{'='*80}")

print("COMPREHENSIVE MODEL COMPARISON VISUALIZATIONS")
print(f"{'='*80}")

print("Generating Top Models Comparison...")
plot_model_comparison(results_df)

# Enhanced hyperparameter tuning summary visualization
if tuning_summary:
    print(f"\nGenerating Hyperparameter Tuning Summary Visualization...")
    plot_hyperparameter_tuning_summary(tuning_summary)

# Final recommendations with hyperparameter considerations
print(f"\n{'='*80}")
print(" FINAL HEALTHCARE RECOMMENDATIONS")
print(f"{'='*80}")

print(f"RECOMMENDED MODEL: {best_model_name}")
best_stats = ranked_models.iloc[0]
print(f"   Healthcare Score: {best_stats['Healthcare_Score']:.4f}")
print(f"   Recall (Sensitivity): {best_stats['Recall']:.4f}")
print(f"   Precision: {best_stats['Precision']:.4f}")
print(f"   F1 Score: {best_stats['F1 Score']:.4f}")
print(f"   Overfitting Risk: {best_stats['Overfitting Risk']}")

# Show hyperparameter tuning info for best model if available
if best_model_name in tuning_summary:
    best_tuning = tuning_summary[best_model_name]
    print(f"   Hyperparameter Optimization:")
    print(f"   Tuning Method: {best_tuning['results']['search_type']}")
```

```
        print(f"   CV Recall Score: {best_tuning['results']['best_score']:.4f}")
        print(f"   Overfitting Gap: {best_tuning['results']['overfitting_gap']:.4f}")

    print(f"\ TOPn 3 SAFE MODELS FOR HEALTHCARE:")
    safe_models = ranked_models[ranked_models['Overfitting Risk'].isin(['LOW', 'MEDIUM'])].head(3)
    for i, (_, row) in enumerate(safe_models.iterrows(), 1):
        risk_indicator = "Low" if row['Overfitting Risk'] == 'LOW' else "high"
        tuning_indicator = " " if row['Model'] in tuning_summary else " "
        print(f"   {i}. {tuning_indicator} {row['Model']} (Score: {row['Healthcare_Score']:.4f}, Risk:
{risk_indicator}{row['Overfitting Risk']})")

    # Models to avoid with hyperparameter tuning context
    avoid_models = ranked_models[ranked_models['Overfitting Risk'].isin(['CRITICAL',
'HIGH'])]['Model'].tolist()
    if avoid_models:
        print(f"\nMODELS TO AVOID IN HEALTHCARE:")

     for model in avoid_models[:5]:  # Show top 5 to avoid
            if model in tuning_summary:
                gap = tuning_summary[model]['results']['overfitting_gap']
                print(f"    {model} (Overfitting Gap: {gap:.4f})")
            else:
                print(f"   {model}")

    # Hyperparameter tuning insights
    if tuning_summary:
        print(f"\nHYPERPARAMETER TUNING INSIGHTS:")

        # Count tuned models by risk level
        tuned_risks = {}
        for model_name, info in tuning_summary.items():
            risk = info['results']['overfitting_risk']
            tuned_risks[risk] = tuned_risks.get(risk, 0) + 1

        print(f"  Tuned Models by Risk Level:")
        risk_order = ['LOW', 'MEDIUM', 'HIGH', 'CRITICAL']
        for risk in risk_order:
            if risk in tuned_risks:
                icon = {'LOW': 'l', 'MEDIUM': 'm', 'HIGH': 'h', 'CRITICAL': 'c'}.get(risk, 'r')
                print(f"      {icon} {risk}: {tuned_risks[risk]} models")

        # Best tuning results
        best_tuned_recall = max(tuning_summary.items(), key=lambda x: x[1]['results']['best_score'])
        lowest_overfitting_tuned = min(tuning_summary.items(), key=lambda x: x[1]['results']['overfitting_gap'])
```

```
    print(f"   Best Tuned Recall: {best_tuned_recall[0]} ({best_tuned_recall[1]['results']['best_score']:.4f})")
    print(f"    Lowest Overfitting (Tuned): {lowest_overfitting_tuned[0]} (Gap:
{lowest_overfitting_tuned[1]['results']['overfitting_gap']:.4f})")

  # Additional comprehensive analysis plots
  print(f"\nGENERATING ADDITIONAL ANALYSIS PLOTS...")

  # Overfitting risk distribution plot
  plt.figure(figsize=(15, 10))

  # Risk distribution
  plt.subplot(2, 3, 1)
  risk_counts = ranked_models['Overfitting Risk'].value_counts()
  colors = {'LOW': 'green', 'MEDIUM': 'orange', 'HIGH': 'red', 'CRITICAL': 'darkred'}
  risk_colors = [colors.get(risk, 'gray') for risk in risk_counts.index]
  plt.pie(risk_counts.values, labels=risk_counts.index, autopct='%1.1f%%', colors=risk_colors)

 plt.title('Overfitting Risk Distribution')

  # Healthcare scores distribution
  plt.subplot(2, 3, 2)
  plt.hist(ranked_models['Healthcare_Score'], bins=15, alpha=0.7, color='skyblue', edgecolor='black')
  plt.xlabel('Healthcare Score')
  plt.ylabel('Number of Models')
  plt.title('Healthcare Scores Distribution')
  plt.grid(True, alpha=0.3)

  # Recall vs Precision scatter plot
  plt.subplot(2, 3, 3)
  colors_risk = ranked_models['Overfitting Risk'].map(colors)
  scatter = plt.scatter(ranked_models['Recall'], ranked_models['Precision'],
                        c=colors_risk, alpha=0.7, s=60, edgecolors='black', linewidth=0.5)
  plt.xlabel('Recall (Sensitivity)')
  plt.ylabel('Precision')
  plt.title('Recall vs Precision (Colored by Risk)')
  plt.grid(True, alpha=0.3)

  # F1 Score vs CV Stability
  plt.subplot(2, 3, 4)
  plt.scatter(ranked_models['F1 Score'], ranked_models['CV Std F1'],
              c=colors_risk, alpha=0.7, s=60, edgecolors='black', linewidth=0.5)
  plt.xlabel('F1 Score')
  plt.ylabel('CV Standard Deviation')
  plt.title('Performance vs Stability (Colored by Risk)')
```

```
plt.grid(True, alpha=0.3)

# Hyperparameter tuning comparison (if available)
if tuning_summary:
    plt.subplot(2, 3, 5)
    tuned_models_data = []
    tuned_scores = []
    tuned_gaps = []
    for model_name in ranked_models['Model']:
        if model_name in tuning_summary:
            tuned_models_data.append(model_name[:15])  # Truncate long names
            tuned_scores.append(tuning_summary[model_name]['results']['best_score'])
            tuned_gaps.append(tuning_summary[model_name]['results']['overfitting_gap'])

    if tuned_models_data:
        plt.scatter(tuned_scores, tuned_gaps, alpha=0.7, s=60,
                    c='purple', edgecolors='black', linewidth=0.5)

        plt.xlabel('Tuned CV Recall Score')
        plt.ylabel('Overfitting Gap')
        plt.title('Hyperparameter Tuning Results')
        plt.grid(True, alpha=0.3)

        # Add model names as annotations for top performers
        for i, (score, gap, name) in enumerate(zip(tuned_scores, tuned_gaps, tuned_models_data)):
            if score > np.percentile(tuned_scores, 75) and gap < np.percentile(tuned_gaps, 50):
                plt.annotate(name, (score, gap), xytext=(5, 5),
                             textcoords='offset points', fontsize=8)

# Model complexity vs performance
plt.subplot(2, 3, 6)
# Create a complexity score based on model type
complexity_map = {
    'Dummy': 1, 'Naive Bayes': 2, 'Logistic Regression': 3, 'LDA': 3, 'QDA': 4,
    'Decision Tree': 4, 'KNN': 4, 'SVM': 5, 'Random Forest': 6, 'Extra Trees': 6,
    'AdaBoost': 6, 'Gradient Boosting': 7, 'XGBoost': 8, 'LightGBM': 8, 'CatBoost': 8,
    'MLP': 9, 'SGD': 3, 'Ridge': 3, 'Bagging': 5
}

complexity_scores = []
for model_name in ranked_models['Model']:
    complexity = 5  # default
    for key, value in complexity_map.items():
        if key.lower() in model_name.lower():
```

```
            complexity = value
            break
    complexity_scores.append(complexity)

plt.scatter(complexity_scores, ranked_models['Healthcare_Score'],
           c=colors_risk, alpha=0.7, s=60, edgecolors='black', linewidth=0.5)
plt.xlabel('Model Complexity')
plt.ylabel('Healthcare Score')
plt.title('Complexity vs Healthcare Performance')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Summary statistics
print(f"\n SUMMARY STATISTICS:")
print(f"   Total Models Evaluated: {len(results_list)}")
print(f"   Models with Hyperparameter Tuning: {len(tuning_summary)}")

print(f"   Low Risk Models: {len(ranked_models[ranked_models['Overfitting Risk'] == 'LOW'])}")
print(f"   Medium Risk Models: {len(ranked_models[ranked_models['Overfitting Risk'] == 'MEDIUM'])}")
print(f"   High Risk Models: {len(ranked_models[ranked_models['Overfitting Risk'] == 'HIGH'])}")
print(f"   Critical Risk Models: {len(ranked_models[ranked_models['Overfitting Risk'] == 'CRITICAL'])}")
print(f"   Average Healthcare Score: {ranked_models['Healthcare_Score'].mean():.4f}")
print(f"   Average Recall: {ranked_models['Recall'].mean():.4f}")
print(f"   Average Precision: {ranked_models['Precision'].mean():.4f}")
return {
    'results': results_dict,
    'results_list': results_list,
    'cv_results': cv_results,
    'overfitting_summary': overfitting_summary,
    'risk_groups': risk_groups,
    'best_model': best_model_name,
    'ranked_models': ranked_models,
    'safe_models': safe_models,
    'trained_models': trained_models,
    'tuning_summary': tuning_summary,
    'hyperparameter_insights': {
        'tuned_models_count': len(tuning_summary),
        'best_tuned_recall': best_tuned_recall if tuning_summary else None,
        'lowest_overfitting_tuned': lowest_overfitting_tuned if tuning_summary else None,
        'risk_distribution': tuned_risks if tuning_summary else None
    }
}
```

Milestone 5: Performance Testing & Hyperparameter Tuning

Activity 1: Testing model with multiple evaluation metrics

Multiple evaluation metrics means evaluating the model's performance on a test set using different performance measures. This can provide a more comprehensive understanding of the model's strengths and weaknesses. We are using evaluation metrics for classification tasks including accuracy, precision, recall, support and F1-score.

The below function was used to calculate the comprehensive metrics for each model.

```python
def calculate_comprehensive_metrics(y_true, y_pred, y_pred_proba):
    """
    Calculate all classification metrics
    """
    return {
        'Test Accuracy': accuracy_score(y_true, y_pred),
        'Precision': precision_score(y_true, y_pred, average='weighted', zero_division=0),
        'Recall': recall_score(y_true, y_pred, average='weighted', zero_division=0),
        'F1 Score': f1_score(y_true, y_pred, average='weighted', zero_division=0),
        'ROC AUC': roc_auc_score(y_true, y_pred_proba) if len(np.unique(y_true)) == 2 else 0,
        'PR AUC': average_precision_score(y_true, y_pred_proba) if len(np.unique(y_true)) == 2 else 0
    }
```

Activity 1.1: Compare the model

Model 1: Random Forest

	precision	recall	f1-score	support
0	1.00	0.88	0.94	50
1	0.83	1.00	0.91	30
accuracy			0.93	80
macro avg	0.92	0.94	0.92	80
weighted avg	0.94	0.93	0.93	80

```
Test Accuracy: 0.9250
Precision: 0.9375
Recall: 0.9250
F1 Score: 0.9260
ROC AUC: 0.9933
PR AUC: 0.9889
```

Model 2: XGBoost

	precision	recall	f1-score	support
0	0.96	0.98	0.97	50
1	0.97	0.93	0.95	30
accuracy			0.96	80
macro avg	0.96	0.96	0.96	80
weighted avg	0.96	0.96	0.96	80

```
Test Accuracy: 0.9625
Precision: 0.9626
Recall: 0.9625
F1 Score: 0.9624
ROC AUC: 0.9933
PR AUC: 0.9905
```

Model 3: LightGBM

	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	1.00	1.00	1.00	30
accuracy			1.00	80
macro avg	1.00	1.00	1.00	80
weighted avg	1.00	1.00	1.00	80

Test Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000
PR AUC: 1.0000

Model 4: Logistic Regression

	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	1.00	1.00	1.00	30
accuracy			1.00	80
macro avg	1.00	1.00	1.00	80
weighted avg	1.00	1.00	1.00	80

Test Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000
PR AUC: 1.0000

Model 5: SVM(RBF)

	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	1.00	1.00	1.00	30
accuracy			1.00	80
macro avg	1.00	1.00	1.00	80
weighted avg	1.00	1.00	1.00	80

Test Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000
PR AUC: 1.0000

Model 6: Gradient Boosting

	precision	recall	f1-score	support
0	0.96	0.96	0.96	50
1	0.93	0.93	0.93	30
accuracy			0.95	80
macro avg	0.95	0.95	0.95	80
weighted avg	0.95	0.95	0.95	80

Test Accuracy: 0.9500
Precision: 0.9500
Recall: 0.9500
F1 Score: 0.9500
ROC AUC: 0.9947
PR AUC: 0.9920

Model 7: CatBoost

	precision	recall	f1-score	support
0	1.00	0.94	0.97	50
1	0.91	1.00	0.95	30
accuracy			0.96	80
macro avg	0.95	0.97	0.96	80
weighted avg	0.97	0.96	0.96	80

Test Accuracy: 0.9625
Precision: 0.9659
Recall: 0.9625
F1 Score: 0.9628
ROC AUC: 0.9973
PR AUC: 0.9958

Model 8: Decision Tree

	precision	recall	f1-score	support
0	0.95	0.84	0.89	50
1	0.78	0.93	0.85	30
accuracy			0.88	80
macro avg	0.87	0.89	0.87	80
weighted avg	0.89	0.88	0.88	80

Test Accuracy: 0.8750
Precision: 0.8883
Recall: 0.8750
F1 Score: 0.8767
ROC AUC: 0.8867
PR AUC: 0.7509

Model 9: KNN

	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	1.00	1.00	1.00	30
accuracy			1.00	80
macro avg	1.00	1.00	1.00	80
weighted avg	1.00	1.00	1.00	80

Test Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000
PR AUC: 1.0000

Model 10: Linear Discriminant Analysis

	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	1.00	1.00	1.00	30
accuracy			1.00	80
macro avg	1.00	1.00	1.00	80
weighted avg	1.00	1.00	1.00	80

Test Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000
PR AUC: 1.0000

Model 11: AdaBoost

	precision	recall	f1-score	support
0	0.95	0.84	0.89	50
1	0.78	0.93	0.85	30
accuracy			0.88	80
macro avg	0.87	0.89	0.87	80
weighted avg	0.89	0.88	0.88	80

Test Accuracy: 0.8750
Precision: 0.8883
Recall: 0.8750
F1 Score: 0.8767
ROC AUC: 0.8867
PR AUC: 0.7509

Model 12: SGD Classifier

	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	1.00	1.00	1.00	30
accuracy			1.00	80
macro avg	1.00	1.00	1.00	80
weighted avg	1.00	1.00	1.00	80

Test Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000
PR AUC: 1.0000

Model 13: Gaussian Naive Bayes

	precision	recall	f1-score	support
0	1.00	0.84	0.91	50
1	0.79	1.00	0.88	30
accuracy			0.90	80
macro avg	0.89	0.92	0.90	80
weighted avg	0.92	0.90	0.90	80

```
Test Accuracy: 0.9000
Precision: 0.9211
Recall: 0.9000
F1 Score: 0.9015
ROC AUC: 1.0000
PR AUC: 1.0000
```

Model 14: Bernoulli Naive Bayes

	precision	recall	f1-score	support
0	1.00	0.96	0.98	50
1	0.94	1.00	0.97	30
accuracy			0.97	80
macro avg	0.97	0.98	0.97	80
weighted avg	0.98	0.97	0.98	80

```
Test Accuracy: 0.9750
Precision: 0.9766
Recall: 0.9750
F1 Score: 0.9751
ROC AUC: 1.0000
PR AUC: 1.0000
```

Activity 2: Hyperparameter tuning for all the models

Random Forest :

```
'Random Forest': {
    'model': RandomForestClassifier(random_state=42, class_weight='balanced'),
    'params': {
        'n_estimators': [10, 25, 50, 100],
        'max_depth': [3, 5, 7, 10, None],
        'min_samples_split': [20, 40, 60, 80],
        'min_samples_leaf': [10, 20, 30, 40],
        'max_features': ['sqrt', 'log2', 0.3, 0.5],
        'max_samples': [0.6, 0.7, 0.8, 0.9],
        'min_impurity_decrease': [0.0, 0.01, 0.02, 0.05],
        'ccp_alpha': [0.0, 0.01, 0.02, 0.05]
    }
},
```

```
BEST HYPERPARAMETERS:
  n_estimators: 10
  min_samples_split: 80
  min_samples_leaf: 40
  min_impurity_decrease: 0.05
  max_samples: 0.6
  max_features: 0.3
  max_depth: 7
  ccp_alpha: 0.05
```

Decision Tree:

```
'Decision Tree': {
    'model': DecisionTreeClassifier(random_state=42, class_weight='balanced'),
    'params': {
        'max_depth': [3, 5, 7, 10],
        'min_samples_split': [40, 60, 80, 100],
        'min_samples_leaf': [20, 30, 40, 50],
        'max_features': ['sqrt', 'log2', 0.3, 0.5, None],
        'min_impurity_decrease': [0.01, 0.02, 0.05, 0.1],
        'ccp_alpha': [0.01, 0.02, 0.05, 0.1, 0.2]
    }
},
```

```
BEST HYPERPARAMETERS:
  min_samples_split: 40
  min_samples_leaf: 40
  min_impurity_decrease: 0.05
  max_features: None
  max_depth: 10
  ccp_alpha: 0.05
```

Gradient Boosting:

```
'Gradient Boosting': {
    'model': GradientBoostingClassifier(random_state=42),
    'params': {
        'n_estimators': [10, 25, 50, 100],
        'learning_rate': [0.01, 0.05, 0.1, 0.2],
        'max_depth': [2, 3, 4, 5],
        'min_samples_split': [40, 60, 80],
        'min_samples_leaf': [20, 30, 40],
        'subsample': [0.6, 0.7, 0.8, 0.9],
        'max_features': ['sqrt', 'log2', 0.3, 0.5],
        'min_impurity_decrease': [0.01, 0.02, 0.05],
        'ccp_alpha': [0.0, 0.01, 0.02]
    }
},
```

```
BEST HYPERPARAMETERS:
  subsample: 0.6
  n_estimators: 100
  min_samples_split: 80
  min_samples_leaf: 40
  min_impurity_decrease: 0.05
  max_features: log2
  max_depth: 2
  learning_rate: 0.01
  ccp_alpha: 0.0
```

XGBoost:

```
'XGBoost': {
    'model': xgb.XGBClassifier(random_state=42, eval_metric='logloss'),
    'params': {
        'n_estimators': [50, 100,150,200],
        'learning_rate': [0.1, 0.15, 0.2, 0.3],
        'max_depth': [5,6,7,8],
        'min_child_weight': [ 3,5,7, 10],
        'subsample': [0.7, 0.8, 0.9, 1],
        'colsample_bytree': [0.6, 0.7, 0.8, 0.9],
        'reg_alpha': [0, 0.01, 0.1, 1],
        'reg_lambda': [0.1, 1, 5, 10],
        'gamma': [0, 0.1, 0.5, 1],
        'scale_pos_weight': [1, 2, 3]  # Handle class imbalance
    }
},
```

```
BEST HYPERPARAMETERS:
  subsample: 0.9
  scale_pos_weight: 1
  reg_lambda: 10
  reg_alpha: 0
  n_estimators: 200
  min_child_weight: 7
  max_depth: 8
  learning_rate: 0.15
  gamma: 0.5
  colsample_bytree: 0.6
```

LightGBM

```
'LightGBM': {
    'model': lgb.LGBMClassifier(random_state=42, verbose=-1),
    'params': {
        'n_estimators': [10, 25, 50, 100],
        'learning_rate': [0.01, 0.05, 0.1, 0.2],
        'max_depth': [2, 3, 4, 5],
        'num_leaves': [7, 15, 31, 63],
        'min_child_samples': [10, 20, 30, 40],
        'min_split_gain': [0.01, 0.1, 0.5, 1],
        'subsample': [0.6, 0.7, 0.8, 0.9],
        'colsample_bytree': [0.3, 0.5, 0.7, 0.9],
        'reg_alpha': [0, 0.01, 0.1, 1],
        'reg_lambda': [0.1, 1, 5, 10],
        'min_child_weight': [0.001, 0.01, 0.1, 1],
        'class_weight': ['balanced', None]
    }
},
```

```
BEST HYPERPARAMETERS:
  subsample: 0.6
  reg_lambda: 0.1
  reg_alpha: 0.01
  num_leaves: 15
  n_estimators: 10
  min_split_gain: 1
  min_child_weight: 0.001
  min_child_samples: 30
  max_depth: 2
  learning_rate: 0.2
  colsample_bytree: 0.5
  class_weight: balanced
```

CatBoost

```
'CatBoost': {
    'model': CatBoostClassifier(random_state=42, verbose=False),
    'params': {
        'iterations': [10, 25, 50, 100],
        'learning_rate': [0.01, 0.05, 0.1, 0.2],
        'depth': [2, 3, 4, 5],
        'min_data_in_leaf': [10, 20, 30, 40],
        'l2_leaf_reg': [1, 3, 5, 10, 20],
        'subsample': [0.6, 0.7, 0.8, 0.9],
        'colsample_bylevel': [0.3, 0.5, 0.7, 0.9],
        'border_count': [32, 64, 128],
        'bagging_temperature': [0, 0.5, 1],
        'class_weights': [[1, 1], [1, 2], [1, 3]]  # Handle imbalance
    }
},
```

```
BEST HYPERPARAMETERS:
  subsample: 0.6
  min_data_in_leaf: 40
  learning_rate: 0.2
  l2_leaf_reg: 5
  iterations: 10
  depth: 4
  colsample_bylevel: 0.5
  class_weights: [1, 3]
  border_count: 64
  bagging_temperature: 0
```

AdaBoost

```
# ADABOOST
'AdaBoost': {
    'model': AdaBoostClassifier(random_state=42),
    'params': {
        'n_estimators': [10, 25, 50, 100],
        'learning_rate': [0.01, 0.1, 0.5, 1.0, 2.0],
        'algorithm': ['SAMME', 'SAMME.R']
    }
},
```

```
BEST HYPERPARAMETERS:
  algorithm: SAMME
  learning_rate: 0.01
  n_estimators: 10
```

Logistic Regression

```
'Logistic Regression': {
    'model': LogisticRegression(random_state=42, max_iter=1000, class_weight='balanced'),
    'params': {
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'penalty': ['l1', 'l2', 'elasticnet'],
        'solver': ['liblinear', 'saga'],
        'l1_ratio': [0.1, 0.3, 0.5, 0.7, 0.9],
        'fit_intercept': [True, False]
    }
},
```

```
BEST HYPERPARAMETERS:
  solver: liblinear
  penalty: l2
  l1_ratio: 0.5
  fit_intercept: False
  C: 1
```

SGD Classifier

```
'SGD Classifier': {
    'model': SGDClassifier(random_state=42, max_iter=1000, class_weight='balanced'),
    'params': {
        'alpha': [0.0001, 0.001, 0.01, 0.1, 1],
        'penalty': ['l1', 'l2', 'elasticnet'],
        'l1_ratio': [0.1, 0.3, 0.5, 0.7, 0.9],
        'learning_rate': ['constant', 'optimal', 'invscaling', 'adaptive'],
        'eta0': [0.001, 0.01, 0.1, 1],
        'early_stopping': [True, False],
        'validation_fraction': [0.1, 0.2, 0.3]
    }
},
```

```
BEST HYPERPARAMETERS:
  validation_fraction: 0.3
  penalty: l1
  learning_rate: constant
  l1_ratio: 0.3
  eta0: 0.01
  early_stopping: True
  alpha: 0.01
```

SVM(RBF)

```
'SVM (RBF)': {
    'model': SVC(random_state=42, probability=True, class_weight='balanced'),
    'params': {
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1],
        'kernel': ['rbf'],
        'shrinking': [True, False],
        'cache_size': [200, 500, 1000]
    }
},
```

```
BEST HYPERPARAMETERS:
  shrinking: False
  kernel: rbf
  gamma: 0.1
  cache_size: 500
  C: 100
```

K Nearest Neighbours

```
'K-Nearest Neighbors': {
    'model': KNeighborsClassifier(),
    'params': {
        'n_neighbors': [3, 5, 7, 9, 11, 15, 21, 31],
        'weights': ['uniform', 'distance'],
        'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
        'metric': ['euclidean', 'manhattan', 'minkowski'],
        'p': [1, 2, 3],
        'leaf_size': [10, 20, 30, 40, 50]
    }
},
```

```
BEST HYPERPARAMETERS:
  weights: distance
  p: 2
  n_neighbors: 21
  metric: euclidean
  leaf_size: 40
  algorithm: auto
```

Linear Discriminant Analysis

```
'Linear Discriminant Analysis': {
    'model': LinearDiscriminantAnalysis(),
    'params': {
        'solver': ['svd', 'lsqr', 'eigen'],
        'shrinkage': [None, 'auto', 0.1, 0.3, 0.5, 0.7, 0.9],
        'priors': [None],
        'n_components': [None, 1, 2, 3]
    }
},
```

```
BEST HYPERPARAMETERS:
  n_components: None
  priors: None
  shrinkage: 0.3
  solver: lsqr
```

To find the best models, we looked for the ones with low risk of overfitting and carefully noted the specific hyperparameters that had been tuned for them so we could recreate and retrain them consistently. After rebuilding the models using those same hyperparameters, we trained them on the original dataset to reproduce the same performance results.

Next, we tested these chosen models on a newly created unseen dataset we made. We ensured that the dataset was filled with valid data. Among all models tested in Version 1, which included both the original selected features and feature-engineered ones, the ***CatBoost***

model performed the best. It was chosen as the top model based on its ability to generalize well (low overfitting) and its high recall, which is especially important in healthcare applications where minimizing false negatives is critical.

```
Training Random Forest...
✓ Random Forest - Accuracy: 0.8400 (84.00%)

Training XGBoost...
✓ XGBoost - Accuracy: 0.9600 (96.00%)

Training Gradient Boosting...
✓ Gradient Boosting - Accuracy: 0.9000 (90.00%)

Training CatBoost...
✓ CatBoost - Accuracy: 1.0000 (100.00%)


============================================================
FINAL LEADERBOARD - RANKED BY ACCURACY (VERSION 1)
============================================================
1 Rank 1: CatBoost             - 1.0000 (100.00%)
2 Rank 2: XGBoost              - 0.9600 (96.00%)
3 Rank 3: Gradient Boosting    - 0.9000 (90.00%)
. Rank 4: Random Forest        - 0.8400 (84.00%)
============================================================


 Leaderboard as DataFrame:
    Rank              Model  Accuracy  Accuracy_Percentage
0     1           CatBoost      1.00                100.0
1     2            XGBoost      0.96                 96.0
2     3  Gradient Boosting      0.90                 90.0
3     4      Random Forest      0.84                 84.0
```

Milestone 6: Model Deployment

Activity 1: Save the best model

Saving the best model after comparing its performance using different evaluation metrics means selecting the model with the highest performance. This can be useful in avoiding the need to retrain the model every time it is needed and also to be able to use it in the future.

```
with open(filename, 'wb') as file:
    pickle.dump(model_to_save, file)

saved_models[model_name] = filename
```

Activity 2: Integrate with Web Framework

We will be building a web application that is integrated to the model we built. A UI is

provided for the uses where he has to enter the values for predictions. The enter values are given to the saved model and prediction is showcased on the UI. This section has the following tasks.

- Building HTML Pages
- Building server-side script
- Run the web application

Activity 2.1: Building Html Page:

For this project create HTML file namely

Index.html

Activity 2.2: Build Python code

Import the libraries. Load the saved model. Importing the flask module in the project is mandatory. An object of Flask class is our WSGI application. Flask constructor takes the name of the current module as argument.

```
with open('v1_1CatBoost.pkl', 'rb') as file:
    model = pickle.load(file)
```

```
with open('scaler.pkl', 'rb') as f:
    scaler = pickle.load(f)
```

```
app = Flask(__name__)
```

Render HTML page:

```
@app.route('/')
def index():
    return render_template('index.html')
```

Here we will be using a declared constructor to route to the HTML page which we have created earlier. Then the home page of the web server is opened in the browser, the html page will be rendered. Whenever you enter the values from the html page the values can be retrieved using POST Method. Retrieves the value from UI:

```python
# Column lists used during training
num_cols = ['age', 'bp', 'sg', 'al', 'su', 'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wc', 'rc']
cat_cols = ['rbc', 'pc', 'pcc', 'ba', 'htn', 'dm', 'cad', 'appet', 'pe', 'ane']

try:
    # Process numerical columns with validation
    for col in num_cols:
        val = form_data.get(col, 0)
        try:
            patient_data[col] = float(val)
        except (ValueError, TypeError):
            logger.warning(f"Invalid numerical value for {col}: {val}, using 0")
            patient_data[col] = 0.0

    # Process categorical columns with validation
    for col in cat_cols:
        val = form_data.get(col, 'unknown')
        patient_data[col] = str(val) if val is not None else 'unknown'

    # Create DataFrame from processed input data (original values)
    df = pd.DataFrame([patient_data])
    logger.info(f"Patient data: {patient_data}")
```

```python
# Extract only the specified features
feature_cols = ['anemia_severity','hemo','sg','comorb_score','eGFR','rbc','sc','dm','htn','bgr','symptom_severity']
```

```python
df_features[features_to_scale] = scaler.transform(df_features[features_to_scale])
```

```python
df_features = pd.get_dummies(df_features, columns=cat_cols_for_encoding, drop_first=True)
```

```python
# Select only the features the model expects
model_input_df = features_df[model.feature_names_]

prediction = model.predict(model_input_df)[0]
```

```python
response = {
    'success': True,
    'prediction': result,
    'confidence': float(confidence),
    'engineered_features': engineered_features,
    'timestamp': datetime.now().isoformat(),
    'patient_data': patient_data
}
```

Here we are routing our app to predict() function. This function retrieves all the values from the HTML page using Post request. That is stored in an array. This array is passed to the model.predict() function. This function returns the prediction. And this prediction value will be rendered to the text that we have mentioned in the submit.html page earlier.

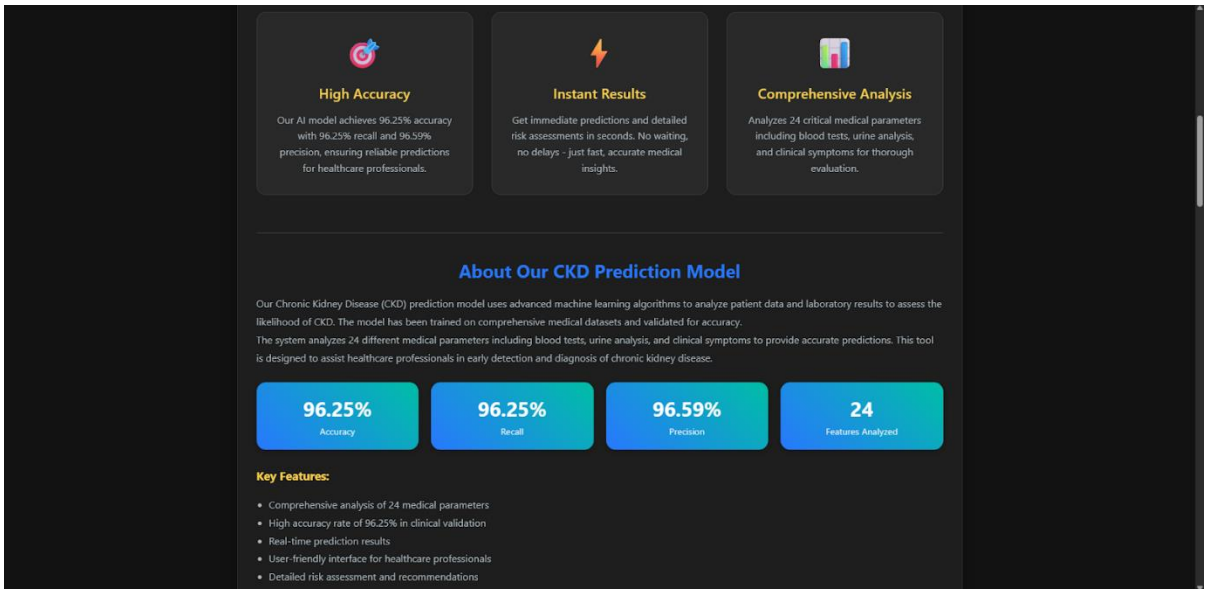
Main Function:

```
if __name__ == '__main__':
    if load_model():
        logger.info("Starting CKD Prediction Flask App")
        app.run(debug=True, host='0.0.0.0', port=5000)
```

Activity 2.3: Run the web application

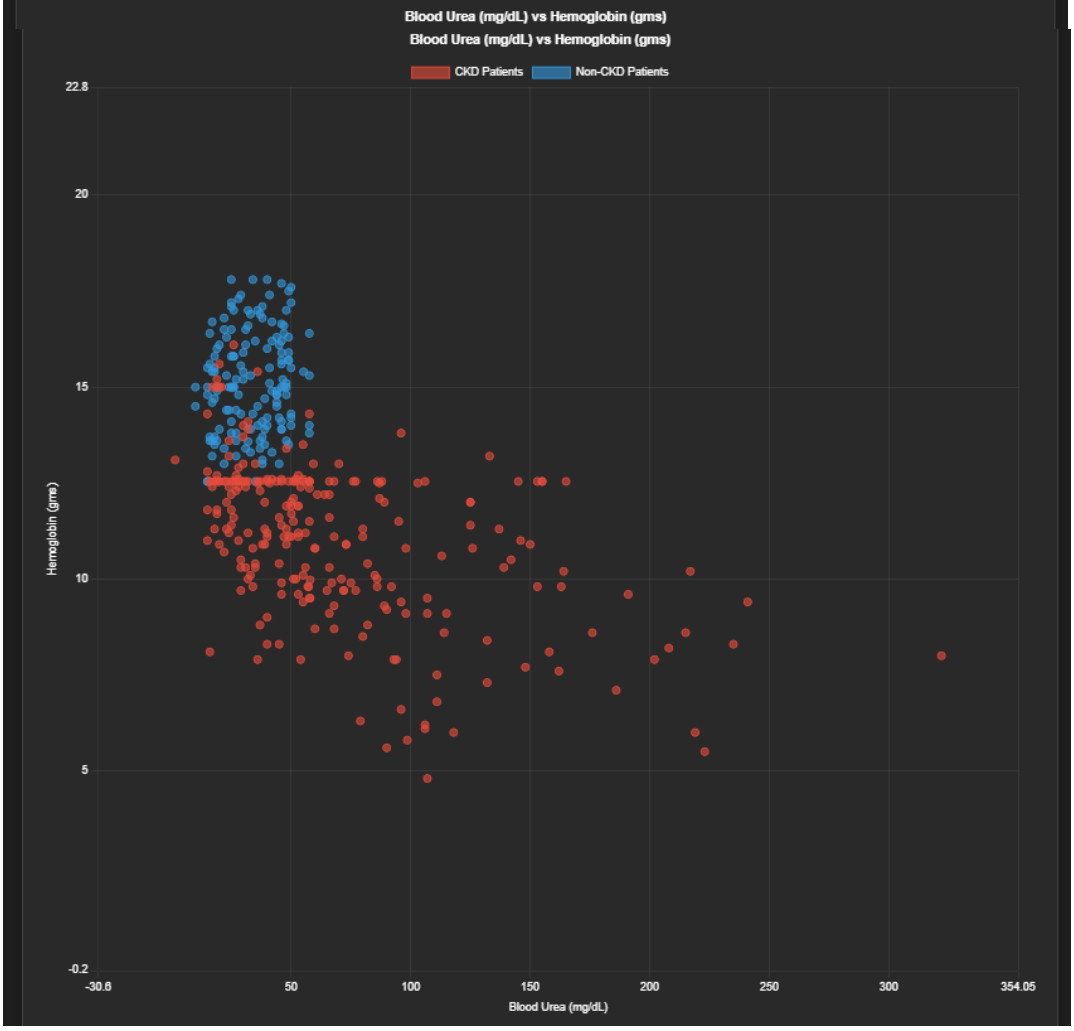
```
(env) PS C:\Users\cheri\projects\MyFlaskApp> python app.py
2025-06-20 15:50:30,539 INFO: Flask app started!
2025-06-20 15:50:31,035 INFO: Model loaded successfully
2025-06-20 15:50:31,035 INFO: Starting CKD Prediction Flask App
 * Serving Flask app 'app'
 * Debug mode: on
2025-06-20 15:50:31,054 INFO: WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.52:5000
2025-06-20 15:50:31,055 INFO: Press CTRL+C to quit
2025-06-20 15:50:31,056 INFO:  * Restarting with stat
2025-06-20 15:50:32,911 INFO: Flask app started!
2025-06-20 15:50:33,447 INFO: Model loaded successfully
2025-06-20 15:50:33,447 INFO: Starting CKD Prediction Flask App
2025-06-20 15:50:33,456 WARNING:  * Debugger is active!
2025-06-20 15:50:33,461 INFO:  * Debugger PIN: 186-335-509
```

Now, go to the web browser to get the below result

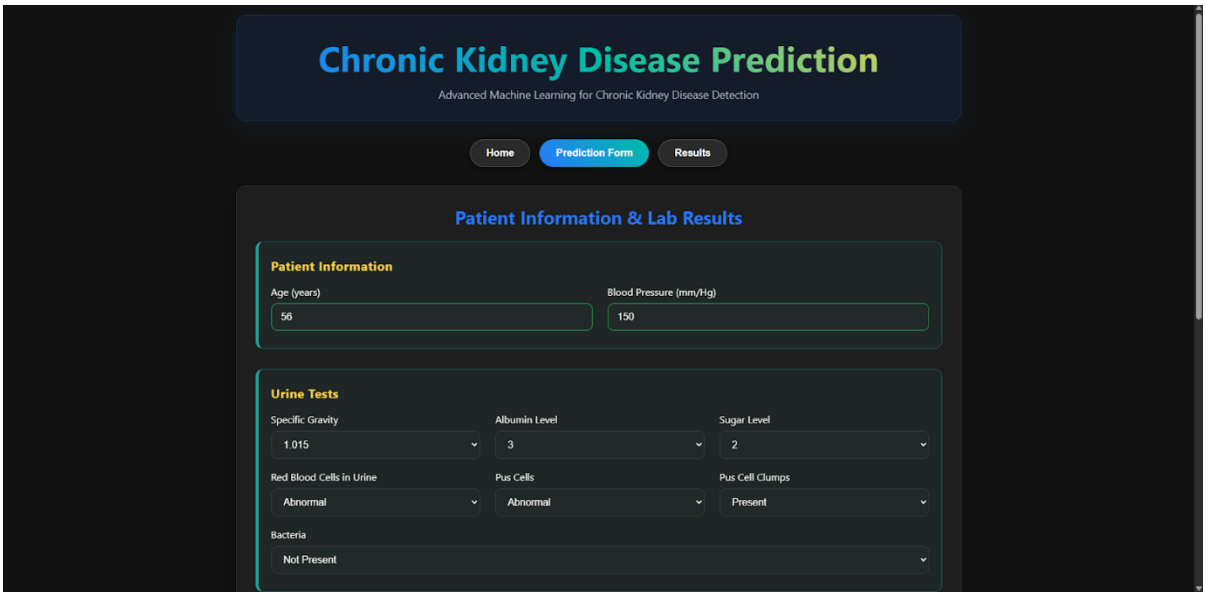


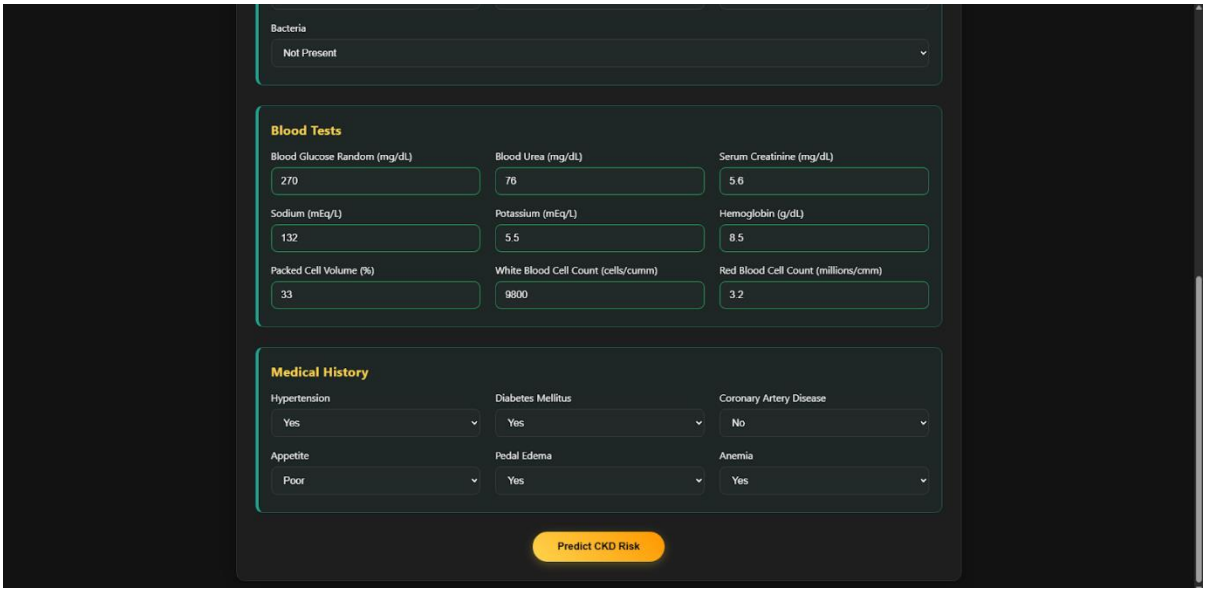
Graphs in the home page showing plots between the important features:


Serum Creatinine (mg/dL) vs Hemoglobin (gms)
CKD Patients
Non-CKD Patients
22.8
20
15
10
5
-0.2
Hemoglobin (gms)
-4.6
10
20
30
40
52.9
Serum Creatinine (mg/dL)

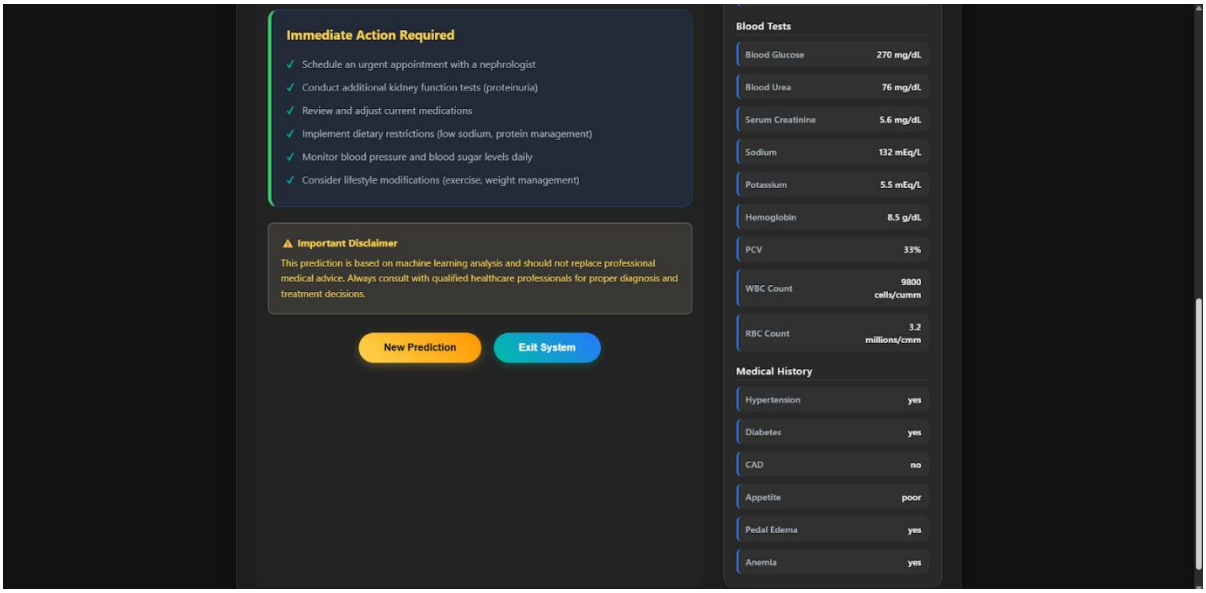

Blood Urea (mg/dL) vs Hemoglobin (gms)
Blood Urea (mg/dL) vs Hemoglobin (gms)
CKD Patients
Non-CKD Patients
22.8
20
15
10
5
-0.2
Hemoglobin (gms)
-30.8
50
100
150
200
250
300
354.05
Blood Urea (mg/dL)

Prediction Form:

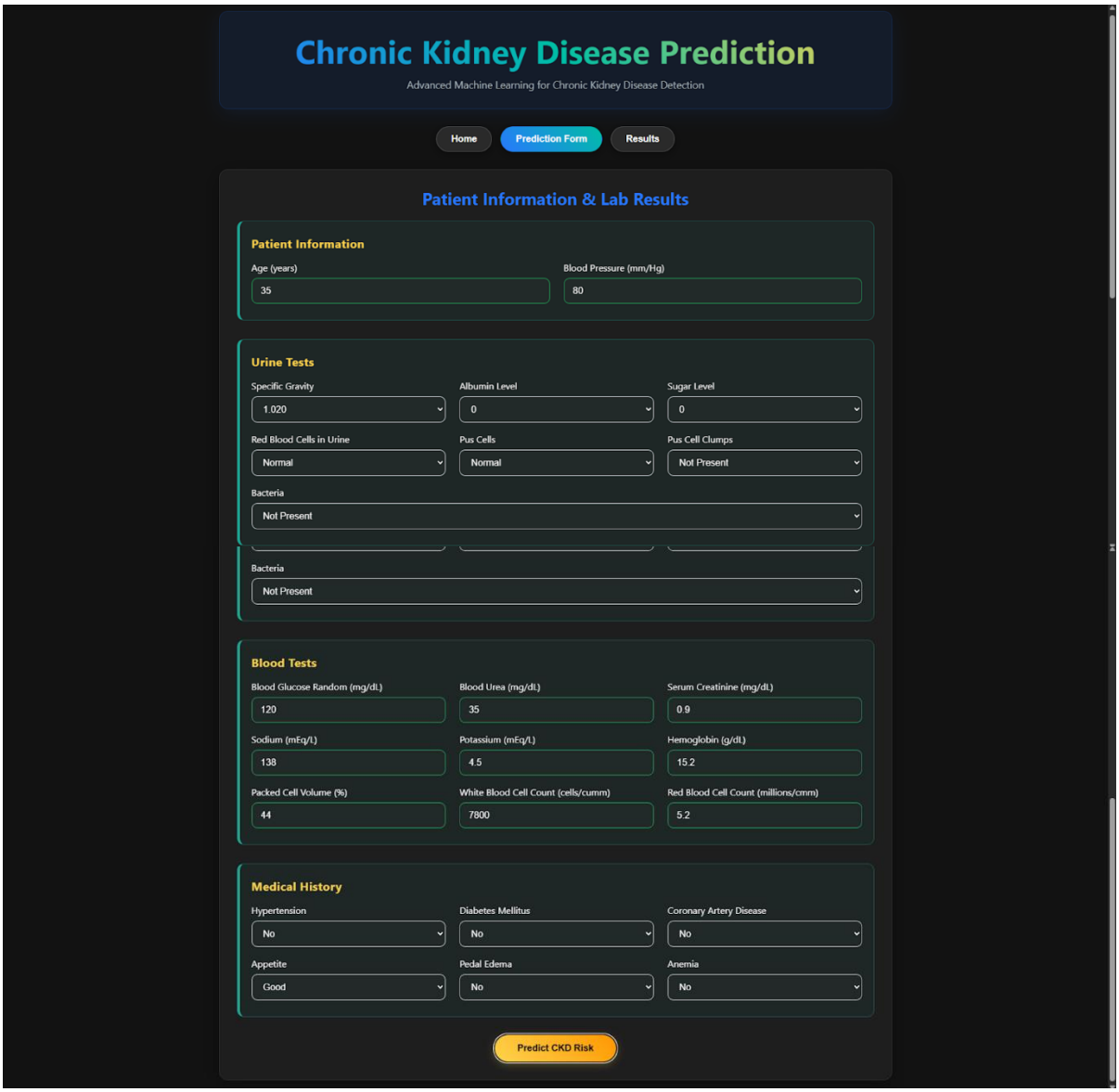


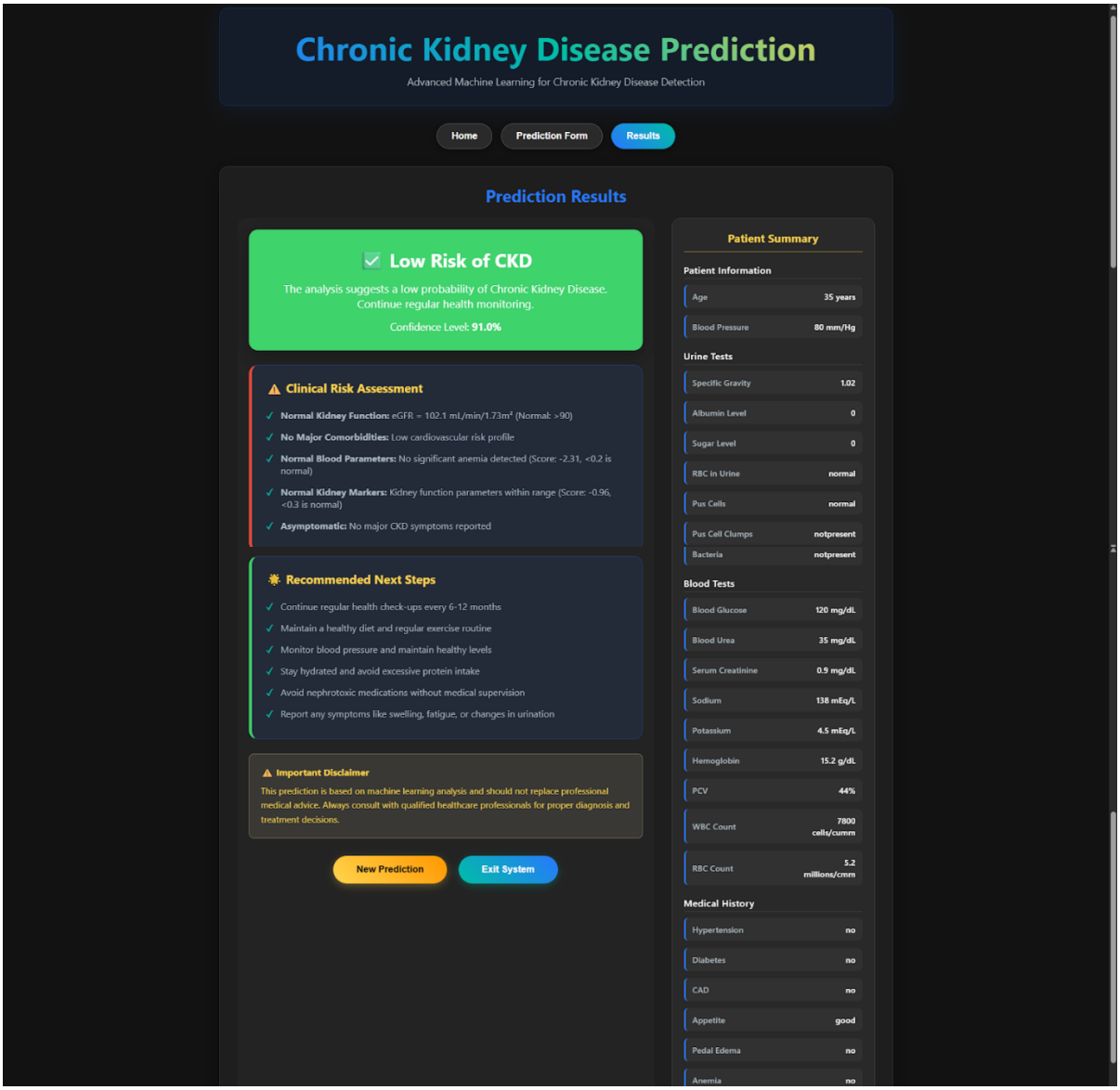


Page predicting high CKD risk:



Click on the “New Prediction” button. Predicting low CKD risk:





Click on the “Exit System” button which will take you to the Thank you page: