

Milestone 1: Define Problem / Problem Understanding

Activity 1: Specify the business problem

Chronic Kidney Disease (CKD) is a serious medical condition that progresses slowly and can lead to end-stage renal failure if not detected early. It often develops silently, with no noticeable symptoms in the early stages, making timely diagnosis a significant challenge.

Healthcare systems worldwide struggle with the early identification and management of CKD due to rising patient volumes and limited access to specialists. Early detection is critical to prevent disease progression and manage complications through timely intervention.

This project aims to predict whether a patient has CKD or not using a machine learning model trained on clinical and laboratory data. The dataset includes features such as age, blood pressure, albumin levels, specific gravity, red blood cell count, blood urea, serum creatinine, haemoglobin, and more.

By analysing these indicators, the model can assist in early diagnosis—particularly in settings where specialists or advanced diagnostic tools are not easily accessible.

Activity 2: Business requirements

Some key business requirements for this CKD prediction project are:

- **Accurate & Reliable Data:** The system must use recent, verified clinical data (such as that provided in open datasets like Kaggle) to ensure prediction accuracy.
- **Scalability:** The model should be capable of handling increasing volumes of patient records without loss of performance.
- **Interpretability:** The prediction results should be explainable and interpretable to aid medical professionals in understanding risk factors.
- **Compliance:** The solution should comply with health data handling and privacy standards (like HIPAA).
- **Integration-Friendly:** The final application (via Flask) should be user-friendly and able to integrate with existing healthcare systems or dashboards.
- **Continuous Improvement:** The system should allow periodic retraining/updating to adapt to new patient data and improve performance over time.

Activity 3: Literature Survey

The goal of this survey is to gather insights into:

- The clinical features most relevant for CKD detection.
- The algorithms and models used previously.
- The effectiveness of those models.
- The gaps or limitations in earlier approaches that this project can aim to address.

Multiple studies have shown that early detection of CKD using machine learning can significantly improve patient outcomes. Algorithms like Support Vector Machines (SVM), Decision Trees, Random Forests, and Logistic Regression have been applied to predict CKD based on patient data such as blood pressure, specific gravity, albumin, blood urea, and serum creatinine levels.

Datasets such as the CKD dataset available on Kaggle provide well-structured clinical records, enabling researchers to train and evaluate their models effectively.

For example, a study published in IJERT (2020) achieved 97% accuracy in CKD detection using ensemble techniques, highlighting the importance of data pre-processing and proper feature selection.

This project builds upon the findings of such research by implementing a clean and interpretable ML model for CKD prediction. It also extends previous work by integrating the model with a Flask-based web application, making it more accessible and usable in real-time scenarios. The literature survey confirms the practical relevance of this approach and guides the design and development of the current solution.

Activity 4: Social or Business Impact

Social Impact:

- **Improved Patient Outcomes:** Early detection leads to timely treatment, preventing serious complications and improving patient quality of life.
- **Accessibility:** A lightweight, app-based prediction tool can benefit rural/remote regions lacking specialist doctors.
- **Awareness & Education:** Such systems can raise awareness among patients regarding kidney health.

Business Impact:

- **Cost Reduction:** Early diagnosis reduces the need for expensive treatments like dialysis and transplant surgeries.
- **Healthcare Efficiency:** Helps hospitals prioritize high-risk patients and manage resources better.
- **Preventive Healthcare Business Models:** Opens doors for health startups to build smart diagnostic tools integrated with ML.

Technology Stack:

- Python
- Pandas, NumPy
- Scikit-learn, XGBoost
- Matplotlib, Seaborn
- LabelEncoder / OneHotEncoder

- Flask
- Jupyter Notebook
- CSV Dataset (Kaggle)

Milestone 2: Data Collection & Preparation

Activity 1: Collect the dataset

For this project, we use a dataset focused on Chronic Kidney Disease (CKD), which contains clinical and laboratory data of patients. The dataset was sourced from Kaggle and is in .csv format.

Dataset Source:

<https://www.kaggle.com/datasets/mansoordaku/ckdisease>

This dataset includes features such as:

- **Demographics:** age, gender
- **Clinical data:** blood pressure, blood urea, serum creatinine, red blood cell count, haemoglobin, etc.
- **Categorical indicators:** hypertension, diabetes, appetite, anemia, etc.
- **Target:** Whether the patient has CKD (ckd) or not (not ckd)

We'll use this dataset to build and train a machine learning model that can predict whether a person has CKD based on their input parameters.

Activity 1.1: Importing the libraries

The following essential Python libraries were used:

```
✓ 0s IMPORTING LIBRARIES  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
from sklearn.model_selection import train_test_split  
from sklearn.preprocessing import LabelEncoder, StandardScaler  
from sklearn.linear_model import LogisticRegression  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.svm import SVC  
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score  
from sklearn.model_selection import GridSearchCV  
from xgboost import XGBClassifier  
import pickle
```

These libraries are important for data handling, manipulation, visualization, and building machine learning models.

Activity 1.2: Read the Dataset

We read the .csv file using pandas:

```
df = pd.read_csv()
```

✓ IMPORTING DATASET

```
✓ [2] # Import the kagglehub library to download datasets from Kaggle
3s import kagglehub
path = kagglehub.dataset_download("mansoordaku/ckdisease")
print(f"Dataset path is {path}")
```

↗ Dataset path is /kaggle/input/ckdisease

```
✓ [3] # Traverse and display all file paths in the downloaded dataset directory to identify available data files
3s import os
for root,dirs,files in os.walk(path):
    for file in files:
        print(os.path.join(root,file))
print(os.listdir(path))
```

↗ /kaggle/input/ckdisease/kidney_disease.csv
['kidney_disease.csv']

```
✓ [4] # Construct the full path to the CSV file and load the dataset into a pandas DataFrame
3s data = os.path.join(path, "kidney_disease.csv")
df = pd.read_csv(data)
```

Activity 2: Data Preparation

Raw medical data often contains missing values, inconsistent types, or outliers. So, we cleaned and prepared the dataset before using it to train our model.

Activity 2.1: Handling Missing Values

We checked for null values using:

`df.isnull().any()`

and to check how many null values are present in a particular feature, we use:

`df.isnull().sum()`

```
df.isnull().sum() # Gives how many missing values are present in a particular feature
```



	0
age	9
blood_pressure	12
specific_gravity	47
albumin	46
sugar	49
red_blood_cells	152
pus_cell_count	65
pus_cell_clumps	4
bacteria	4
blood_glucose_random	44
blood_urea	19
serum_creatinine	17
sodium	87
potassium	88
haemoglobin	52
packed_cell_volume	71
white_blood_cell_count	106
red_blood_cell_count	131
hypertension	2
diabetes_mellitus	2
coronary_artery_disease	2
appetite	1
pedal_edema	1
anemia	1
classification	0

dtype: int64

We found that several columns like red_blood_cells, white_blood_cell_count, packed_cell_volume, and potassium contained missing values.

Techniques used:

- **Mode Imputation** for categorical features
- **Mean or Median Imputation** for numerical features

Replacing Missing Values using Statistical imputations

```
pd.set_option('future.no_silent_downcasting', True)
# This line ensures that pandas throws a warning or error if it has to silently change data types during imputation.
# It helps avoid unintended "downcasting" of data types (e.g., from float64 to float32) when filling missing values.

df['blood_glucose_random'] = df['blood_glucose_random'].fillna(df['blood_glucose_random'].mean())
df['blood_pressure'] = df['blood_pressure'].fillna(df['blood_pressure'].mean())
df['blood_urea'] = df['blood_urea'].fillna(df['blood_urea'].mean())
df['haemoglobin'] = df['haemoglobin'].fillna(df['haemoglobin'].mean())
df['packed_cell_volume'] = df['packed_cell_volume'].fillna(df['packed_cell_volume'].mean())
df['potassium'] = df['potassium'].fillna(df['potassium'].mean())
df['red_blood_cell_count'] = df['red_blood_cell_count'].fillna(df['red_blood_cell_count'].mean())
df['serum_creatinine'] = df['serum_creatinine'].fillna(df['serum_creatinine'].mean())
df['sodium'] = df['sodium'].fillna(df['sodium'].mean())
df['white_blood_cell_count'] = df['white_blood_cell_count'].fillna(df['white_blood_cell_count'].mean())
```

```
# Filling missing values using the most frequent value (mode)

df['age'] = df['age'].fillna(df['age'].mode()[0])
df['hypertension'] = df['hypertension'].fillna(df['hypertension'].mode()[0])
df['pus_cell_clumps'] = df['pus_cell_clumps'].fillna(df['pus_cell_clumps'].mode()[0])
df['appetite'] = df['appetite'].fillna(df['appetite'].mode()[0])
df['albumin'] = df['albumin'].fillna(df['albumin'].mode()[0])
df['pus_cell_count'] = df['pus_cell_count'].fillna(df['pus_cell_count'].mode()[0])
df['red_blood_cells'] = df['red_blood_cells'].fillna(df['red_blood_cells'].mode()[0])
df['coronary_artery_disease'] = df['coronary_artery_disease'].fillna(df['coronary_artery_disease'].mode()[0])
df['bacteria'] = df['bacteria'].fillna(df['bacteria'].mode()[0])
df['anemia'] = df['anemia'].fillna(df['anemia'].mode()[0])
df['sugar'] = df['sugar'].fillna(df['sugar'].mode()[0])
df['diabetes_mellitus'] = df['diabetes_mellitus'].fillna(df['diabetes_mellitus'].mode()[0])
df['pedal_edema'] = df['pedal_edema'].fillna(df['pedal_edema'].mode()[0])
df['specific_gravity'] = df['specific_gravity'].fillna(df['specific_gravity'].mode()[0])
```

To avoid data type conversion issues, we used:

```
pd.set_option('future.no_silent_downcasting', True)
```

This ensures safe type casting while filling missing values, especially when using statistical imputations.

Activity 2.2: Handling Incorrect Data Types

Some numeric columns were mistakenly classified as object types due to mixed formatting.

We corrected this with:

```
[ ] df['packed_cell_volume'] = pd.to_numeric(df['packed_cell_volume'], errors = 'coerce')
    df['white_blood_cell_count'] = pd.to_numeric(df['white_blood_cell_count'], errors = 'coerce')
    df['red_blood_cell_count'] = pd.to_numeric(df['red_blood_cell_count'], errors = 'coerce')
```

Activity 2.3: Handling Outliers

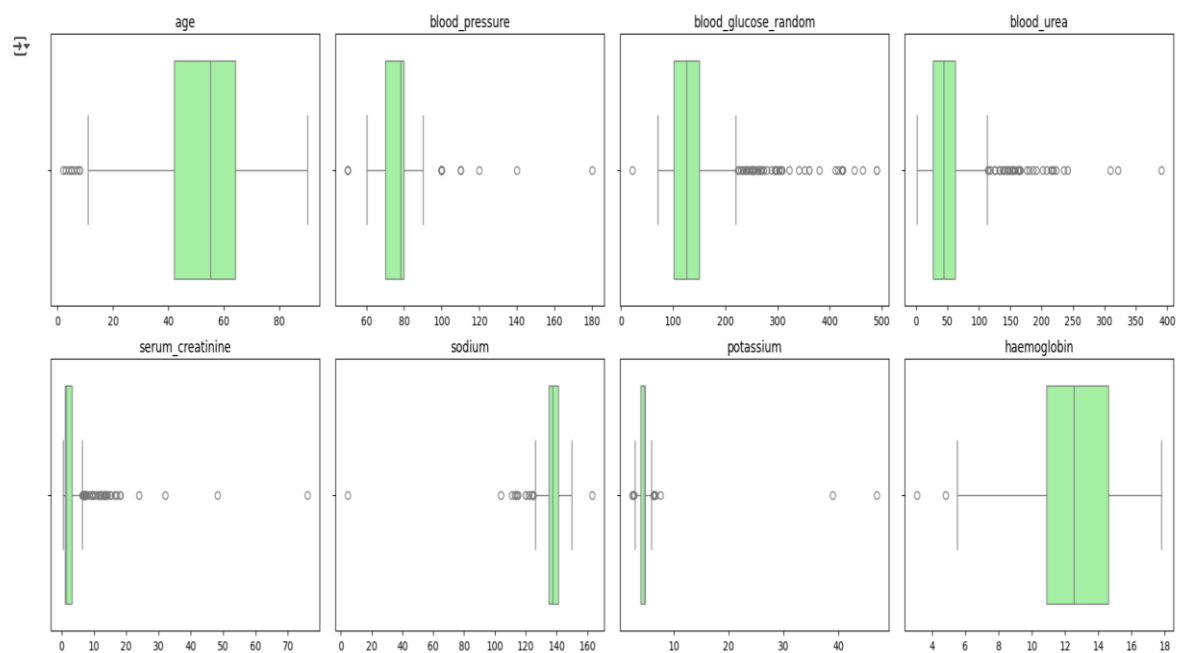
Outliers can negatively affect model performance. We used boxplots to identify outliers in features like:

✓
1s

```
# Boxplots to visualize outliers
fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(18, 8))
fig.suptitle("Boxplots of Numerical Features", fontsize=16)
axes = axes.flatten()

for i, col in enumerate(numerical_cols):
    sns.boxplot(x=df[col], ax=axes[i], color='palegreen')
    axes[i].set_title(f'{col}')
    axes[i].set_xlabel('')

plt.tight_layout()
plt.subplots_adjust(top=0.88)
plt.show()
```



To detect the presence of outliers, we used boxplots for each numerical feature. These boxplots were plotted in a grid layout using subplots to give a consolidated view of data spread and extreme values. From these visualizations, we identified the features where significant outliers exist and further considered techniques like IQR method or model-based handling if necessary during model training phase.

Milestone 3: Exploratory Data Analysis

Activity 1: Descriptive statistical

Descriptive analysis helps us understand the distribution, central tendency, and spread of the dataset. Using the describe() function from the pandas library, we gained statistical insights such as mean, standard deviation, minimum and maximum values, and percentiles of the numerical features. This helped us detect outliers, skewness, and range of values.

```
# Generate descriptive statistics for all numerical columns in the DataFrame
df.describe()
```

	age	blood_pressure	specific_gravity	albumin	sugar	blood_glucose_random	blood_urea	serum_creatinine	sodium	potassium	haemoglobin
count	391.000000	388.000000	353.000000	354.000000	351.000000	356.000000	381.000000	383.000000	313.000000	312.000000	348.000000
mean	51.483376	76.469072	1.017408	1.016949	0.450142	148.036517	57.425722	3.072454	137.528754	4.627244	12.526437
std	17.169714	13.683637	0.005717	1.352679	1.099191	79.281714	50.503006	5.741126	10.408752	3.193904	2.912587
min	2.000000	50.000000	1.005000	0.000000	0.000000	22.000000	1.500000	0.400000	4.500000	2.500000	3.100000
25%	42.000000	70.000000	1.010000	0.000000	0.000000	99.000000	27.000000	0.900000	135.000000	3.800000	10.300000
50%	55.000000	80.000000	1.020000	0.000000	0.000000	121.000000	42.000000	1.300000	138.000000	4.400000	12.650000
75%	64.500000	80.000000	1.020000	2.000000	0.000000	163.000000	66.000000	2.800000	142.000000	4.900000	15.000000
max	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

Visual exploration gives us better intuition of how the data behaves. It also helps to quickly identify unusual patterns, imbalance, or significant differences.

Activity 2.1: Univariate Analysis

Univariate analysis focuses on single features at a time.

- **Histograms and KDE plots** were used for numerical features like blood_urea, age, and haemoglobin to understand their distribution.
 - Most features showed right-skewed distributions, especially serum creatinine and blood urea, which are critical markers in CKD.

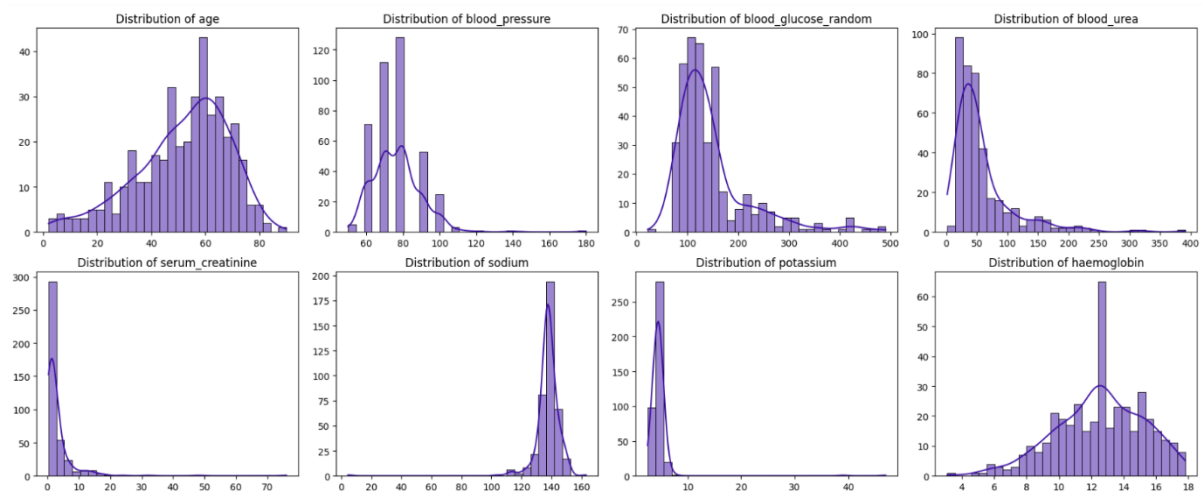
```
# Distribution of numerical features
numerical_cols = ['age', 'blood_pressure', 'blood_glucose_random', 'blood_urea', 'serum_creatinine', 'sodium', 'potassium', 'haemoglobin']

# Add this line to create subplot grid
fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(18, 8))
fig.suptitle("Distribution of Numerical Features", fontsize=16)

# Flatten axes array for easy access
axes = axes.flatten()

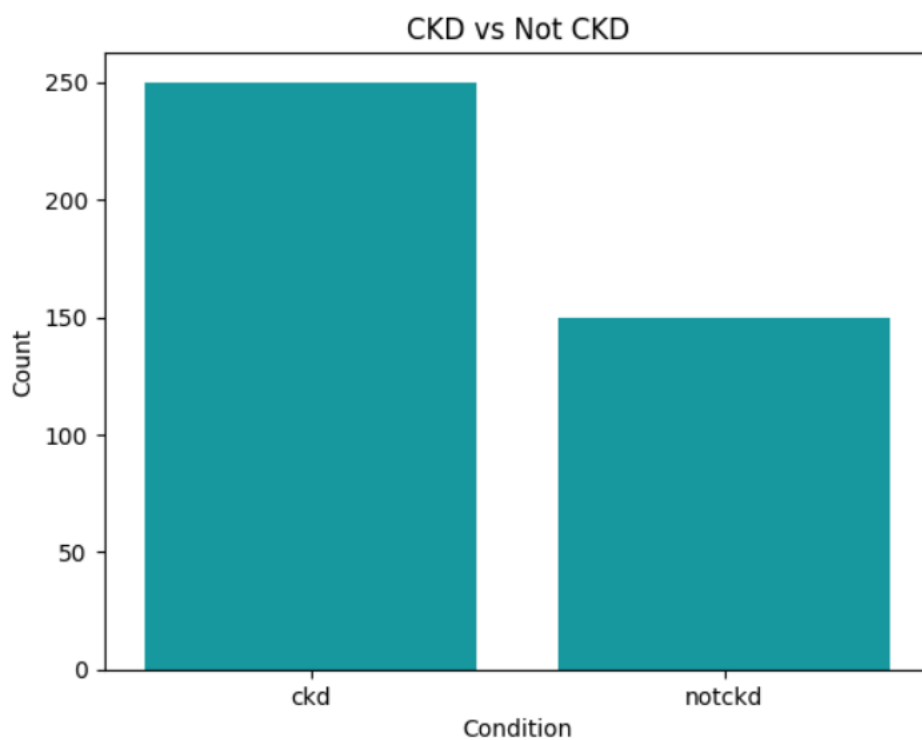
for i, col in enumerate(numerical_cols):
    sns.histplot(data=df, x=col, kde=True, bins=30, ax=axes[i], color = '#3A0CA3')
    axes[i].set_title(f'Distribution of {col}')
    axes[i].set_xlabel('')
    axes[i].set_ylabel('')

plt.tight_layout()
plt.subplots_adjust(top=0.88)
plt.show()
```



- **Countplots** were used for categorical features like hypertension, diabetes_mellitus, anemia, and the target variable classification (CKD or not).
 - There was a class imbalance: More patients had CKD compared to non-CKD.

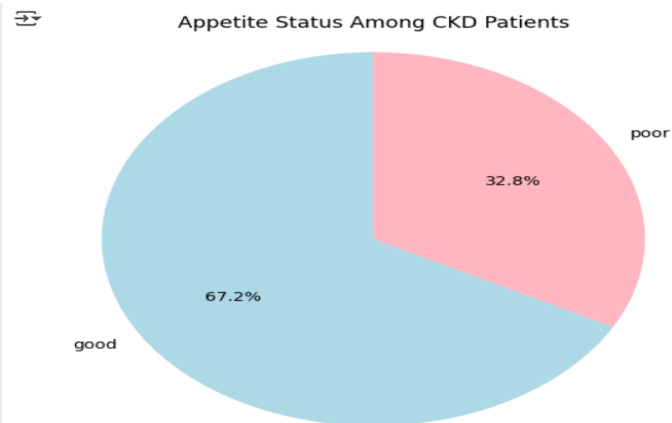
```
# Countplot for target variable
sns.countplot(x='classification', data=df, color = '#00ADB5')
plt.title("CKD vs Not CKD")
plt.xlabel("Condition")
plt.ylabel("Count")
plt.show()
```



- **Pie Charts** were used to visualize the distribution of binary categorical features like appetite and pedal_edema.

```
# Pie chart showing appetite levels among CKD patients
ckd_appetite = df[df['classification'] == 'ckd']['appetite'].value_counts()

plt.figure(figsize=(6,6))
plt.pie(ckd_appetite, labels=ckd_appetite.index, autopct='%1.1f%%', startangle=90, colors=['lightblue', 'lightpink'])
plt.title('Appetite Status Among CKD Patients')
plt.axis('equal')
plt.show()
```



Activity 2.2: Bivariate Analysis

Here we explored the **relationship between two features**, especially in relation to the target variable.

- **Barplots & Countplots (with hue='classification')** showed how categories like anemia, diabetes_mellitus, and hypertension differed across CKD and non-CKD groups.
 - For instance, patients with diabetes or hypertension had a significantly higher occurrence of CKD.

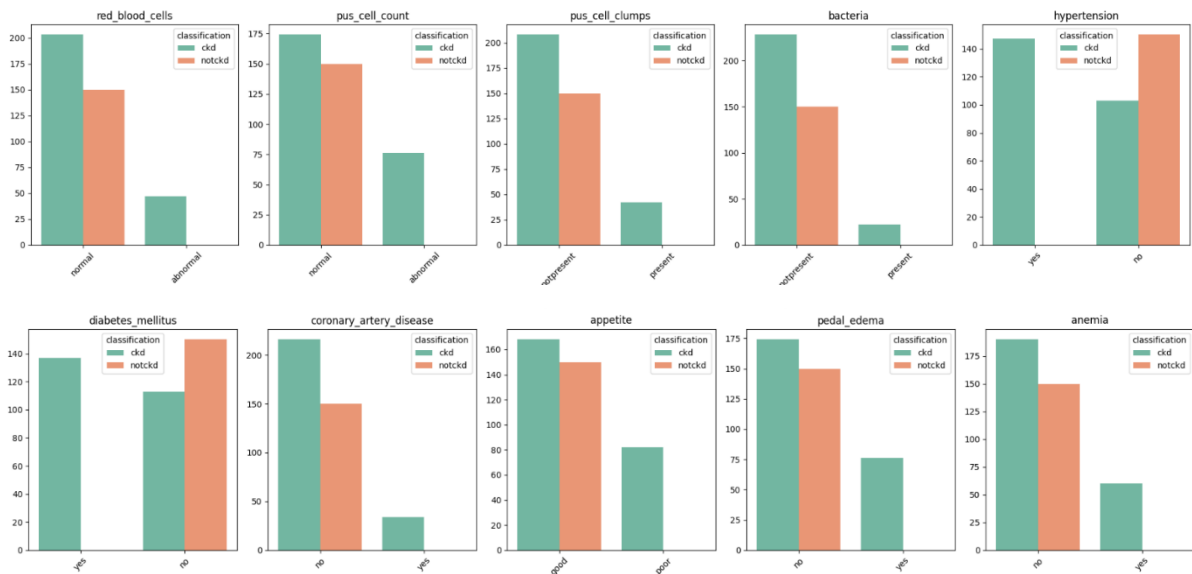
```
# Barplots of categorical features vs CKD classification
categorical_cols = ['red_blood_cells', 'pus_cell_count', 'pus_cell_clumps', 'bacteria', 'hypertension', 'diabetes_mellitus', 'coronary_artery_disease',
                   'appetite', 'pedal_edema', 'anemia']

# Create subplot grid
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(20, 10))
fig.suptitle("Categorical Features vs CKD Classification", fontsize=16)
axes = axes.flatten()

# Plot each feature
for i, col in enumerate(categorical_cols):
    sns.countplot(x=col, hue='classification', data=df, ax=axes[i], palette='Set2')
    axes[i].set_title(f'{col}')
    axes[i].tick_params(axis='x', rotation=45)
    axes[i].set_xlabel('')
    axes[i].set_ylabel('')

# Adjust layout
plt.tight_layout()
plt.subplots_adjust(top=0.90)
plt.show()
```

Categorical Features vs CKD Classification



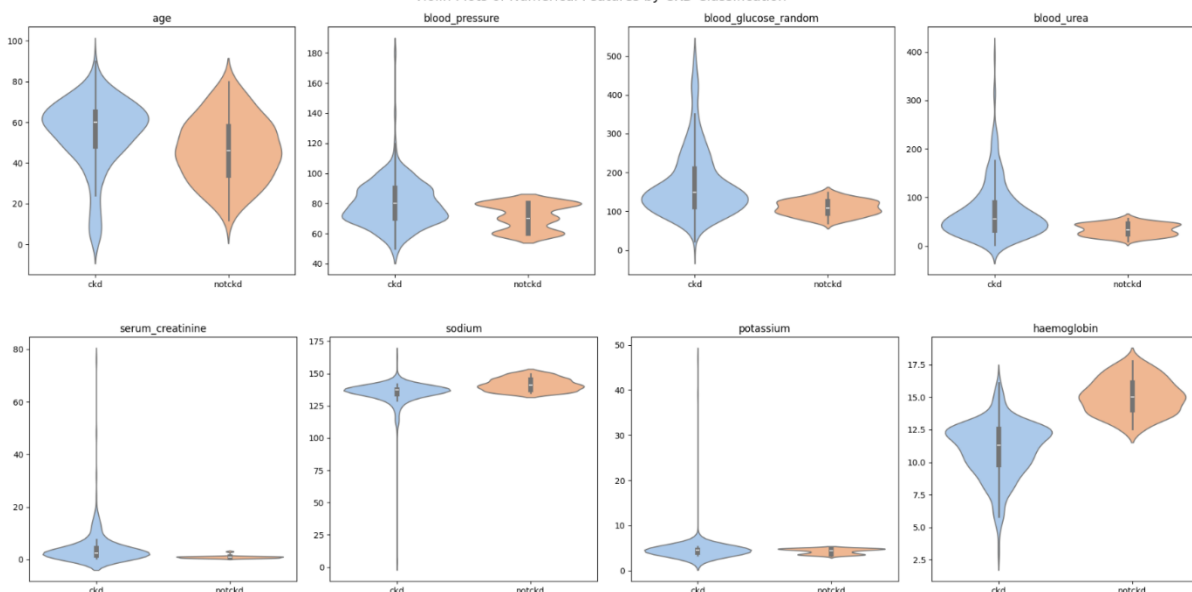
- **Violin plots** were used to compare numerical features like haemoglobin across CKD classes, revealing how distributions shifted for patients with the disease.

```
# Violin plots for numerical features with subplots
fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(20, 10))
fig.suptitle("Violin Plots of Numerical Features by CKD Classification", fontsize=16)
axes = axes.flatten()

for i, col in enumerate(numerical_cols):
    sns.violinplot(x='classification', y=col, hue='classification', data=df, ax=axes[i],
                  palette='pastel', legend=False)
    axes[i].set_title(f'{col}')
    axes[i].set_xlabel('')
    axes[i].set_ylabel('')

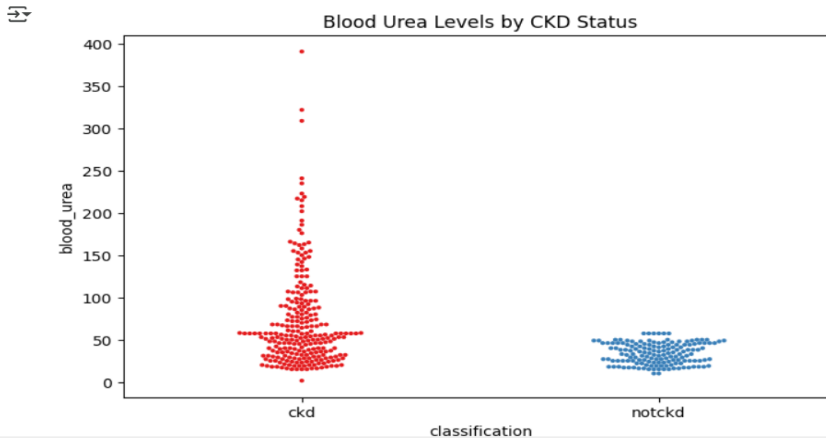
plt.tight_layout()
plt.subplots_adjust(top=0.92)
plt.show()
```

Violin Plots of Numerical Features by CKD Classification



- **Swarmplots** helped visualize point-wise variations in blood urea and serum creatinine by CKD classification.

```
# Swarmplot showing the distribution and individual variation of blood urea levels across CKD and non-CKD patients
plt.figure(figsize=(8, 5))
sns.swarmplot(x='classification', y='blood_urea', hue='classification', data=df, palette='Set1', size=3, legend=False)
plt.title("Blood Urea Levels by CKD Status")
plt.show()
```

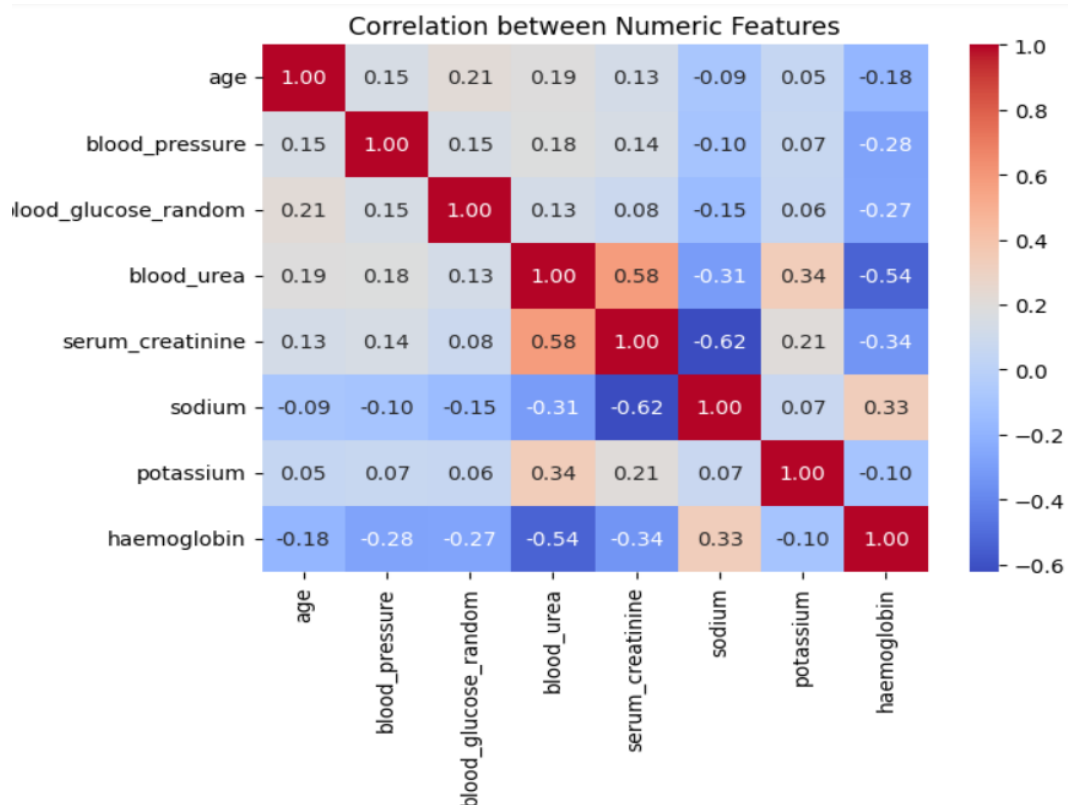


Activity 2.3: Multivariate Analysis

This activity involved studying relationships between **multiple variables**.

- A **correlation heatmap** was generated using `seaborn.heatmap()`.
 - *Features like serum_creatinine, blood_urea, and haemoglobin showed stronger correlations with the CKD outcome.*
 - *There were weak correlations among many features, suggesting most provide unique information.*
- This helped in **feature selection** and identifying redundant or highly correlated features, guiding better model performance.

```
# Correlation heatmap
plt.figure(figsize=(10,8))
sns.heatmap(df[numerical_cols].corr(), annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation between Numeric Features")
plt.show()
```



Activity: Encoding the Categorical Features

In this step, we convert the categorical variables into a numerical format so they can be understood by the machine learning models.

- Categorical features such as red_blood_cells, pus_cell_count, bacteria, hypertension, etc., cannot be directly fed into most ML algorithms.
- Hence, we apply **Label Encoding**, which assigns a unique numerical value to each category in the feature column.
- We imported LabelEncoder from the **scikit-learn** library and applied fit_transform() method to each categorical column.
- This transformation ensures that all features in the dataset are numeric and suitable for model training.

```

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier

df_encoded = df.copy()
le_dict = {}

# Label Encode All Object Columns
for col in df_encoded.columns:
    if df_encoded[col].dtype == 'object' and col != 'classification':
        le = LabelEncoder()
        df_encoded[col] = le.fit_transform(df_encoded[col])
        le_dict[col] = le # Store encoder for later use (prediction)

# Encode the target variable
le_target = LabelEncoder()
df_encoded['classification'] = le_target.fit_transform(df_encoded['classification'])

```

Activity: Splitting the Data into Train and Test Sets

In order to train and evaluate the machine learning model effectively, we first need to split the dataset into independent features (X) and the target variable (y).

In our project, the target variable is 'classification', which indicates whether a patient is diagnosed with Chronic Kidney Disease (CKD) or not.

We perform the following steps:

1. Create X and y Variables:
 - X contains all the features except the target (classification).
 - y contains only the target feature (classification).
2. Use train_test_split() from sklearn:
 - This function splits the data into training and testing subsets.
 - We use a test size of 20%, meaning 80% of the data is used for training and 20% for testing.
 - The random_state parameter ensures the split is reproducible every time the code is run.

This process helps in assessing the model's performance on unseen data, ensuring that it generalizes well beyond the training data.

```

# Define features and target
X_full = df_encoded.drop('classification', axis=1)
y = df_encoded['classification']

# Split data
X_train, X_test, y_train, y_test = train_test_split(X_full, y, test_size=0.2, random_state=42)

# Apply RFE
X_train_cols = X_train.columns
rfe = RFE(estimator=RandomForestClassifier(n_estimators=100, random_state=42), n_features_to_select=8)
rfe.fit(X_train, y_train)

# Select final features
selected_features = X_train_cols[rfe.get_support()]
print("Selected Features:", selected_features.tolist())

# Overwrite X_train/X_test
X_train = X_train[selected_features]
X_test = X_test[selected_features]

```

Selected Features: ['specific_gravity', 'albumin', 'blood_glucose_random', 'serum_creatinine', 'haemoglobin', 'packed_cell_volume', 'red_blood_cell_count', 'hypertension']

Milestone 4: Model Building

Activity 1: Training the Model with Multiple Algorithms

Now that the dataset is pre-processed and ready, it is time to train Machine Learning models. In this project, we are using multiple classification algorithms to compare their performance and select the best-performing one for predicting Chronic Kidney Disease (CKD). Each algorithm brings a unique approach to learning from the data.

We are using the following classification models:

Activity 1.1: K-Nearest Neighbors (KNN)

KNN is a lazy learning algorithm that makes predictions by identifying the 'k' nearest data points in the training set. It assigns the most frequent class among those neighbors to the input instance. It works well for small datasets and when decision boundaries are not complex.

▼ KNN

✓
3s

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Define the hyperparameter grid
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

# Grid search with 5-fold cross-validation
grid_knn = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5, n_jobs=-1, verbose=1)
grid_knn.fit(X_train, y_train)

# Best model from grid search
best_knn = grid_knn.best_estimator_

# Predictions and evaluation
y_pred_knn = best_knn.predict(X_test)

print(" Best Hyperparameters:", grid_knn.best_params_)
print(f" Training Accuracy: {accuracy_score(y_train, best_knn.predict(X_train))}")
print(f" Test Accuracy: {accuracy_score(y_test, y_pred_knn)}\n")

print(" Confusion Matrix:\n", confusion_matrix(y_test, y_pred_knn))
print("\n Classification Report:\n", classification_report(y_test, y_pred_knn))
```

Activity 1.2: Decision Tree Classifier

A Decision Tree uses a tree-like structure to split the dataset based on conditions that improve information gain. It's easy to understand and interpret. However, a single tree can easily overfit, so ensemble methods are often preferred.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.model_selection import cross_val_score

# Define a Decision Tree with limited depth
dtc = DecisionTreeClassifier(max_depth=5, random_state=42)
dtc.fit(X_train, y_train)

# Evaluate on train and test sets
dtc_acc = accuracy_score(y_test, dtc.predict(X_test))

print(f"Training Accuracy of Decision Tree Classifier is {accuracy_score(y_train, dtc.predict(X_train)):.4f}")
print(f"Test Accuracy of Decision Tree Classifier is {dtc_acc:.4f} \n")

print(f"Confusion Matrix :- \n{confusion_matrix(y_test, dtc.predict(X_test))}\n")
print(f"Classification Report :- \n{classification_report(y_test, dtc.predict(X_test))}")

# Cross-Validation Accuracy
cv_scores_dtc = cross_val_score(dtc, X_full[selected_features], y, cv=5)
print(f"Cross-Validation Accuracy of Decision Tree (5-Fold): {cv_scores_dtc.mean():.4f}")
```

Activity 1.3: Random Forest Classifier

Random Forest is an ensemble technique that builds multiple decision trees and averages their predictions for better performance. It reduces variance and handles non-linear relationships effectively. It's robust and widely used in structured data problems.

Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.model_selection import cross_val_score

# Define the model
rf = RandomForestClassifier(random_state=0)

# Hyperparameter grid
param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [3, 5, 7],
    'max_features': ['sqrt'],
    'min_samples_split': [4, 6],
    'min_samples_leaf': [2, 4],
    'criterion': ['gini']
}

# Grid search
grid_rf = GridSearchCV(rf, param_grid, cv=5, n_jobs=-1, verbose=1)
grid_rf.fit(X_train, y_train)

# Best model
best_rf = grid_rf.best_estimator_
y_pred_rf = best_rf.predict(X_test)

# Evaluation
print("Best Hyperparameters:", grid_rf.best_params_)
print(f"Training Accuracy: {accuracy_score(y_train, best_rf.predict(X_train)):.4f}")
print(f"Test Accuracy: {accuracy_score(y_test, y_pred_rf):.4f}\n")

print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
print("\nClassification Report:\n", classification_report(y_test, y_pred_rf))

# Cross-validation accuracy
cv_rf = cross_val_score(best_rf, X_full[selected_features], y, cv=5)
print(f"Cross-Validation Accuracy (5-Fold): {cv_rf.mean():.4f}")
```

Activity 1.4: AdaBoost Classifier (Adaptive Boosting)

AdaBoost combines several weak learners (usually shallow decision trees) and improves them iteratively by focusing on the samples that were previously misclassified. It performs well on imbalanced data and is resistant to overfitting if tuned properly.

ADABOOST Classifier

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Base estimator (a shallow tree)
base_dtc = DecisionTreeClassifier(random_state=0)

# AdaBoost classifier with base decision tree
ada = AdaBoostClassifier(estimator=base_dtc, random_state=0)

# Parameter grid including base estimator hyperparameters
param_grid = {
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.01, 0.1, 1],
    'estimator__max_depth': [1, 2, 3, 4],
    'estimator__min_samples_split': [2, 5],
    'estimator__criterion': ['gini', 'entropy']
}

# Grid search
grid_ada = GridSearchCV(ada, param_grid, cv=5, n_jobs=-1, verbose=1)
grid_ada.fit(X_train, y_train)

# Best model
best_ada = grid_ada.best_estimator_
y_pred_ada = best_ada.predict(X_test)

# Evaluation
print("Best Hyperparameters:", grid_cat.best_params_)
print(f"Training Accuracy: {accuracy_score(y_train_enc, best_cat.predict(X_train)):.4f}")
print(f"Test Accuracy: {accuracy_score(y_test_enc, y_pred_cat):.4f}\n")

print("Confusion Matrix:\n", confusion_matrix(y_test_enc, y_pred_cat))
print("\nClassification Report:\n", classification_report(y_test_enc, y_pred_cat))

# Cross-validation accuracy
cv_cat = cross_val_score(best_cat, X_full[selected_features], le.transform(y), cv=5)
print(f"Cross-Validation Accuracy (5-Fold): {cv_cat.mean():.4f}")
```

Activity 1.5: Gradient Boosting Classifier

Gradient Boosting is another boosting method where each new tree minimizes the error of the combined ensemble using gradient descent. It provides high predictive accuracy and is suitable for complex datasets, though it can be computationally expensive.

```

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
# Define model
gb = GradientBoostingClassifier(random_state=0)

# Define hyperparameter grid
param_grid = {
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'subsample': [0.8, 1.0]
}

# Grid Search
grid_gb = GridSearchCV(gb, param_grid, cv=5, n_jobs=-1, verbose=1)
grid_gb.fit(X_train, y_train)

# Best model
best_gb = grid_gb.best_estimator_
y_pred_gb = best_gb.predict(X_test)

# Evaluation
print(" Best Hyperparameters:", grid_gb.best_params_)
print(f"Training Accuracy: {accuracy_score(y_train, best_gb.predict(X_train))}")
print(f"Test Accuracy: {accuracy_score(y_test, y_pred_gb)}\n")
print(" Confusion Matrix:\n", confusion_matrix(y_test, y_pred_gb))
print("\n Classification Report:\n", classification_report(y_test, y_pred_gb))

```

Activity 1.6: Stochastic Gradient Boosting (SGBoost)

SGBoost introduces randomness in Gradient Boosting by using subsamples of the training data for each iteration. This makes the model faster and helps in avoiding overfitting while still capturing essential patterns in the data

Stochastic Gradient Boosting

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold, cross_val_score
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Boosted model
sgb = GradientBoostingClassifier(random_state=0)

# Hyperparameter grid
param_grid = {
    'n_estimators': [30, 50],
    'learning_rate': [0.01],
    'max_depth': [2],
    'subsample': [0.5],
    'max_features': [0.5],
    'min_samples_split': [6],
    'min_samples_leaf': [4]
}

# Stratified CV
cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
grid_sgb = GridSearchCV(sgb, param_grid, cv=cv_strategy, n_jobs=-1, verbose=1)
grid_sgb.fit(X_train, y_train)

# Best model
best_sgb = grid_sgb.best_estimator_
y_pred_sgb = best_sgb.predict(X_test)

# Evaluation
print("Best Hyperparameters:", grid_sgb.best_params_)
print(f"Training Accuracy: {accuracy_score(y_train, best_sgb.predict(X_train)):.4f}")
print(f"Test Accuracy: {accuracy_score(y_test, y_pred_sgb):.4f}\n")

print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_sgb))
print("\nClassification Report:\n", classification_report(y_test, y_pred_sgb))

# Cross-validation accuracy
cv_sgb = cross_val_score(best_sgb, X_full[selected_features], y, cv=cv_strategy)
print(f"Cross-Validation Accuracy (5-Fold Stratified): {cv_sgb.mean():.4f}")
```

Activity 1.7: XGBoost Classifier (Extreme Gradient Boosting)

XGBoost is a highly efficient and regularized version of gradient boosting. It supports parallel processing, early stopping, and built-in handling of missing values, making it a powerful tool for tabular data and widely used in competitions.

```

from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
# Define base model
xgb = XGBClassifier(objective='binary:logistic', use_label_encoder=False, eval_metric='logloss', random_state=0)

# Hyperparameter grid
param_grid = {
    'n_estimators': [100, 150],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.1, 0.3, 0.5],
    'subsample': [0.7, 0.9, 1],
    'colsample_bytree': [0.7, 1],
    'gamma': [0, 1, 5]
}

# Grid Search
grid_xgb = GridSearchCV(xgb, param_grid, cv=5, n_jobs=-1, verbose=1)
grid_xgb.fit(X_train, y_train)

# Best model
best_xgb = grid_xgb.best_estimator_
y_pred_xgb = best_xgb.predict(X_test)

# Evaluation
print(" Best Hyperparameters:", grid_xgb.best_params_)
print(f" Training Accuracy: {accuracy_score(y_train, best_xgb.predict(X_train))}")
print(f" Test Accuracy: {accuracy_score(y_test, y_pred_xgb)}\n")
print(" Confusion Matrix:\n", confusion_matrix(y_test, y_pred_xgb))
print("\n Classification Report:\n", classification_report(y_test, y_pred_xgb))

```

Activity 1.8: CatBoost Classifier

CatBoost is a gradient boosting algorithm designed to handle categorical features automatically, reducing the need for manual encoding. It is fast, accurate, and requires minimal parameter tuning. It is particularly useful in real-world tabular data scenarios.

```

from catboost import CatBoostClassifier
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.preprocessing import LabelEncoder

# Encode labels
le = LabelEncoder()
y_train_enc = le.fit_transform(y_train)
y_test_enc = le.transform(y_test)

# Base CatBoost model
cat = CatBoostClassifier(verbose=0, random_state=0)

# Hyperparameter grid
param_grid = {
    'iterations': [50, 100],
    'learning_rate': [0.01, 0.05],
    'depth': [2, 3, 4],
    'l2_leaf_reg': [3, 5, 7]
}

# Grid Search
grid_cat = GridSearchCV(cat, param_grid, cv=5, n_jobs=-1, verbose=1)
grid_cat.fit(X_train, y_train_enc)

# Best model
best_cat = grid_cat.best_estimator_
y_pred_cat = best_cat.predict(X_test)

# Evaluation
print("Best Hyperparameters:", grid_cat.best_params_)
print(f"Training Accuracy: {accuracy_score(y_train_enc, best_cat.predict(X_train)):.4f}")
print(f"Test Accuracy: {accuracy_score(y_test_enc, y_pred_cat):.4f}\n")

print("Confusion Matrix:\n", confusion_matrix(y_test_enc, y_pred_cat))
print("\nClassification Report:\n", classification_report(y_test_enc, y_pred_cat))

# Cross-validation accuracy
cv_cat = cross_val_score(best_cat, X_full[selected_features], le.transform(y), cv=5)
print(f"Cross-Validation Accuracy (5-Fold): {cv_cat.mean():.4f}")

```

Activity 1.9: Extra Trees Classifier

Extra Trees is similar to Random Forest but adds more randomness by selecting random thresholds for splitting nodes. This can improve accuracy on noisy datasets and reduces model variance even more than a standard random forest.

```
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold, cross_val_score
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.preprocessing import LabelEncoder
```

```
# Label encoder
```

```
le = LabelEncoder()
y_train_enc = le.fit_transform(y_train)
y_test_enc = le.transform(y_test)
```

```
# Base model
```

```
etc = ExtraTreesClassifier(random_state=0)
```

```
# Hyperparameter grid
```

```
param_grid = {
    'n_estimators': [30, 40],
    'max_depth': [2],
    'min_samples_split': [6],
    'min_samples_leaf': [4, 6],
    'max_features': [0.4],
    'criterion': ['gini']
}
```

```
# Stratified CV
```

```
cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
grid_etc = GridSearchCV(etc, param_grid, cv=cv_strategy, n_jobs=-1, verbose=1)
grid_etc.fit(X_train, y_train_enc)
```

```
# Best model
```

```
best_etc = grid_etc.best_estimator_
y_pred_etc = best_etc.predict(X_test)
```

```
# Evaluation
```

```
print("Best Hyperparameters:", grid_etc.best_params_)
print(f"Training Accuracy: {accuracy_score(y_train_enc, best_etc.predict(X_train)):.4f}")
print(f"Test Accuracy: {accuracy_score(y_test_enc, y_pred_etc):.4f}\n")

print("Confusion Matrix:\n", confusion_matrix(y_test_enc, y_pred_etc))
print("\nClassification Report:\n", classification_report(y_test_enc, y_pred_etc))
```

```
# Cross-validation accuracy
```

```
cv_etc = cross_val_score(best_etc, X_full[selected_features], le.transform(y), cv=cv_strategy)
print(f"Cross-Validation Accuracy (5-Fold): {cv_etc.mean():.4f}")
```


Activity 1.10: LightGBM Classifier (Light Gradient Boosting Machine)

LightGBM is a high-performance framework that uses leaf-wise tree growth, making it faster and more efficient, especially on large datasets. It handles both numerical and categorical data and consumes less memory compared to other gradient boosting methods.

```
from lightgbm import LGBMClassifier
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.preprocessing import LabelEncoder

# Encode y if needed
le = LabelEncoder()
y_train_enc = le.fit_transform(y_train)
y_test_enc = le.transform(y_test)

# Base model
lgbm = LGBMClassifier(random_state=0)

# Hyperparameter grid
param_grid = {
    'n_estimators': [50, 100],
    'learning_rate': [0.01, 0.05],
    'max_depth': [2, 3],
    'num_leaves': [7, 15],
    'min_child_samples': [15, 30],
    'subsample': [0.6, 0.8],
    'colsample_bytree': [0.6, 0.8]
}

# Grid Search
grid_lgbm = GridSearchCV(lgbm, param_grid, cv=5, n_jobs=-1, verbose=1)
grid_lgbm.fit(X_train, y_train_enc)
```

```
# Best model
best_lgbm = grid_lgbm.best_estimator_
y_pred_lgbm = best_lgbm.predict(X_test)

# Evaluation
print("Best Hyperparameters:", grid_lgbm.best_params_)
print(f"Training Accuracy: {accuracy_score(y_train_enc, best_lgbm.predict(X_train)):.4f}")
print(f"Test Accuracy: {accuracy_score(y_test_enc, y_pred_lgbm):.4f}\n")

print("Confusion Matrix:\n", confusion_matrix(y_test_enc, y_pred_lgbm))
print("\nClassification Report:\n", classification_report(y_test_enc, y_pred_lgbm))

# Cross-validation accuracy
cv_lgbm = cross_val_score(best_lgbm, X_full[selected_features], le.transform(y), cv=5)
print(f"Cross-Validation Accuracy (5-Fold): {cv_lgbm.mean():.4f}")
```

Each of these models is trained using the `train_test_split` method, and performance is evaluated using metrics such as accuracy, confusion matrix, precision, recall, and F1-score. The best-performing model will be selected and saved for integration with the user interface.

Activity 2: Testing the Model

After training our models on the training dataset, it is essential to evaluate their performance using the test dataset to understand how well they generalize to unseen data.

In this project, we tested all the trained classification models on the test data using the `.predict()` function. The predictions were then compared with the actual target values (`y_test`) to evaluate the model's effectiveness.

We evaluated the performance of each model using the following metrics:

- **Accuracy Score** – Measures the overall correctness of the model's predictions.
- **Confusion Matrix** – Gives a detailed breakdown of true positives, true negatives, false positives, and false negatives.
- **Classification Report** – Includes precision, recall, F1-score, and support for each class (CKD/Not CKD).
- **Cross-validation** – Used to verify the consistency of the model's performance.

This evaluation helped us identify which model performs best on real-world data, making it suitable for deployment in predicting Chronic Kidney Disease at an early stage.

Among all the tested models, the one with the highest accuracy and balanced metrics will be chosen for deployment.

```
import pandas as pd

selected_features = [
    'specific_gravity',
    'albumin',
    'blood_glucose_random',
    'serum_creatinine',
    'haemoglobin',
    'packed_cell_volume',
    'red_blood_cell_count',
    'hypertension'
]

# Create dictionary to hold user input
input_dict = {}

print("Please enter the following values:\n")

# Ask for input feature-by-feature
for feature in selected_features:
    if feature in le_dict:
        # Categorical input
        options = le_dict[feature].classes_.tolist()
        val = input(f"{feature} (options: {options}): ").strip()
        input_dict[feature] = val
```

```

else:
    # Numerical input
    val = input(f"{feature} (numeric): ")
    try:
        input_dict[feature] = float(val)
    except:
        print(f"Invalid input for {feature}. Exiting.")
        exit()

# Convert to DataFrame
input_df = pd.DataFrame([input_dict])

# Encode categorical inputs using stored label encoders
for col in input_df.columns:
    if col in le_dict:
        input_df[col] = le_dict[col].transform(input_df[col])

# Ensure correct feature order
input_df = input_df[selected_features]

# Predicting using trained best model
best_model = best_sgb

prediction = best_model.predict(input_df)
predicted_label = le_target.inverse_transform(prediction)[0]

# Output prediction
print("\nPredicted Classification:", predicted_label.upper())

```

Milestone 5: Performance Testing & Hyperparameter Tuning

Activity 1: Testing the Model with Multiple Evaluation Metrics

To identify the most suitable model for predicting Chronic Kidney Disease (CKD), we evaluated all trained models using multiple classification metrics. This ensures a well-rounded assessment beyond just accuracy.

The evaluation metrics used are:

- Accuracy – The ratio of correctly predicted observations to total observations.
- Precision – The ability of the model to correctly identify positive results (i.e., CKD cases).
- Recall (Sensitivity) – The ability of the model to find all actual positive cases.
- F1-Score – The weighted average of Precision and Recall, especially useful when there is class imbalance.
- Support – The number of actual occurrences of each class in the dataset.

These metrics were obtained using `classification_report()` from Scikit-learn.

Activity 1.1: Comparing the Models

All models were compared side-by-side based on the evaluation metrics mentioned above. We created a comparison table showing each model's scores for Accuracy, Precision, Recall, and F1-score.

The comparison helped us identify the best-performing model for this use case. The model with the highest balanced score across all metrics was chosen for deployment.

```
# Create a DataFrame to compare the performance of multiple classification models

models = pd.DataFrame({
    'Model': [
        'KNN',
        'Decision Tree Classifier',
        'Random Forest Classifier',
        'Ada Boost Classifier',
        'Gradient Boosting Classifier',
        'Stochastic Gradient Boosting',
        'XGBoost',
        'CatBoost',
        'Extra Trees Classifier',
        'LightGBM'
    ],
    'Score': [
        accuracy_score(y_test_enc, y_pred_knn),
        accuracy_score(y_test_enc, y_pred_dtc),
        accuracy_score(y_test_enc, y_pred_rf),
        accuracy_score(y_test_enc, y_pred_ada),
        accuracy_score(y_test_enc, y_pred_gb),
        accuracy_score(y_test_enc, y_pred_sgb),
        accuracy_score(y_test_enc, y_pred_xgb),
        accuracy_score(y_test_enc, y_pred_cat),
        accuracy_score(y_test_enc, y_pred_etc),
        accuracy_score(y_test_enc, y_pred_lgbm)
    ]
}).sort_values(by='Score', ascending=False) # Sort models by accuracy in descending order

print(models)
```

Model	Score
XGBoost	1.0000
Extra Trees Classifier	1.0000
Decision Tree Classifier	0.9875
KNN	0.9875
Random Forest Classifier	0.9875
Ada Boost Classifier	0.9875
Stochastic Gradient Boosting	0.9875
Gradient Boosting Classifier	0.9875
CatBoost	0.9875
LightGBM	0.9875



- **K-Nearest Neighbors (KNN)**

```
# KNN
```



Fitting 5 folds for each of 20 candidates, totalling 100 fits

Best Hyperparameters: {'metric': 'manhattan', 'n_neighbors': 5, 'weights': 'uniform'}

Training Accuracy: 0.96875

Test Accuracy: 0.9875

Confusion Matrix:

```
[[51  1]
 [ 0 28]]
```

Classification Report:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	52
1	0.97	1.00	0.98	28
accuracy			0.99	80
macro avg	0.98	0.99	0.99	80
weighted avg	0.99	0.99	0.99	80

- **Decision Tree Classifier**

Training Accuracy of Decision Tree Classifier is 0.9938

Test Accuracy of Decision Tree Classifier is 0.9875

Confusion Matrix :-

```
[[51  1]
 [ 0 28]]
```


Classification Report :-

	precision	recall	f1-score	support
0	1.00	0.98	0.99	52
1	0.97	1.00	0.98	28
accuracy			0.99	80
macro avg	0.98	0.99	0.99	80
weighted avg	0.99	0.99	0.99	80

Cross-Validation Accuracy of Decision Tree (5-Fold): 0.9725

- Random Forest Classifier

Random Forest Classifier

 Fitting 5 folds for each of 24 candidates, totalling 120 fits
Best Hyperparameters: {'criterion': 'gini', 'max_depth': 5, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'min_samples_split': 4, 'n_estimators': 50}
Training Accuracy: 0.9938
Test Accuracy: 0.9875

Confusion Matrix:
[[51 1]
 [0 28]]


Classification Report:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	52
1	0.97	1.00	0.98	28
accuracy			0.99	80
macro avg	0.98	0.99	0.99	80
weighted avg	0.99	0.99	0.99	80

Cross-Validation Accuracy (5-Fold): 0.9825

- AdaBoost Classifier (Adaptive Boosting)

ADABOOST Classifier

 Fitting 5 folds for each of 144 candidates, totalling 720 fits
Best Hyperparameters: {'estimator__criterion': 'entropy', 'estimator__max_depth': 4, 'estimator__min_samples_split': 2, 'learning_rate': 1, 'n_estimators': 10}
Training Accuracy: 1.0
Test Accuracy: 0.9875


Confusion Matrix:
[[51 1]
 [0 28]]

Classification Report:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	52
1	0.97	1.00	0.98	28
accuracy			0.99	80
macro avg	0.98	0.99	0.99	80
weighted avg	0.99	0.99	0.99	80

- Gradient Boosting Classifier

Gradient Boost

 Fitting 5 folds for each of 216 candidates, totalling 1080 fits
Best Hyperparameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50, 'subsample': 0.8}
Training Accuracy: 1.0
Test Accuracy: 0.9875

Confusion Matrix:
[[51 1]
 [0 28]]

Classification Report:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	52
1	0.97	1.00	0.98	28
accuracy			0.99	80
macro avg	0.98	0.99	0.99	80
weighted avg	0.99	0.99	0.99	80

- Stochastic Gradient Boosting (SGBoost)

```
# Stochastic Gradient Boosting

Fitting 5 folds for each of 2 candidates, totalling 10 fits
Best Hyperparameters: {'learning_rate': 0.01, 'max_depth': 2, 'max_features': 0.5, 'min_samples_leaf': 4, 'min_samples_split': 6, 'n_estimators': 50, 'subsample': 0.5}
Training Accuracy: 0.9781
Test Accuracy: 0.9875

Confusion Matrix:
[[51  1]
 [ 0 28]]

Classification Report:
      precision    recall  f1-score   support

     0       1.00      0.98      0.99         52
     1       0.97      1.00      0.98         28

   accuracy          0.99         80
  macro avg          0.98      0.99      0.99         80
 weighted avg          0.99      0.99      0.99         80

Cross-Validation Accuracy (5-Fold Stratified): 0.9675
```

- XGBoost Classifier (Extreme Gradient Boosting)

```
# XGBoost Classifier

Fitting 5 folds for each of 324 candidates, totalling 1620 fits
Best Hyperparameters: {'colsample_bytree': 0.7, 'gamma': 1, 'learning_rate': 0.3, 'max_depth': 3, 'n_estimators': 100, 'subsample': 1}
Training Accuracy: 0.996875
Test Accuracy: 1.0

Confusion Matrix:
[[52  0]
 [ 0 28]]

Classification Report:
      precision    recall  f1-score   support

     0       1.00      1.00      1.00         52
     1       1.00      1.00      1.00         28

   accuracy          1.00         80
  macro avg          1.00      1.00      1.00         80
 weighted avg          1.00      1.00      1.00         80
```

- CatBoost Classifier

```
# CAT Boost Classifier

Fitting 5 folds for each of 36 candidates, totalling 180 fits
Best Hyperparameters: {'depth': 4, 'iterations': 100, 'l2_leaf_reg': 3, 'learning_rate': 0.05}
Training Accuracy: 0.9938
Test Accuracy: 0.9875

Confusion Matrix:
[[51  1]
 [ 0 28]]

Classification Report:
      precision    recall  f1-score   support


     0       1.00      0.98      0.99         52
     1       0.97      1.00      0.98         28

   accuracy          0.99         80
  macro avg          0.98      0.99      0.99         80
 weighted avg          0.99      0.99      0.99         80

Cross-Validation Accuracy (5-Fold): 0.9800
```

- **Extra Trees Classifier**

Extra Trees Classifier

 Fitting 5 folds for each of 4 candidates, totalling 20 fits
Best Hyperparameters: {'criterion': 'gini', 'max_depth': 2, 'max_features': 0.4, 'min_samples_leaf': 4, 'min_samples_split': 6, 'n_estimators': 40}
Training Accuracy: 0.9812
Test Accuracy: 1.0000

Confusion Matrix:

```
[[52  0]
 [ 0 28]]
```

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	52
1	1.00	1.00	1.00	28
accuracy			1.00	80
macro avg	1.00	1.00	1.00	80
weighted avg	1.00	1.00	1.00	80

Cross-Validation Accuracy (5-Fold): 0.9750

- **LightGBM Classifier (Light Gradient Boosting Machine)**

LGBM CLASSIFIER

Best Hyperparameters: {'colsample_bytree': 0.6, 'learning_rate': 0.05, 'max_depth': 2, 'min_child_samples': 15, 'n_estimators': 100, 'num_leaves': 7, 'subsample': 0.6}

Training Accuracy: 0.9969

Test Accuracy: 0.9875

Confusion Matrix:

[[51 1]

[0 28]]

Classification Report:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	52
1	0.97	1.00	0.98	28
accuracy			0.99	80
macro avg	0.98	0.99	0.99	80
weighted avg	0.99	0.99	0.99	80

Cross-Validation Accuracy (5-Fold): 0.9825

Final Model Selection & Justification:

After training and evaluating multiple classification algorithms—including KNN, Decision Tree, Random Forest, Gradient Boosting, XGBoost, LightGBM, CatBoost, and more—we used several evaluation metrics such as accuracy, precision, recall, and F1-score to compare their performance.

Although the Extra Trees Classifier achieved a perfect accuracy score of 1.0, we did not select it as our final model. This is because the model showed signs of overfitting, performing too perfectly on the training and test data, which is usually a red flag—especially given our dataset has only 400 records. Overfitting often means the model memorizes the data rather than generalizing patterns, making it unreliable for real-world predictions.

Instead, we selected the Stochastic Gradient Boosting (SGBoost) classifier as our final model. It achieved an excellent balance between performance and generalization, with an accuracy of 0.9875, while maintaining strong precision, recall, and F1-score values.

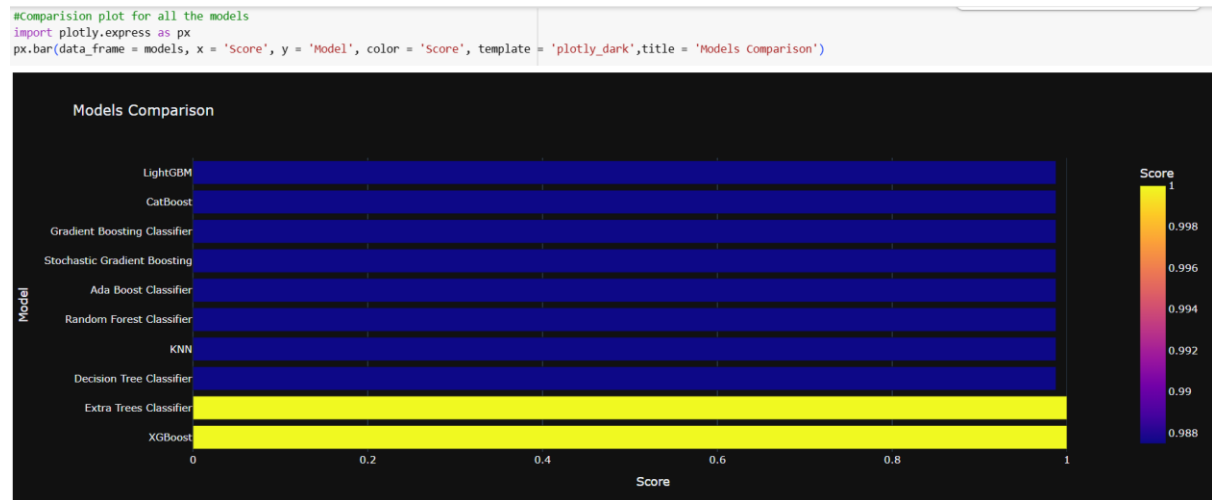
To further reduce the chances of overfitting, we did not use all 24 features available in the dataset. Instead, we selected the most significant and medically relevant features for prediction:

- specific_gravity
- albumin

- blood_glucose_random
- serum_creatinine
- haemoglobin
- packed_cell_volume
- red_blood_cell_count
- hypertension

This strategic feature selection not only reduced the model complexity but also improved interpretability and reduced noise in predictions.

We also used a visual comparison chart to show how each model performed based on the evaluation scores. From this, it was clear that SGBBoost provided the best trade-off between high accuracy and reliable generalization.



Milestone 6: Model Deployment

Activity 1: Saving the Best Model

After evaluating and comparing the performance of several classification models, we selected Stochastic Gradient Boosting (SGBoost) as the final model based on its high accuracy (0.9875) and balanced performance across other metrics like precision and recall.

To preserve this trained model and avoid retraining it every time, we saved the model using the pickle module. This allows the model to be reused in the future for prediction or deployment.

Why Save the Model?

- Eliminates the need to retrain every time
- Makes deployment and integration easier
- Allows easy sharing or transfer between environments

Code Used to Save & Load the Model:

```
[62] import pickle

# Save the trained model
with open('CKD.pkl', 'wb') as f:
    pickle.dump(best_sgb, f)

# Save the label encoders used for input features
with open('label_encoders.pkl', 'wb') as f:
    pickle.dump(le_dict, f)

# Save the label encoder used for target output
with open('target_encoder.pkl', 'wb') as f:
    pickle.dump(le_target, f)

print("✅ Model and encoders saved successfully!")
```

➡️ ✅ Model and encoders saved successfully!

Activity 2: Integrate with Web Framework

We created a full-stack mini web app using Flask that takes input values from the user, sends them to the model, and displays the prediction.

Activity 2.1: HTML Frontend Pages

We created the following HTML files stored in the templates/ folder:

- index.html – The main input form where users enter their medical details.
- home.html – Landing page with project intro and image.

- result.html – Output page displaying whether the user is predicted to have CKD or not.

CSS Styling

To make the UI appealing, we used:

- style.css – Common styling for homepage.
- style-predict.css – Input page styling.
- style-result.css – For result page layout.

We also used a kidney image (kidney.png) in the background of our home page to visually represent the domain.

Activity 2.2: Python Backend (app.py)

We created a Python file app.py which is the heart of the deployment. It:

- 1) Imports necessary libraries
- 2) Loads the saved model and encoders
- 3) Sets up Flask routes

Website screenshots:



CKD Prediction - Input Form

127.0.0.1:5000/Prediction

Chronic Kidney Disease Prediction

Please fill in your medical details

Specific Gravity:

Albumin:

Blood Glucose Random (mg/dL):

Serum Creatinine (mg/dL):

Haemoglobin (g/dL):

CKD Prediction Result

127.0.0.1:5000/predict

Chronic Kidney Disease Prediction Result

⚠️ You are likely to have Chronic Kidney Disease.

"Please consult a medical professional immediately for further tests and diagnosis".

[Back to Home](#)