

# Chronic Kidney Disease Prediction Using Machine Learning Algorithms

Machine Learning in  
Data Processing and Analytics



Team Name:	Dragon Tamers
Team Coordinator:	Rolfis Ramses Solano Mendez
Team Members:	Bianca-Gabriela Leoveanu, Rolfis Ramses Solano Mendez, Costa Massena

# 1 Project Description

This project explores the use of machine learning techniques to analyze a clinical dataset related to chronic kidney disease (CKD). Our work involves building a complete data pipeline, including pre-processing and feature selection, followed by the application of K-Means Clustering, Naive Bayes and Decision Tree models. The aim is to discover meaningful patterns, evaluate model performance and support early identification of chronic kidney disease through data-driven results.

## 2 Data Source and Characteristics

### 2.1 Source

This data comes from the UC Irvine Machine learning Repository, a collection of databases, domain theories, and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms.

### 2.2 Dataset

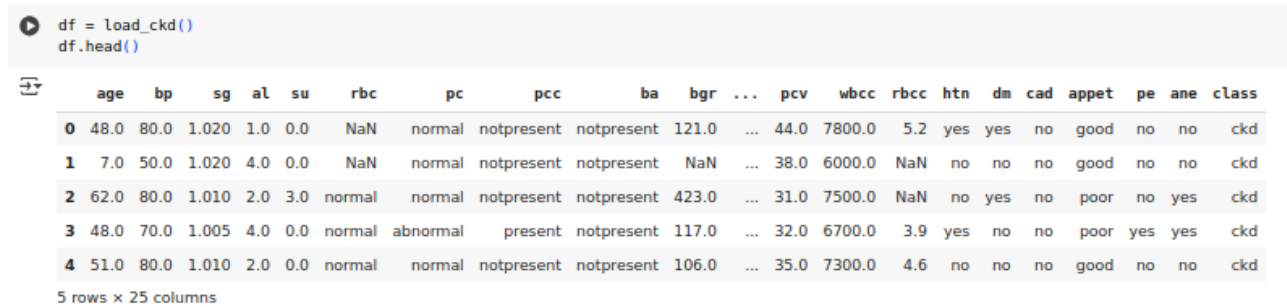
The dataset we chose is the results of a 2 month period where 25 characteristics of 400 patients suspected of having a chronic kidney disease were collected. Those features are such:

- **age** in years
- **bp** patient's blood pressure in mm/Hg
- **sg** patient's specific gravity (categorical)
- **al** patient's albumin (categorical)
- **su** patient's sugar (categorical)
- **rbc** patient's red blood cells (binary)
- **pc** patient's pus cell (binary)
- **pcc** patient's pus cell clumps (binary)
- **ba** patient's bacteria (binary)
- **bgr** patient's blood glucose random in mgs/dl
- **bu** patient's blood urea in mgs/dl
- **sc** patient's serum creatinine in mgs/dl
- **sod** patient's sodium in mEq/L
- **pot** patient's potassium in mEq/L
- **hemo** patient's hemoglobin in gms
- **pcv** patient's packed cell volume
- **wbcc** patient's white blood cell count in cells/cmm
- **rbcc** patient's red blood cell count in millions/cmm
- **htn** patient's hypertension (binary)
- **dm** patient's diabetes mellitus (binary)
- **cad** patient's coronary artery disease (binary)
- **appet** patient's appetite (binary)
- **pe** patient's pedal edema (binary)
- **ane** patient's anemia (binary)

And finally one target:

- **class** "ckd" if the patient has the disease, "not ckd" otherwise

All these features have missing values, and they are not from the same type, which we need to keep in mind when pre-processing the data. Let's display the dataframe head:



```
df = load_ckd()
df.head()
```

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv	wbcc	rbcc	htn	dm	cad	appet	pe	ane	class
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44.0	7800.0	5.2	yes	yes	no	good	no	no	ckd
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38.0	6000.0	NaN	no	no	no	good	no	no	ckd
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31.0	7500.0	NaN	no	yes	no	poor	no	yes	ckd
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...	32.0	6700.0	3.9	yes	no	no	poor	yes	yes	ckd
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...	35.0	7300.0	4.6	no	no	no	good	no	no	ckd

5 rows x 25 columns

Figure 1: dataset head

### 3 Data Pre-processing

This section presents the full pipeline used to prepare the Chronic Kidney Disease dataset for the machine learning algorithms. It covers data cleaning, missing value handling, encoding, scaling, and feature selection.

#### 3.1 Formatting and Cleaning

Some values were malformed due to tab characters. These were corrected manually and the rows with missing targets and duplicates were removed.

```
df['dm'] = df['dm'].replace('\tno', 'ckd')
df['class'] = df['class'].replace('ckd\t', 'ckd')

df.dropna(subset=['class'], inplace = True)
df.drop_duplicates(inplace=True)
df.reset_index(drop=True, inplace=True)
```

#### 3.2 Removing Columns with Too Much Missing Data

Columns with missing values greater than 25% were eliminated to improve the reliability of the model.

```
drop_thresh = 0.25
df = df.loc[:, df.isnull().mean() <= drop_thresh]
```

#### 3.3 Identifying Column Types

We identified which columns were numerical or categorical, excluding the target.

```
numerical_cols = df.select_dtypes(include=[np.number]).columns.tolist()
categorical_cols = df.select_dtypes(include=['object', 'category', 'bool']).columns.tolist()
if 'class' in categorical_cols:
    categorical_cols.remove('class')
```

#### 3.4 Missing Value Imputation

Missing values in numerical features were filled based on skewness:

- Mean, if skew  $\in [-1, 1]$
- Median, otherwise

The categorical columns were filled with their mode.

```

for col in numerical_cols:
    skew_val = df[col].skew()
    if -1 <= skew_val <= 1:
        df[col] = df[col].fillna(df[col].mean())
    else:
        df[col] = df[col].fillna(df[col].median())

for col in categorical_cols:
    df[col] = df[col].fillna(df[col].mode()[0])

```

### 3.5 Categorical Encoding

Binary categorical features were label-encoded. Multiclass features were one-hot encoded, excluding the first category to avoid multicollinearity. The target 'class' was also label-encoded.

```

label_mappings = {}
binary_cats = [col for col in categorical_cols if df[col].nunique() == 2]
for col in binary_cats:
    vals = df[col].dropna().unique()
    label_mappings[col] = {val: idx for idx, val in enumerate(vals)}
    df[col] = df[col].map(label_mappings[col])

multi_cat_cols = [col for col in categorical_cols if col not in binary_cats]
df = pd.get_dummies(df, columns=multi_cat_cols, drop_first=True)

from sklearn.preprocessing import LabelEncoder
target_encoder = LabelEncoder()
df['class'] = target_encoder.fit_transform(df['class'])

```

### 3.6 Scaling the Data

Standard scaling was applied to ensure compatibility with models sensitive to feature magnitude. Features were scaled to have mean=0 and std=1.

```

from sklearn.preprocessing import StandardScaler

X = df.drop(columns='class')
y = df['class']

scaler = StandardScaler()
X_scaled = pd.DataFrame(scaler.fit_transform(X), columns=X.columns)

```

### 3.7 Correlation Matrix Heatmap

To assess linear relationships among features and their relevance to the target variable, we generated a correlation matrix heatmap, shown in Figure 2. This matrix quantifies the Pearson correlation coefficient between every pair of features, including the target.

A correlation value ranges from:

- **+1** - perfect positive linear relationship,
- **0** - no linear relationship,
- **-1** - perfect negative linear relationship.

This analysis helps identify multicollinearity, potential feature redundancy, and feature redundancy, and features that may be predictive of the target.

Figure 2 shows that the hemo, pcv and sg features are positively correlated with the target, while al, bu and sc features are negatively correlated. Strong correlations between hemo and pcv, and bu and sc suggest potential feature redundancy.

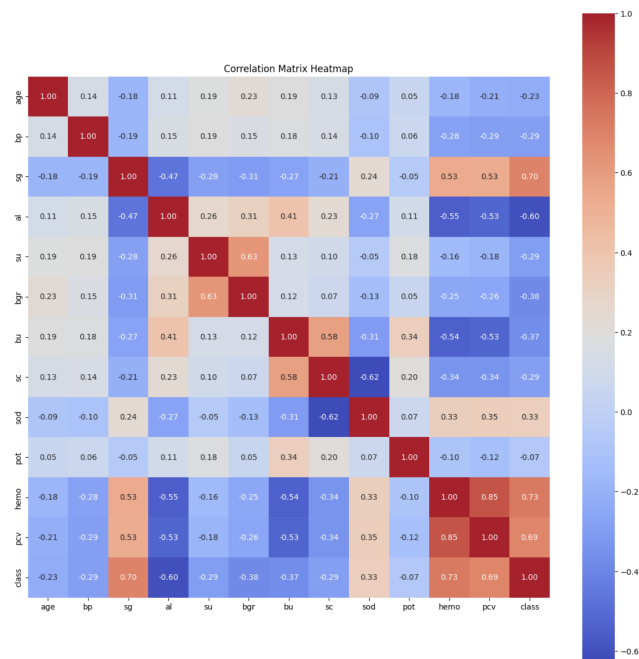


Figure 2: Correlation Matrix Heatmap

### 3.8 Feature Variance Analysis

To evaluate the importance and variability of each numerical feature, we calculated their variance values. Their printed values are shown in Figure 3 and their respective bar chart is shown in Figure 4. Features with extremely low variance contribute little to distinguishing between classes and may be removed during dimensionality reduction or feature selection.

Figure 3 shows that bgr and bu features have the highest variance, suggesting a wide spread in their values and a strong potential to influence the mode. In contrast, sg, pot and hemo features show very low variance, indicating limited variability across the dataset.

Based on this analysis, low-variance features were marked for potential exclusion to simplify the model and reduce noise.

```

Feature Variances:
age      288.149479
bp       181.974311
sg        0.000029
al       1.618793
su       1.081679
bgr     5664.184054
bu      2439.891530
sc       31.684359
sod      84.756540
pot       7.960058
hemo     7.377586
pcv     66.440128
dtype: float64

```

Figure 3: Printed Feature Variance Values

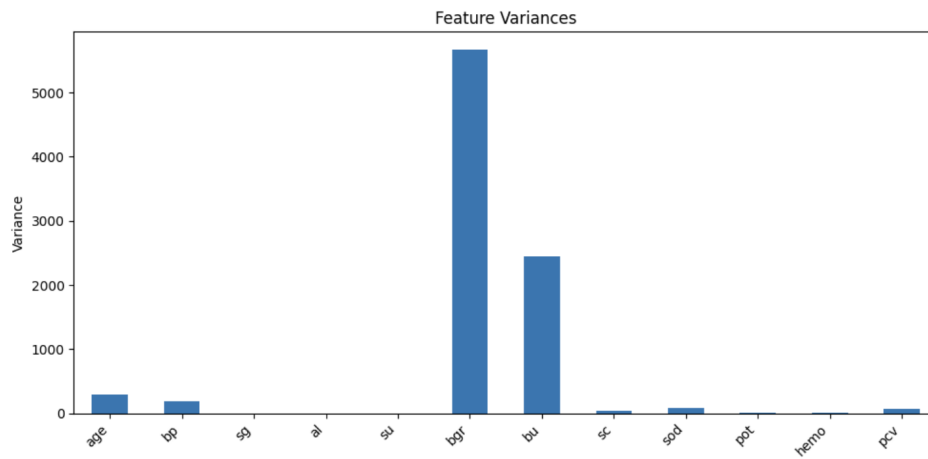


Figure 4: Bar Chart of Features Variance Values

### 3.9 Feature Selection

Following the variance analysis, a feature selection step was applied to eliminate attributes with very low variability. Using a variance threshold of 0.01, only features with sufficient spread across samples were kept. As a result, features like `sg`, which showed extremely low variance in Figure 3 and Figure 4, were removed from the dataset.

The filtered features were dropped using the following code:

```
low_variance_cols = variances[variances < 0.01].index.tolist()
X_selected = X.drop(columns=low_variance_cols)
X_scaled_selected = X_scaled.drop(columns=low_variance_cols)
```

This process produced two optimized feature datasets:

- `X_selected` for use with models that don't require scaling, like Decision Trees
- `X_scaled_selected` for models that are scale-sensitive, like Naive Bayes and K-Means Clustering

## 4 Definition and Short Characteristics

### 4.1 K-means Clustering

#### 4.1.1 Definition and Main Idea

K-Means is an unsupervised machine learning algorithm used for partitioning data into  $k$  distinct clusters based on feature similarity. It minimizes the distance between data points and their assigned centroids.

#### 4.1.2 How it works?

The algorithm can be broken down into five steps:

1. Choose the number of clusters  $k$ .
2. Initialize randomly  $k$  centroids.
3. Assign data points to the nearest cluster centroid using the Euclidean distance metric.
4. Re-initialize centroids as the mean of the assigned points.
5. Repeat steps 3 and 4 until convergence.

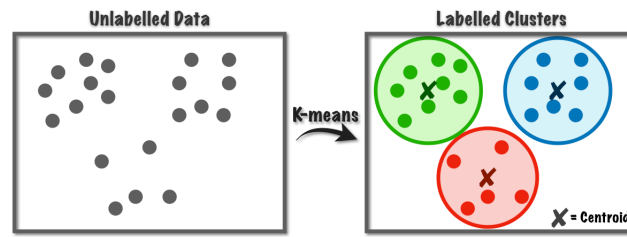


Figure 5: K-Means Algorithm Mechanism<sup>1</sup>

#### 4.1.3 Typical Use Cases

- **Customer Segmentation in Marketing:** Identify user groups for targeted marketing.
- **Image compression:** Cluster similar pixels to reduce color palette.
- **Anomaly detection in Cybersecurity:** Detect outliers or anomalies in network activity.
- **Healthcare data analysis:** Group patients with similar symptoms or outcomes.
- **Recommendation systems:** Cluster similar users/items to improve suggestions.

#### 4.1.4 Advantages

- **Simplicity and Ease of Use:** Easy to understand and implement.
- **Efficiency and Speed:** Fast to compute and works well with large datasets.
- **Scalability:** Perform efficiently as the dataset size grows.
- **Unsupervised Learning:** Does not require labeled data to find structure and patterns.
- **Customizable Number of Clusters:** Allows users to choose  $k$ .

#### 4.1.5 Disadvantages

- **Sensitive to Initial Placement:** Different results depending on centroids.
- **Assumption of Equal-Sized Clusters and Spherical Shapes:** Prefers spherical, equal-sized clusters.
- **Dependence on Number of Clusters:** Must define number of clusters.
- **Sensitive to Outliers:** Centroids affected by anomalies.
- **Not Suitable for Non-Linear Data:** Struggles with non-linear data.

## 4.2 Decision Tree Algorithm (DT)

### 4.2.1 Definition and Main Idea

Decision tree is a supervised learning algorithm which represents decisions and their consequences in a tree. It can be used for classification and regression, which we will see examples of.

At every knot in that tree, we test a feature. Each branch represents the result of a test, and each leaf represents the final decision (which class does it belong to for classification, and the value for regression).

<sup>1</sup>Source: <https://images.app.goo.gl/Dtrziw1K2JY5FmUa9>

### 4.2.2 How it works?

The decision tree algorithm can be broken down into five steps:

#### For classification

1. Calculate Initial Impurity (using Gini or Entropy)
2. Evaluate All Possible Splits: for each feature and split value, divide it into left and right subsets and calculate the weighted average impurity of the split, and calculate  
$$\text{Information Gain} = \text{Initial Impurity} - \text{Weighted Average Impurity}$$
3. Select Best Split: Choose the feature and split value that maximizes Information Gain and create a decision node with this split
4. Recursive Splitting: Apply steps 1-3 recursively to each subset until the conditions are met (Pure node, which means all samples belong to same class, Minimum samples per node reached, Maximum depth reached, No significant improvement in impurity).
5. Assign Class Labels

#### For regression

1. Calculate Initial Variance: Calculate the variance of target values in the entire dataset
2. Evaluate All Possible Splits: For each feature and each possible split value we divide the data into left and right subsets, calculate the mean target value for each subset, and calculate the weighted variance reduction as such:  
$$\text{Variance Reduction} = \text{Initial Variance} - \text{Weighted Average Variance}$$
3. Select Best Split: Choose the feature and split value that maximizes variance reduction, this minimizes the Mean Squared Error (MSE)
4. Recursive Splitting: Apply steps 1-3 recursively to each subset until the conditions are met (Variance below threshold, Minimum samples per node reached, Maximum depth reached, No significant improvement in variance)
5. Assign Predicted Values

### 4.2.3 Typical Use Cases

#### For Classification

- **Email Spam Detection:** Identifying spam vs. legitimate emails.
- **Medical Diagnosis:** Heart disease risk assessment.
- **Customer Churn Prediction:** Anticipate if a customer will stop using a company's product or service during a specific time period.
- **Loan Approval:** Deciding whether to approve loan applications.

#### For Regression

- **House Price Prediction:** Estimating property values.
- **Employee Salary Prediction:** Using diploma and experience to calculate an employee's wage.
- **Stock Price Movement:** Predicting next-day price changes.
- **Energy Consumption Forecasting:** Estimating power usage.

### 4.2.4 Advantages

- **Interpretability:** Trees are easily understandable and visualizable, even by non-experts.
- **Data Type Handling:** Can handle both numerical and categorical variables.
- **Data Preparation:** Little preparation needed, no normalization or standardization required.
- **Automatic Feature Selection:** Algorithm automatically selects the most important variables.



### 4.2.5 Disadvantages

- **Overfitting:** Tendency to create overly complex trees that memorize noise in training data.
- **Instability:** Small changes in data can produce very different trees.
- **Bias:** Favors variables with more categories during split selection.
- **Limited Performance:** Generally less performant than tree ensembles (Random Forest, Gradient Boosting).

## 4.3 Naive Bayes (NB)

### 4.3.1 Definition and Main Idea

Naive Bayes is a statistical machine learning algorithm used for classification tasks. It is called naive because it assumes that all features are independent. Although this case rarely presents in real life, it works very well in practice. The name Bayes comes from the Bayes Theorem, on which the algorithm is based. It calculates the probability for each class and selects the class with the highest probability. There are 3 main variants of the Naive Bayes classifiers: Gaussian, Multinomial and Bernoulli.

### 4.3.2 How it works?

The algorithm can be broken down into the following steps:

1. Calculate the prior probabilities for each class.
2. Calculate the likelihood for each feature (the way of calculation depends on the variant).
3. Compute the posterior score for each class.
4. Pick the class with the highest score.

#### Bernoulli NB

The Bernoulli NB is based on the assumption that, even having multiple features, each one of them is binary-valued, having just 2 possible categories.

#### Multinomial NB

The Multinomial NB assumes, as its name says, that the data is distributed in a multinomially form, where we have more than two categories and multiple counts per category.

#### Gaussian NB

This variant of the Naive Bayes algorithm assumes that the likelihood of a feature is Gaussian. The following formula is used to calculate this likelihood:

$$P(x | C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}}$$

The parameters  $\sigma^2$  and  $\mu$  are respectively the variance and the mean. It works well with imbalanced data and stays robust even when there are irrelevant or redundant features.

Due to the mentioned characteristics of this variant we decided to use this one in our project.

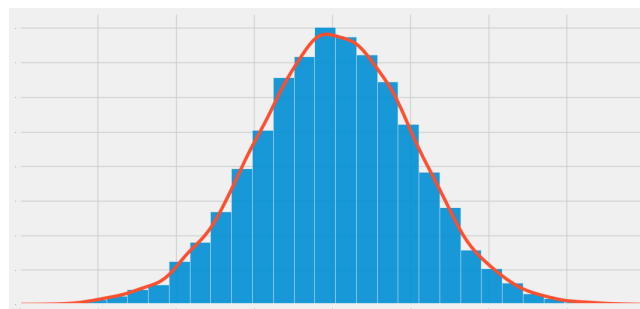


Figure 6: Desired Gaussian Distribution for Gaussian Naive Bayes<sup>2</sup>

---

<sup>2</sup>Source: <https://proclusacademy.com/blog/practical/normal-distribution-python-scipy/>

### 4.3.3 Typical Use Cases

- **Spam Filtering:** Classifies emails as spam or not.
- **Text Classification:** Sentiment analysis, document categorization, and topic classification.
- **Medical Diagnosis:** Helps predicting the likelihood of a disease based on symptoms.
- **Credit Scoring:** Evaluates the creditworthiness of individuals for loan approval.
- **Weather Prediction:** Classifies weather conditions.

### 4.3.4 Advantages

- Easy to implement and efficient.
- Effective in cases with high dimensionality.
- Does not need much data to estimate the necessary parameters.
- Performs well with categorical features

### 4.3.5 Disadvantages

- Propense to be influenced by irrelevant attributes.
- Can not model complex relationships.
- Sensitivity to unbalanced data.
- Does not ignore useless features, treats all features equally.

## 5 Implementation

### 5.1 K-Means Clustering

The K-Means Clustering algorithm was applied on the standardized and feature-selected dataset to identify underlying patterns in the data without using class labels.

To determine the optimal number of clusters  $k$ , values from 2 to 10 were tested. For each value of  $k$ , the model was trained using KMeans from scikit-learn, and two performance metrics were calculated:

- Inertia - the sum of squared distances within clusters,
- Silhouette score - a measure of how well-separated the clusters are.

```
ks = range(2, 11)
inertias, sil_scores = [], []

for k in ks:
    km = KMeans(n_clusters=k, random_state=0, n_init=10, max_iter=300)
    lbl = km.fit_predict(X_scaled_selected)
    inertias.append(km.inertia_)
    sil_scores.append(silhouette_score(X_scaled_selected, lbl))
```

Two plots were generated to assist with cluster selection: Figure 7 shows how inertia decreases with increasing  $k$ , and Figure 8 shows the average silhouette score for each  $k$ .

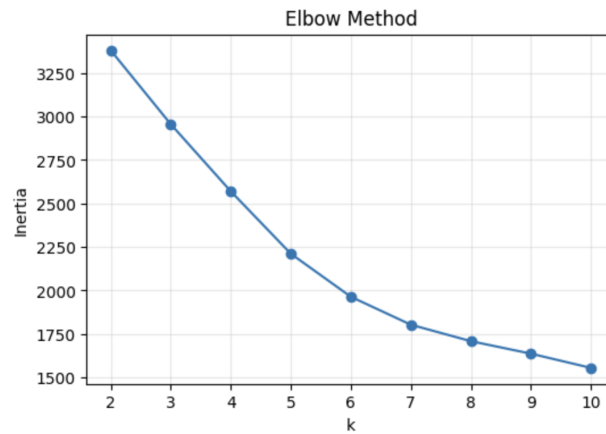


Figure 7: Elbow Method

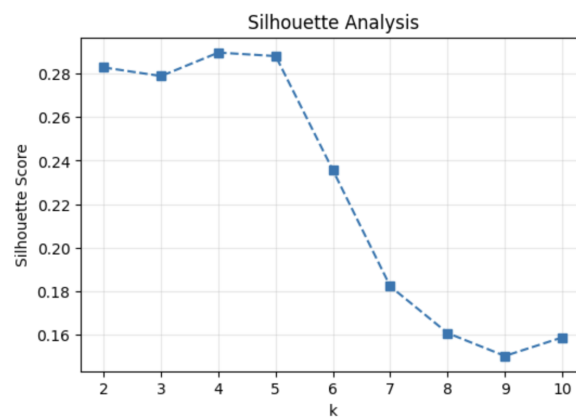


Figure 8: Silhouette Analysis

The silhouette score indicated that k=4 was optimal:

```
best_k = ks[int(np.argmax(sil_scores))]
print(f"Optimal k by silhouette score: {best_k}")
```

Output:

Optimal k by silhouette score: 4

Lastly, a final KMeans model was then trained with k=4:

```
k = 4
km_final = KMeans(n_clusters=k, random_state=0, n_init=10, max_iter=300).fit(X_scaled_selected)
labels = km_final.labels_
X_scaled_selected['cluster_label'] = labels
```

## 5.2 Decision Tree

Implementing the decision tree is simple, we split the data into a train set and a test set, then use the function "DecisionTreeClassifier" from the sklearn.tree library. However it is important to find the best possible parameters and the best split for our DT to be optimal:

### Choosing the best data split:

We will use DecisionTreeClassifier's default parameters with several different split sizes, and analyse which one gives the best results. Here is how we will do so:

- **1st Step:** Make a list of the different test size we want to compare.

```
testsize = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7]
```

- **2nd Step:** Create empty lists that will contain the accuracy, precision, and recall score for each test size.

```
accu_list = []  
prec_list = []  
rec_list = []
```

- **3rd Step:** Using a for loop, create a DT for each test size, then add the accuracy, precision and recall score to the lists.
- **4th Step:** Create a last list, containing the means of those 3 evaluators for each testsize. The maximum value in this list corresponds to the optimal test size for our DT.

Here is what the results look like:

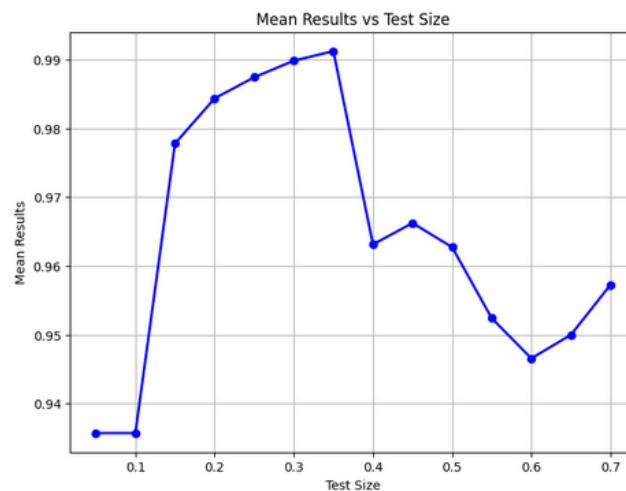


Figure 9: mean results vs test size

### Choosing the best parameters for the `DecisionTreeClassifier` function:

We will focus on 2 parameters, maximum depth of the DT and the ccp alpha parameter (that controls Cost Complexity Pruning).

- **1st Step:** Make a list of the different parameter values we want to compare.

```
max_depth = [2, 3, 4, 5, 6, 7, 8, 9, 10]  
ccp_alpha = [0.001, 0.01, 0.1, 1]
```

- **2nd Step:** Create empty lists that will contain the accuracy, precision, and recall score for each of these.

```
accu_list = []  
prec_list = []  
rec_list = []
```

- **3rd Step:** Using a for loop, create a DT for each parameter, then add the accuracy, precision and recall score to the lists. We start with a fixed ccp alpha and different values for maximum depth, then the other way around
- **4th Step:** Create a last list, containing the means of those 3 evaluators for each parameter value. The maximum values of those two lists corresponds to the optimal parameters for our decision tree.

Here is what those results look like:

We notice that the Maximum depth does not change our results, so we take the smallest one (`maxdepth = 2`) to create a simpler, more interpretable model.

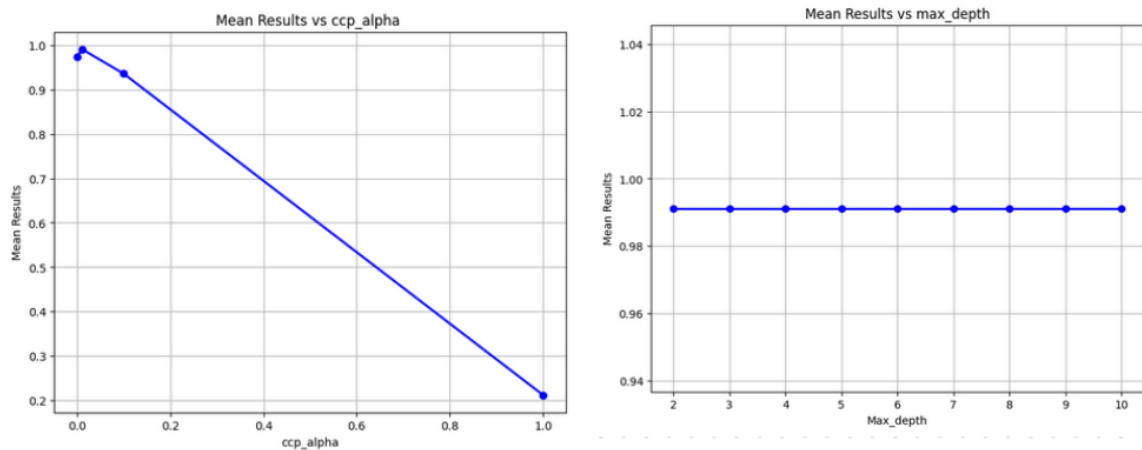


Figure 10: mean results vs ccp alpha/max depth

### Remove the unimportant features for the creation of our DT:

To do so we create a dataframe containing the feature names and their importance using the "clf.featureimportances" parameter:

```
feature_names = X_scaled.columns
feature_importance =
pd.DataFrame(clf.feature_importances_, index = feature_names).sort_values(0, ascending=False)
feature_importance
```

We then remove the features that are not important in our DT creation using the "drop" function. These are the only 3 features we keep:

<b>hemo</b>	0.749777
<b>sg</b>	0.226089
<b>sc</b>	0.024134
<b>age</b>	0.000000

Figure 11: feature importance

### Creation of our final DT:

We use the chosen dataset, test split, and parameters, create our decision tree, calculate it's accuracy, precision and recall score, display the tree and the corresponding confusion matrix.

## 5.3 Gaussian Naive Bayes

This selected variant of the Naive Bayes algorithm, which despite having not much room for tuning through parameters, is an interesting and common used model in the real world, specially for medical data.

As already mentioned, it is intended to have a Gaussian distribution for each feature in the data set so that the model can achieve a good performance. Since the data has been standardize, its assured that the variances are favorable for our model.

However, it is always worthwhile to take a look at the data and plotting important information about it.

- **QQ-Plot:** It is used to compare the distribution of our data to a theoretical Gaussian Distribution. If the data follows this normal distribution, then the points will align to the theoretical quantile line.
- **Distribution Plot with Gaussian Function:** This demonstrates how the data is distributed and a visualization of the bell curve.

- **Shapiro-Wilk test:** This test presents how close our data is of being in the desired distribution. A W-statistic value equally to one is a perfect data spread.

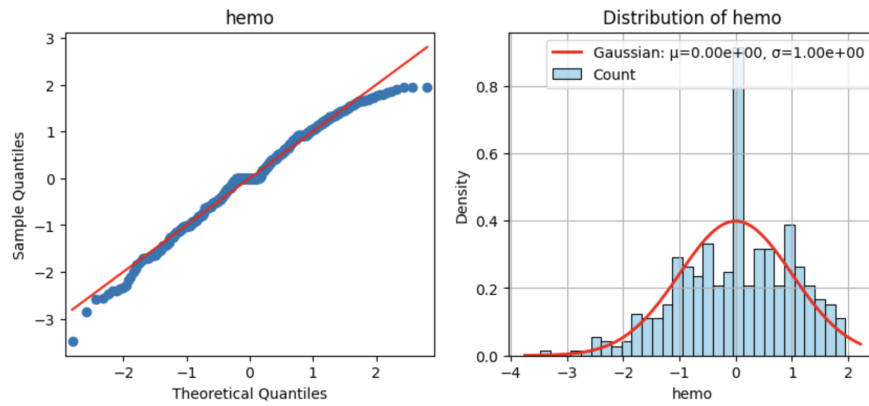


Figure 12: QQ-Plot (left) and Distribution Plot (Right) with good values

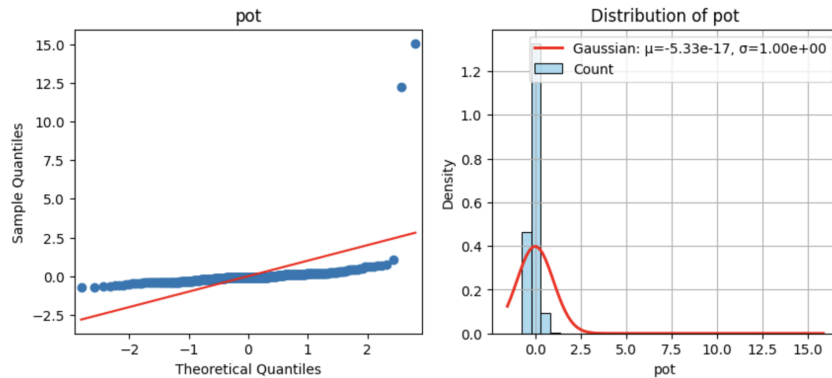


Figure 13: QQ-Plot (left) and Distribution Plot (Right) with questionable values

These two features have the following variances and W-statistic. One can see that, in accordance to the plots, the *pot* and the *hemo* features show respectively low and high W values.

	W-statistic	variances
age	0.964753	1.002506
bp	0.867299	1.002506
sg	0.905992	1.002506
al	0.775108	1.002506
su	0.433899	1.002506
bgr	0.741094	1.002506
bu	0.712249	1.002506
sc	0.389889	1.002506
sod	0.566884	1.002506
pot	0.182373	1.002506
hemo	0.984219	1.002506
pcv	0.972725	1.002506

Firstly, a baseline model with a partition of 80% for train data and 20% for test data was executed. No parameters were used for the classifier. This was followed by a new classification with *prior* parameters. These parameters define the likelihood for each class before seeing any features.

```
...
gaussian_NBClassifier = GaussianNB(priors=[0.86, 0.14])
...
```

Finally, cross validation was carried out. The goal of this was in the first place to evaluate how well the machine generalizes to unseen data and avoid overfitting or underfitting, and to test the impact of the test sizes on the model.

```

folds = [2, 3, 4, 5, 6, 7, 8, 9]
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
scores = {}

for fold in folds:
    scores[fold] = cross_validate(GaussianNB(),
                                  X_scaled,
                                  y,
                                  cv=fold,
                                  scoring=scoring,
                                  return_train_score=True)

```

## 6 Analysis of the Results

### 6.1 K-Means Clustering Results

After applying K-Means clustering with the optimal number of clusters ( $k = 4$ ), we evaluated the quality of the clustering by comparing the predicted cluster assignments to the true class labels. Although K-Means is an unsupervised learning algorithm, the availability of ground truth labels allowed for external validation using two key metrics:

- **Adjusted Rand Index (ARI):** Measures the agreement between predicted clusters and true labels, adjusted for chance. Values closer to 1 indicate stronger correspondence.
- **Purity:** Measures the extent to which clusters contain members from a single class. It is computed by summing the maximum class frequencies within each cluster and dividing by the total number of instances.

**Evaluation Results for  $k = 4$ :**

```

Results for k=4:
Adjusted Rand Index: 0.280
Purity:              0.830

```

These results suggest that while the clusters are not perfectly aligned with the true class distribution (as indicated by the relatively low ARI), there is a moderately high level of purity. This means most clusters are dominated by samples from a single class, particularly for the CKD class.

**Cluster Distribution and Composition:**

```

Cluster sizes:
0    218
1    135
2     45
3      2
Name: count, dtype: int64

Cluster vs True Class:
class    0    1
row_0
0         68  150
1        135    0
2         45    0
3          2    0

```

Cluster 0 predominantly contains samples from class 1 (CKD), while clusters 1, 2, and 3 consist entirely of class 0 samples (non-CKD). This indicates that only one cluster effectively captures the CKD population, whereas the remaining clusters segment the non-CKD group into distinct subclusters.

Figure 14 visualizes this relationship through a heatmap comparing predicted cluster labels to actual class labels. Darker shades indicate higher counts. The figure shows that despite operating without supervision, K-Means was able to isolate the majority of CKD cases into a single cluster, revealing meaningful structure in the data.



Figure 14: Cluster vs True Class Heatmap

## 6.2 Decision Tree Results

To analyse the efficiency of our decision tree algorithm, we will use three evaluation metrics:

- **accuracy**, which measures the overall correctness of your decision tree - what percentage of all predictions were correct.
- **Precision**, which measures how reliable your positive predictions are - when your decision tree predicts class 1, how often is it actually correct?
- **Recall**, which measures how well your decision tree finds all the actual positive cases - of all the true class 1 instances, how many did your tree correctly identify?

For a test size of 0.35, a maximum tree depth of 2 and a ccp.alpha of 0.01, which we determined to be the best possible parameters for our decision tree, and while only keeping hemoglobin, specific gravity and serum creatinine as features to avoid overfitting, we get our final decision tree:



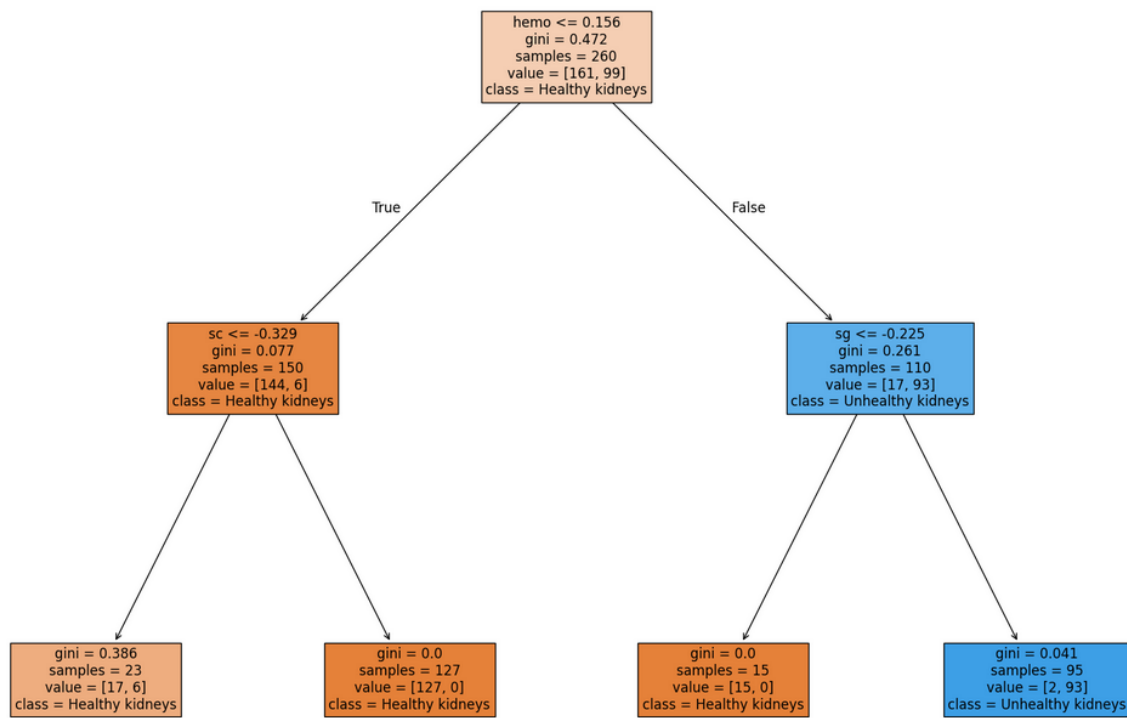


Figure 15: Decision Tree

And here are the resulting evaluation parameters:

Accuracy: 0.9928571428571429  
 Precision: 0.9807692307692307  
 Recall: 1.0

Those values are more than acceptable:

- our algorithm will make the right call in 99percent of cases.
- The precision is also very close to 1, which means that when a kidney disease was predicted, it was almost always present.
- Finally recall is equal to 1, and recall is extremely important in medical data as the worst possible case is to diagnose a patient as healthy when he is not.

Let's display our confusion matrix before concluding about the efficiency of our decision tree:

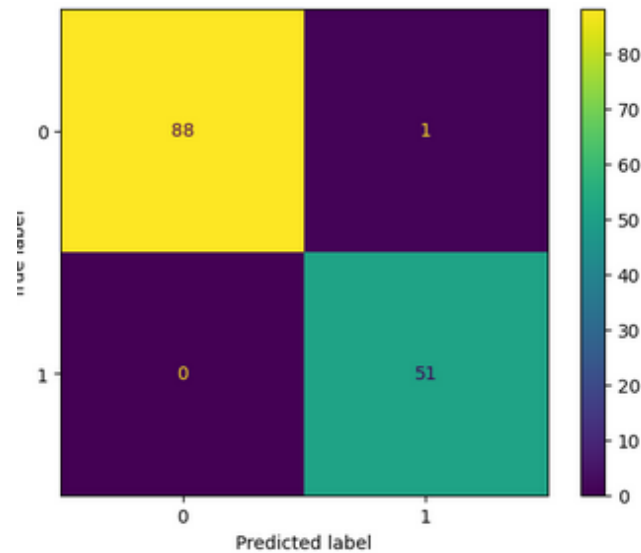


Figure 16: Decision Tree confusion matrix

As expected, the confusion matrix shows us that the results are perfectly acceptable: only one out of the 140 patients in the test bracket was misdiagnosed; furthermore this patient is a false positive, which is way less worrying than a false negative.

### Cross validation

In order to check if our tree is memorizing training data rather than learning generalizable patterns, we will use cross validation. To do so we split the data in k equal parts, train on k-1 parts and test on others, repeating the operation k times to get the full results.

After doing so with our cleaned dataset, these are the results we get:

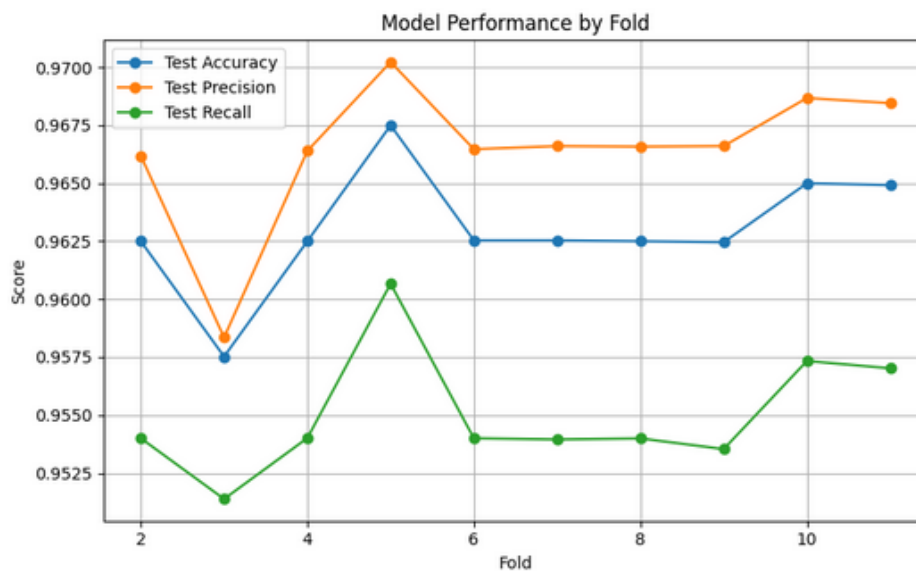


Figure 17: Cross validation results

We notice that all three evaluators stay above 0.95, which confirm the suitability of the decision tree for our classification.

**Conclusion:** The decision tree algorithm seems to perfectly suit our chronic kidney disease classification.

### 6.3 Gaussian Naive Bayes Results

In addition to the evaluation metrics described in section 6.2, following performance metrics were applied:

- **F1 Score**, which balances precision and recall into a single value. It is useful in situations where false positives and false negatives carry significant consequences, like in our case.
- **Area Under the Curve** represents the model's ability to distinguish between classes across all possible class labels, that is to say, an evaluation of random guessing.

The baseline model with a data partition of 4 : 1 let to the following evaluation metrics:

Accuracy: 0.9375  
Precision: 0.8484848484848485  
Recall: 1.0  
F1 Score: 0.9180327868852459  
AUC Score: 0.9945054945054945

It is remarkable that the precision is below 85%, which exhibits that the model was not able to correctly identify all false cases. On the other hand, it was capable of correctly identifying all positive cases, which in our medical case is highly relevant. A look at the confusion matrix reveals us these details better.

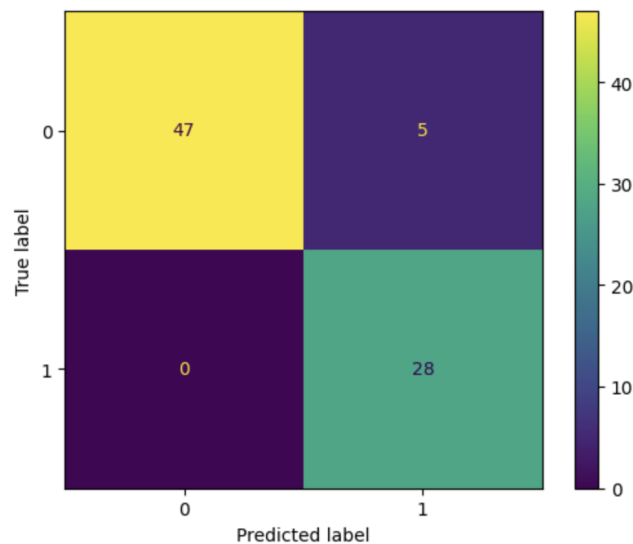


Figure 18: Naive Bayes Confusion Matrix

Since no prior parameters were given, this first model calculated the priors for each class [class 0 class 1]:

Class Priors: [0.61875 0.38125]

This leads to the second alternative with given prior. These values (0.86, 0.14) are based on the global statistic about people having chronic kidney diseases. By passing the parameters the precision was increased by 0.03 points and the false positives decreased by 1.

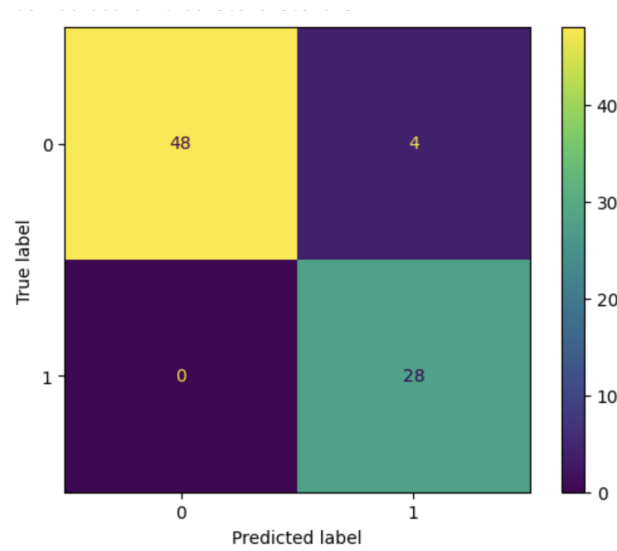


Figure 19: Naive Bayes Confusion Matrix with prior parameters

Lastly, with the cross validation, different test sizes were checked. Because the number of folds in cross validation determines how the data is split, the defined folds from 2 to 11 show which test split is the most appropriate for the algorithm and additionally, how good the model generalizes, testing different arranged data.

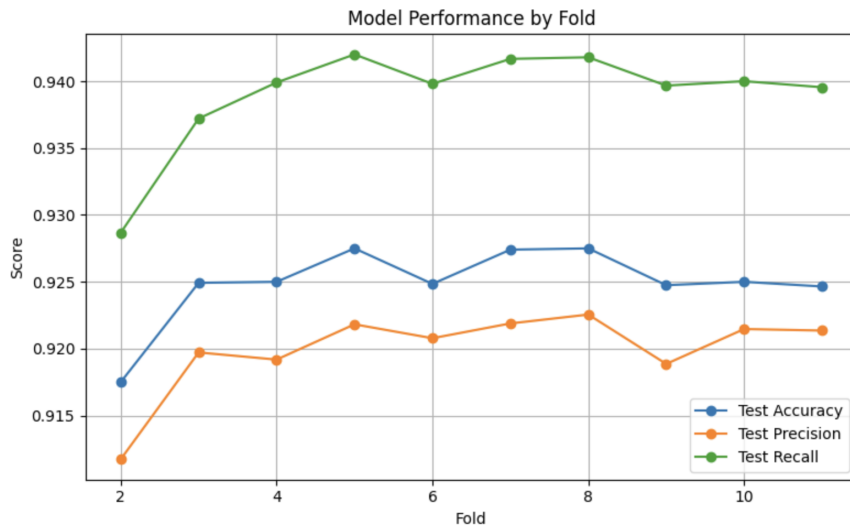


Figure 20: Mean metrics for each fold

As the graph shows, the fold with the best performance was 8, meaning that a test split of 12.5% represents the most suitable for this model. Although having increased the precision compared to the baseline and the prior models, a reduction of the recall is notable. This could be explained by the model becoming more conservative in its predictions, favoring fewer positive classifications to increase accuracy, which in turn leads to more false negatives. As a result, while the model is better at correctly identifying true positives (high precision), it may be failing to capture all relevant cases (lower recall), which can be critical in the case of our application.

## 7 Conclusions

This project aimed to develop the most effective machine learning algorithm for predicting chronic kidney disease (CKD) using data collected by a hospital o a two month span. We implemented and compared three distinct approaches: Naive Bayes classification, K-means clustering, and Decision Tree classification. Each algorithm brings unique strengths and limitations to the medical diagnosis domain, and our comprehensive evaluation provides valuable insights into their applicability for CKD prediction.

- **K-Means Clustering:** The level of purity is moderately high, and even though the clusters are not perfect, most of the predictions are true. The model is effective in identifying natural groupings in patient data and can identify subtypes of patients with CKD with similar characteristics.

Nevertheless, the results remain too poor for practical use in the medical field, where we expect the algorithm to have a close to perfect evaluation of the patient's state. Furthermore it requires a predefined number of clusters that do not match the natural disease patterns.

- **Decision Tree:** All accuracy, precision and recall values obtained are perfectly acceptable for medical classification. The algorithm's effectiveness when it comes to identifying the most important features is also one of its strengths, as we can use this to know what to look for and what characteristics to check on future patients. We can also add that the cross-validation results are on point, as the model's performance only changes by around 1% maximum.
- **Gaussian Naive Bayes:** The accuracy level we get from the algorithm is acceptable, and the recall score (which is in the case of medical data a capital value) is equal to 1 which is an outstanding result. We can also note that it gives the fastest training and predictions out of our three methods.

The main problem with the algorithm is the precision value, which does not get higher than 85%, a score too low to be used in the medical domain, as we must choose between getting more false positives or more false negatives. This method also assumes feature independence, not always guaranteed in medical data.

To conclude, we found out that our best option is to use the decision tree algorithm to detect chronic kidney disease as it provides the best balance of accuracy and interpretability, and applicability.

This project made us learn how to properly deal with missing data, the importance of preprocessing, data-scaling, and cross-validation but also that optimization is a crucial aspect of machine learning.

We would like to thank both our professors, Joanna Kołodziej and Adrian Widłak for all the lessons, help, and advice they gave us along this semester, and we hope that this project responds to all of your criteria.