NAME: RAPHAEL LAWRENCE FARODOYE
COURSE: COM804

EXPLORATORY DATA ANALYSIS (EDA) & PREPROCESSING

Dataset
This study uses a Chronic Kidney Disease (CKD) dataset to preridct disease progression and stage. The dataset contains two target variables: **CKD_Progression,** a binary indicator of whether a disease is advancing and **CKD_Stage** a multiclass variable representing the current disease stage (2 - 5). The primary objective is to apply Case-Based Reasoninng (CBR), a lazy learner AI technique that predicts outcomes for new cases based on its similarity to previously solved past cases.
The dataset comprises 1138 patient cases and 23 features.

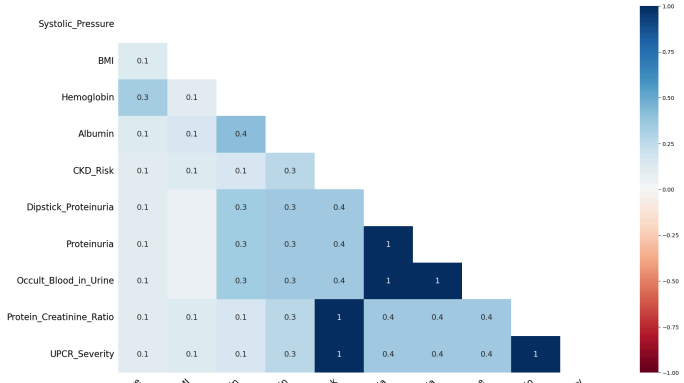
---

1. MISSING DATA

I quantified the missingness per feature and visualized patterns using missingno (matrix + heatmap).
I calculated the missing values and the percentage of missing values in the dataset.

- **BMI** has the highest missing value of 137 missing rows (12.04% of the entire dataset)
- **Protein_Creatinine_Ratio, CKD_Risk, UPCR_Severity** each have 88 missing rows (7.7%)
- **Systolic_Pressure, Dipstick_Proteinuria, Proteinuria** each have 16 missing rows (1.41%)

To understand the pattern of missingness, missingno matrix and heatmap visualizations were generated. In the matrix, it indicates that where there are values of 1, there are missing values on the same row, while the heatmap indicated a strong correlation in missingness among *Systolic_Pressure, Hemoglobin, Albumin, CKD_Risk, Dipstick_Poteinuria, Proteinuria, Occult_Blood_in_Urine, Protein_Creatinine_Ratio, CKD_Risk,* and *UPCR_Severity.* I observed many features share missing rows for the same patient records. This pattern suggests the data is **Not Missing at Random (NMAR).** This is likely due to incomplete medical records; imputing these would likely inject **bias.**



2. HANDLING MISSING DATA

- **Row Deletion:** Since the nature of the missing columns are not random, I dropped the 88 null rows in *CKD_Risk, Protein_Creatinine_Ratio, UPCR_Severity.* After removing those rows, it was realized that this action also resolved the missingness in other correlated features. The missingness value in **Systolic_Pressure, Dipstick_Proteinuria, Proteinuria** dropped from 16 to 2. This confirmed that most of the missingness were for the same patients (incomplete records).
- **Continuous Features Imputation:** I plotted a skewplot (fig:1) and its value for the continuous features to understand the rate of skewness
    - Systolic_Pressure: 0.52 (slightly right-skewed)
    - BMI: 0.82 (Moderately right-skewed)
    - Albumin: 0.95 (Moderately left-skewed)

  Since the three feature distributions were skewed (skew > 0.5), the median was chosen for imputation. The median is more robust and less sensitive to outliers than mean.

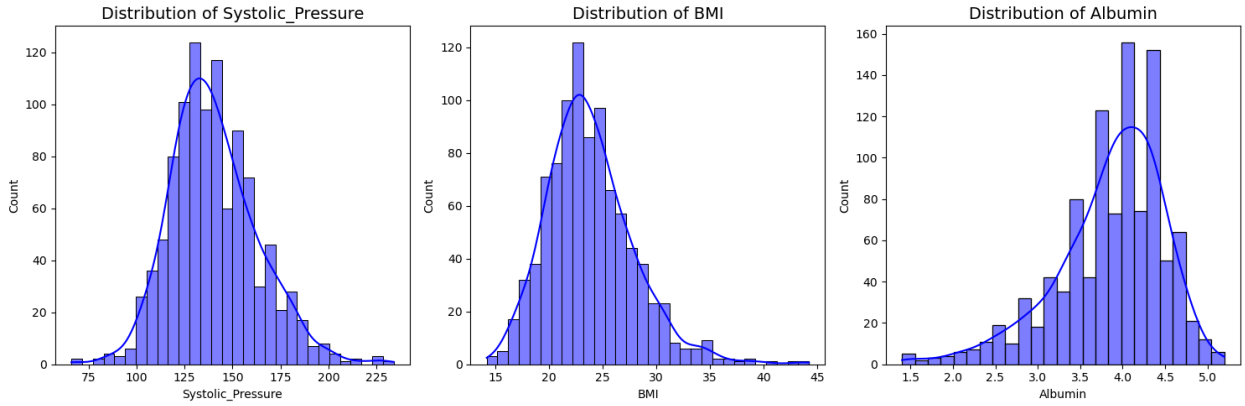


fig: 1

- Imputation of Categorical Features: The missing values in the categorical features (**Systolic_Pressure, Dispstick_Proteinuria, Protenuria)** were handled using a class-conditional imputation method. Instead of using a global mode, the mode was calculated within each **CKD_Stage** class. This approach is more precise than global imputation, as it preserves the data structure and reduces the risk of introducing noise and reducing bias. Each missing value was replaced with the most frequent value of its corresponding **CKD_Stage.**

CKD_Stage	CKD_Risk	Dipstick_Proteinuria	Proteinuria	Occult_Blood_in_Urine
4	9.0	NaN	NaN	NaN
3	6.0	NaN	NaN	NaN

Missing rows

CKD_Stage	CKD_Risk	Dipstick_Proteinuria	Proteinuria	Occult_Blood_in_Urine
4	9.0	3.0	1.0	0.0
3	6.0	2.0	1.0	1.0

imputed categorical rows

3. CORRELATION

A correlation analysis was performed where we plotted a correlation heatmap *(Fig: 3)* to understand the relationship of the features with each other.

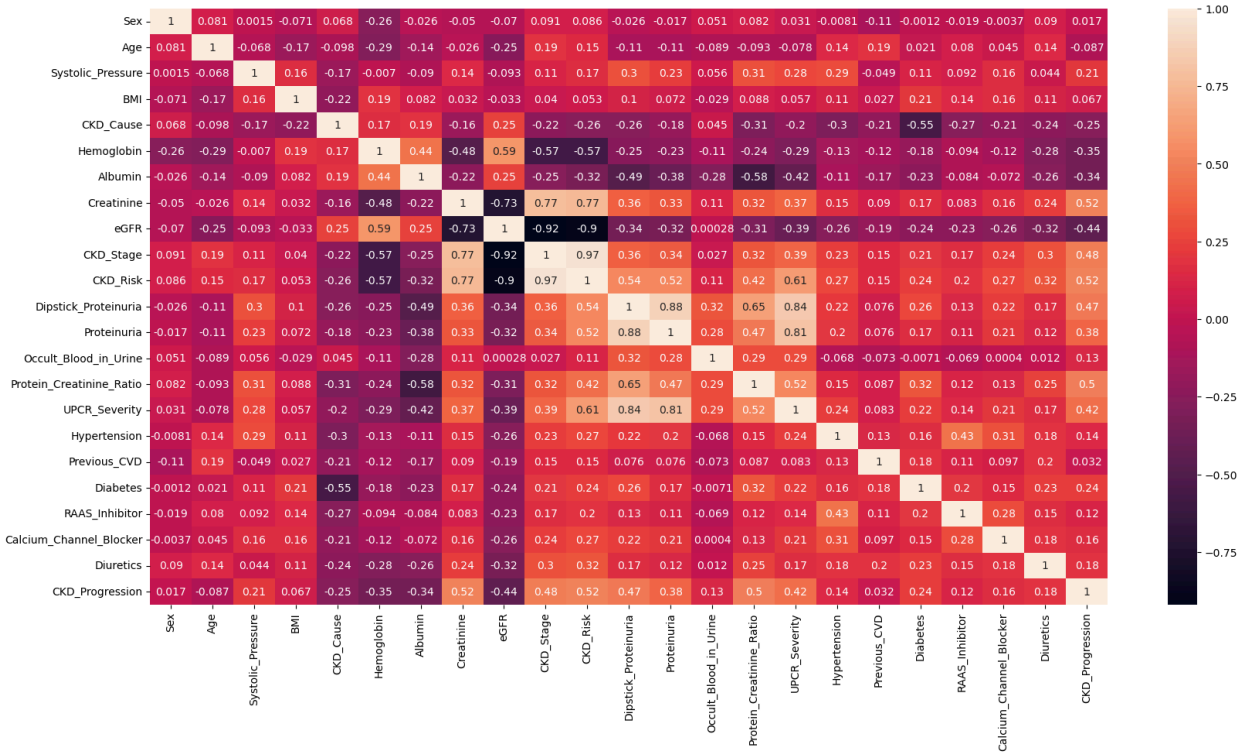


Fig: 3

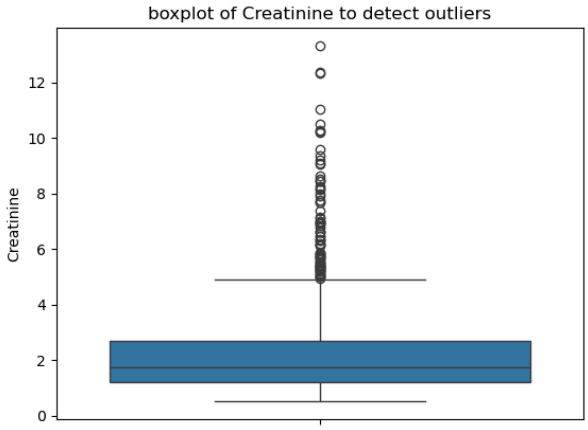
- With the CKD_stage(Multi-class Target):
    - ***CKD_Risk:*** has a strong positive correlation(0.97)
    - ***eGFR***: Strong negativer correlation (-0.92)
    - ***Creatinine***: Strong positive correlation (0.77)
    - ***Occult_Blood_in_Urine:*** Weak corelation (0.02)
- With **CKD_Progression (Binary Target):**
  No features has strong correlation with the binary target.

3. OUTLIER DETECTION AND REMOVAL

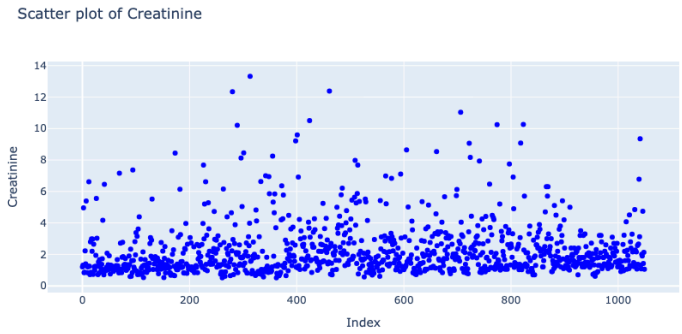
In the descriptive statistics summary *(Fig: 4)* of the dataset, it could be observed that **Creatinine** has some outliers. There is a huge gap between the 75th percentile (2.68) and the maximum value (13.32). The scatter plot and boxplot visually confirmed the outliers.

	Sex	Age	Systolic_Pressure	BMI	CKD_Cause	Hemoglobin	Albumin	Creatinine	eGFR	CKD_Stage
count	1050.000000	1050.000000	1050.000000	1050.000000	1050.000000	1050.000000	1050.000000	1050.000000	1050.000000	1050.000000
mean	1.300000	67.315238	140.239048	23.758095	2.248571	11.962476	3.854000	2.260276	32.913067	3.598095
std	0.458476	13.571920	22.678568	3.834425	1.010961	2.281517	0.632341	1.712716	18.795473	0.881110
min	1.000000	22.000000	66.000000	14.200000	1.000000	5.900000	1.400000	0.510000	2.470000	2.000000
25%	1.000000	61.000000	125.000000	21.400000	1.000000	10.200000	3.500000	1.200000	17.570000	3.000000
50%	1.000000	70.000000	138.000000	23.300000	2.000000	12.000000	4.000000	1.740000	29.885000	4.000000
75%	2.000000	77.000000	153.000000	25.600000	3.000000	13.600000	4.300000	2.680000	45.867500	4.000000
max	2.000000	94.000000	234.000000	44.200000	4.000000	18.500000	5.200000	13.320000	89.980000	5.000000

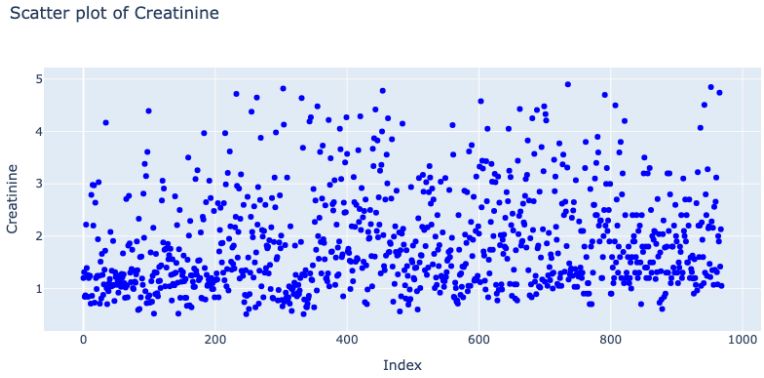
Descriptive summary



Boxplot



scatterplot with outliers



Scatterplot without outliers

Fig: 4

The outliers were flagged and filtered from the dataset using the interquartile range (IQR) method. Data points falling outside the range defined by 1.5 * IQR below the first quartile or above the third quartile were removed. Which could dominate the distance computations in **CBR**.

4. DATA VISUALIZATION

Plotted the pie chart of the binary classification with the Muliticlass classification

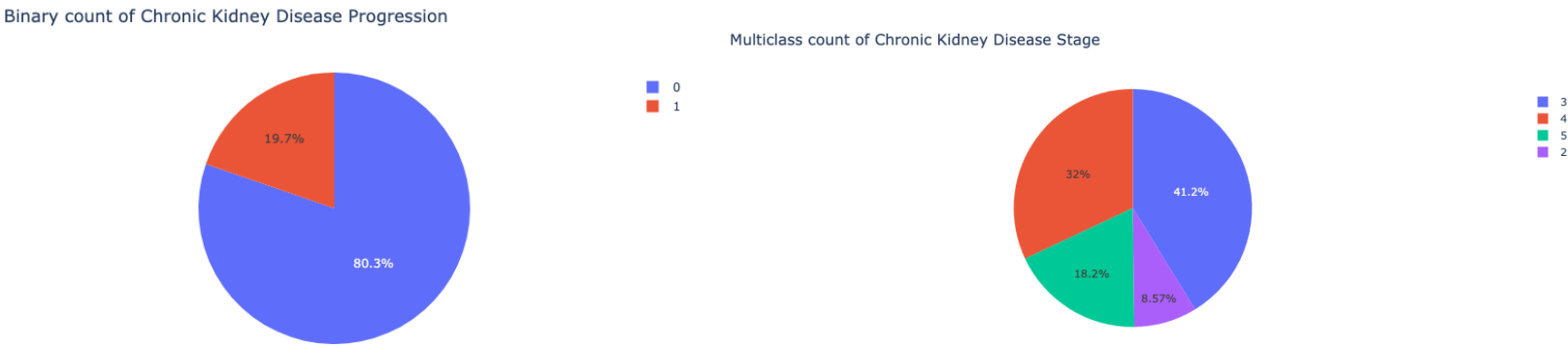


Fig: 5.1

- **The Binary Pie** chart in Fig: 5.1 CKD_Progression revealed a class imbalance with **80.3%** of cases labeled as 0 : “Not Progressing” and **19.7%** as ‘Progressing.’

- **The Multiclass Pie** chart lets us understand the distrubution of the Chronic Kidney Disease Stage. The distribution are: Stage 3 (44.7%), Stage 4 (34.7%), Stage 5 (11.4%), and Stage 2 (9.2%).

- A large number of patients in the dataset are diagnosed with Stage 3 Chronic Kidney Disease which could be that most of the patients seek medical care only after the diease has progressed rather than earlier stages.

  In the correlation matrix (Fig: 3), we discovered there is a strong negative correlation between Creatinine and eGFR.

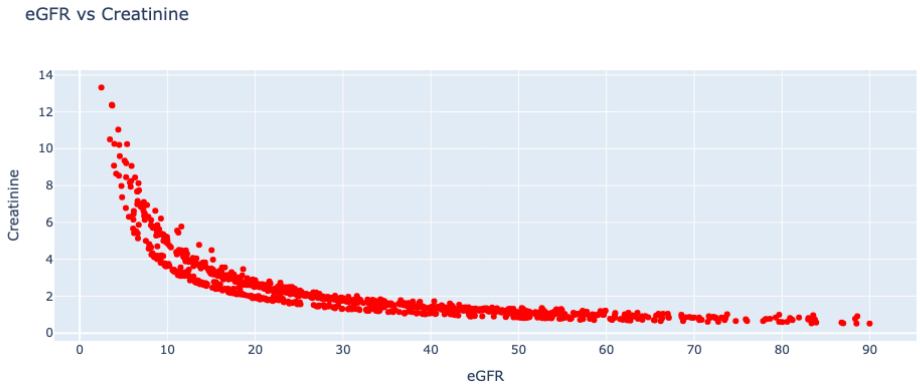


Fig: 5.2
The scatterplot lets us understand that as the value of eGFR increases, Creatinine decreases

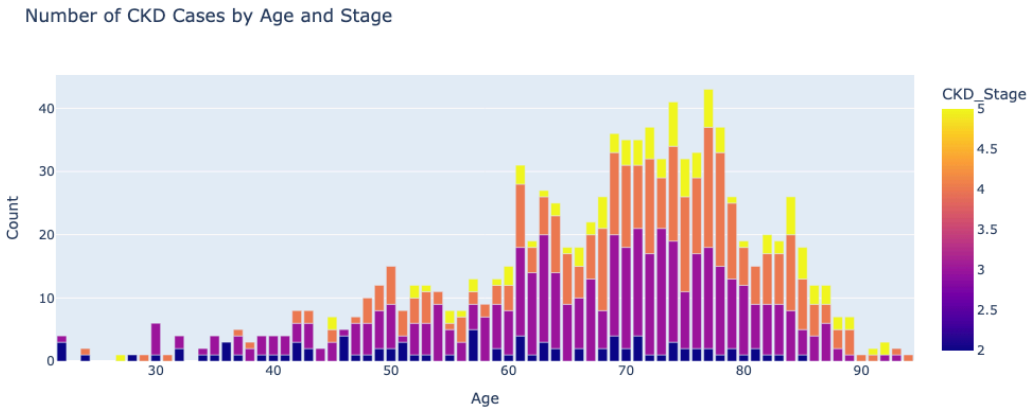


Fig: 5.3

The interactive bar chart in Fig: 5.3 allows us to analyze the distribution of CKD stages across different age groups.
(2.g. At age 77, there are 6 individuals in stage 5, 19 in stage 4, 16 in stage 3 and 2 in stage 2.

Preprocessing:

After the clean data, I checked for correlation of other features with the Binary label, all labels are weakly correlated with the binary label and was stored as *Binary_data*.
While for the Multiclass label, **CKD_Risk** and **eGFR** were identified to have a strong correlation of above 90% which is above the threshold and was removed and stored as *Multi_data.*

The dataset was divided into a training set (70%) and a test set (30%). Stratified sampling was used to preserve the class distribution from the original dataset in both subsets

Case-Based Reasoning (CBR) System Implementation

To address the prediction task, a custom Case-Based Reasoning (CBR) system was implemented in Python. The system is designed around the four core processes of the CBR cycle: **Retrieve, Reuse, Revise, and Retain**. The implementation is encapsulated in a CBR class that manages the case base, computes similarity, derives solutions, and optimizes its own performance through feature weight tuning.

1. Case Representation and Initialization

The system is initialized with predefined lists of **continuous** and **categorical** features, target class names, and an optional dictionary of feature weights. Each "case" in the case base is a structured object containing a unique ID, a vector of feature values, and the corresponding solution (i.e., the target class). The create_case_base method stores the training dataset (X, y) and calculating necessary statistics (min, max, mean, std) for each continuous feature, which are stored for later use in similarity calculations. The dataset was divided into a training set (70%) and a test set (30%). Stratified sampling was used to preserve the class distribution from the original dataset in both subsets

---

2. The Retrieve Phase: Similarity Calculation

The retrieval of similar cases is defined in retrieve_similar_cases, which identifies the **top-k most similar cases** from the case base for a new query case. This process uses a weighted similarity metric computed by the local_similarity function.

The overall similarity between two cases is calculated as the **weighted sum of the local similarities** for each feature:

$$\text{GlobalSimilarity}(C_{query}, C_{case}) = \frac{\sum_{i=1}^{n} w_i \times \text{sim}(f_{i,query}, f_{i,case})}{\sum_{i=1}^{n} w_i}$$

W_i is the weight of feature i, where we used 1.0 as the initial weight and $sim(f_i, f_j)$ is the local similarity for that feature.

- **For categorical features**, a simple identity metric is used: similarity is **1** if the values are identical and **0** otherwise.
- **For continuous features**, the similarity is calculated based on the normalized distance. The implementation supports two normalization methods:
    - **Min-Max Normalization (default):** The similarity is calculated as $1 - \frac{|v_1 - v_2|}{\text{range}}$

      where the range is the difference between the maximum and minimum values of that feature in the case base.

---

3. The Reuse Phase: Solution Generation

Once the top-k similar cases are retrieved, the reuse_solution method is employed to generate a prediction for the query case. This function aggregates the solutions from the retrieved neighbors. Several voting strategies are implemented to ensure flexibility:

- **Majority Voting:** The most frequent solution among the neighbors is chosen.
- **Weighted Voting:** Each neighbor's vote is weighted by its similarity score, giving more influence to more similar cases.
- **Distance-Weighted Voting:** Similar to weighted voting, but the weights are squared

---

4. The Revise and Retain Phase: Feature Weight Optimization

A key feature of this CBR system is its ability to learn and adapt, which corresponds to the **Revise** and **Retain** stages of the CBR cycle. This is implemented in the optimize_weights_gradient_descent method.

This function automatically tunes the feature weights (w_i) to improve predictive accuracy. It uses a **gradient descent** algorithm on a validation dataset. For each case in the validation set, the algorithm:

1. **Predicts** an outcome using the current feature weights.
2. **Calculates the error** by comparing the similarity score of retrieved neighbors to an "ideal" similarity (1 for neighbors with the correct solution, 0 otherwise).
3. **Computes the gradient** of this error with respect to each feature's weight.
4. **Updates the weights** in the direction that minimizes the error.

This process is repeated for a set number of epochs, effectively allowing the model to **learn the importance of each feature** from data. The weights that yield the highest accuracy on the validation set are retained, representing the learned knowledge of the system. This automated optimization makes the model more adaptive and accurate compared to one with manually

assigned weights.

CODE EXPLANATION

Here's a more direct translation of what each major function in the code does.

- __init__(...): The **constructor**. It sets up the CBR system's configuration. It needs to know which features are continuous vs. categorical to apply the correct similarity logic. It sets the weight for all features as 1 as default.
- create_case_base(...): **Creates a casebase**. It takes the training data and converts each row into a "case" dictionary. It calls _compute_feature_ranges to pre-calculate the min, max, mean, and standard deviation for all continuous features.
- local_similarity(...): The **core similarity engine**. This function compares two cases, one feature at a time.
    - It checks if a feature is continuous or categorical.
    - If **categorical**, it scores it as 1 if they match, 0 if they don't.
    - If **continuous**, it calculates the distance between the values and normalizes it to a 0-1 range, then subtracts from 1 to turn distance into similarity (e.g., zero distance = 1 similarity).
    - It multiplies each feature's similarity score by its learned **weight** and calculates the final weighted average.
- retrieve_similar_cases(...): It takes a new query case, compares it against *every single case* in its memory using local_similarity, sorts them from most to least similar, and returns the top k cases. Which is similar to how k-Nearest Neighbors (k-NN) works.
- reuse_solution(...): It takes the list of top-k similar cases from the previous step and makes a final prediction. It does this by using "majority rules" vote where the opinion of a highly similar case matters more than a less similar one.
- optimize_weights_gradient_descent(...): This function learns the weights automatically and discovers more important features.
    - It iterates through a validation dataset many times (epochs).
    - In each iteration, it tries to predict the outcome for a case.
    - It then looks at the neighbors it used for the prediction. If a neighbor had the *wrong*
    - It calculates the "error" based on this and uses calculus (gradient descent) to slightly adjust the feature weights. If a feature consistently contributes to bad predictions, its weight is lowered. If it's crucial for good predictions, its weight is increased.
    - It then saves the best set of weights it found during this process.

RESULT

Binary (Progression).

I trained the model for 300 epochs to learn the best feature weights. The training process showed that the model achieved a high accuracy of 81%. After the optimization was complete, the final learned weights for the features were all very close to 1.0. This means the model concluded that all the features were equally important for predicting whether the disease was progressing.

Performance on Test Data

I evaluated the model on the test data to see how well it performed on new cases.

- **Accuracy:** The model achieved an accuracy of 84%, which means it correctly predicted the outcome for 84% of the patients in the test set
- **ROC-AUC-SCORE**: The model achieved an ROC-AUC of 0.87. It showed the modelis excellent at differentiating between a progressing and non-progressing case.

Classification Report

	Predicted: Class 0	**Predicted: Class 1**
Actual: Class 0	**215**	**19**
Actual: Class 1	**27**	**30**

Class 0 (Non Progressing cases): The model correctly identified 215 out of 234 non progressing cases (92% recall)

Class 1 (Progressing cases): The model correctly identified 30 out of 57 progressing casses (53% recall). This performance is because of the class imbalance and low proportion of Class 1.

CKD_STAGE (Multiclass Classification)

Model Training

Predicting the exact stage was complicated and the training results showed that:

- Over the 300 epochs, the model's accuracy steadily improved from 57% to 70%.
- At the same time, the loss (a measure of error) consistently decreased.

The model was learning and getting better over time.

The model assigned different weights ot the features and it learned some features were much more important than others and it decreasedthe weight of other features that are less important to the target.

The most important features are;

1. **Sex** (Weight: 4.34)
2. **Occult_Blood_in_Urine** (Weight: 3.00)
3. **BMI** (Weight: 1.44)
4. **Creatinine** (Weight: 1.43)
5. **Hemoglobin** (1.22)

Classification Report

	Predicted: Stage 2	**Predicted: Stage 3**	**Predicted: Stage 4**	**Predicted: Stage 5**
Actual: Stage 2	**3**	**23**	**1**	**0**
Actual Stage 3	**0**	**114**	**15**	**1**
Actual Stage 4	**0**	**32**	**65**	**4**
Actual Stage 5	**0**	**3**	**23**	**7**

Stage 2: The model predicted 3 correctly and predicted 23 patiewnts as bing in stage 3

Stage3: The model correctly identified 114 Stage 3 patients, it wrongly classified 15 of them as Stage 4.

Stage 4: The model correctly classified 65 patients correctly and classified 3 patients to belong to Stage 3

Stage 5: The model correctly classified 7 patients as stage 5 and mistakenly labeled 23 Stage 5 patients as being in Stage 4.