

STATS 3DA3

Assignment 6

Woojoo Kim (400296574), Anu Kumar (400304424), Derek Kusmenko (400313439)

2024-04-15

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import array
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

1 Classification Problem Identification

Classification Problem: Our goal is to employ machine learning (ML) methods to forecast, diagnose, and manage chronic kidney disease (CKD). We are given a wide range of variables in the data including age and whether the patient has certain conditions like anemia or coronary artery disease. We will build models that use these variables to best predict whether a given patient will have chronic kidney disease. We are given the target variable 'class' which specifies whether the patient has chronic kidney disease or not.

2 Variable Transformation

Variable Transformation: By looking out the output, we can see that the hypertension, diabetes mellitus, coronary artery disease, appetite, pedal edema, anemia are all supposed to be of type binary. In the dataset however, the input for these columns are written as yes or no, good or poor. We need to convert these into binary types. Furthermore, we also see that red blood cells, pus cell, pus cell clumps and bacteria have the same problem, as well as the Target, which should definitely be a binary number. We need to convert these string type variables into binary variables so that the machine learning algorithms can better understand them, and use the data.

```
import pandas as pd
from ucimlrepo import fetch_ucirepo

# Fetch dataset
```

```

chronic_kidney_disease = fetch_ucirepo(id=336)

# Create pandas DataFrame for features (X)
X = pd.DataFrame(data=chronic_kidney_disease.data.features,
                  columns=chronic_kidney_disease.feature_names)

# Create pandas DataFrame for targets (y)
y = pd.DataFrame(data=chronic_kidney_disease.data.targets, columns=["class"])

# Combining features and targets into a single DataFrame
data = pd.concat([X, y], axis=1)

# Clean column names for categorical variable "dm" (diabetes mellitus)
data['dm'] = data['dm'].replace(['\tno', 'no'], 'no')
data['dm'] = data['dm'].replace(['yes'], 'yes')

# Clean column for categorical variables "rbc"
# Replace missing values with the mode of the column (mode imputation).
# This allows for easy categorical to binary conversion.
data['rbc'].fillna("normal", inplace=True)

# Clean column names for target variable "class"
data['class'] = data['class'].replace(['ckd\t', 'notckd'], 'notckd')
data['class_ckd'] = (data['class'] == 'ckd').astype(int)
# Create binary column indicating presence of CKD

# Drop original "class" column
data.drop('class', axis=1, inplace=True)

# Create binary columns for each remaining categorical variable
data_with_dummies = pd.get_dummies(data,

```

```

        columns=["rbc", "pc", "pcc", "ba",
                  "htn", "dm", "cad", "appet",
                  "pe", "ane"],
        drop_first=True, dtype=int)

print(data_with_dummies.dtypes)

```

```

age                float64
bp                 float64
sg                 float64
al                 float64
su                 float64
bgr                float64
bu                 float64
sc                 float64
sod                float64
pot                float64
hemo               float64
pcv                float64
wbcc               float64
rbcc               float64
class_ckd          int32
rbc_normal          int32
pc_normal           int32
pcc_present         int32
ba_present          int32
htn_yes             int32
dm_yes              int32
cad_yes             int32
appet_poor          int32
pe_yes              int32
ane_yes             int32

```

```
dtype: object
```

3 Dataset Overview

After the data transformation, we have a dataset that comprises of 14 continuous float and 11 integer binary type variables. We also note that the data has 400 observations. When we look at the summary statistics, we see that the counts for each of the variables are all different. This indicates the presence of missing values. We also see a wide range of values, for example the sugar variable (su) has values ranging from 5 to 351. We also see that blood glucose random has a high standard deviation of 79, which indicates high variability. Lastly, we take a look at the distribution of the class variable. We see 248 instances of potential for kidney risk and 152 instances of no potential for kidney risk.

```
# Data Set Overview
print(data_with_dummies.head())
print(data_with_dummies.dtypes)
print(data_with_dummies.shape)
print("\nSummary Statistics:")
print(data_with_dummies.describe())

missing_values = data_with_dummies.isnull().sum()
print(missing_values)
print(data.shape)

# Count occurrences of each category
ckd_counts = data_with_dummies['class_ckd'].value_counts()

print(ckd_counts)

# Plotting
plt.figure(figsize=(8, 6))
```

```

ckd_counts.plot(kind='bar', color=['salmon', 'skyblue']) # Swapped color order
plt.title('Distribution of Kidney Disease Risk')
plt.xlabel('Kidney Disease Risk')
plt.ylabel('Count')
plt.xticks(ticks=[0, 1], labels=['Yes', 'No'], rotation=0) # Swapped labels
plt.show()

```

| | age | bp | sg | al | su | bgr | bu | sc | sod | pot | ... | rbc_normal | \ |
|---|------|------|-------|-----|-----|-------|------|-----|-------|-----|-----|------------|---|
| 0 | 48.0 | 80.0 | 1.020 | 1.0 | 0.0 | 121.0 | 36.0 | 1.2 | NaN | NaN | ... | 1 | |
| 1 | 7.0 | 50.0 | 1.020 | 4.0 | 0.0 | NaN | 18.0 | 0.8 | NaN | NaN | ... | 1 | |
| 2 | 62.0 | 80.0 | 1.010 | 2.0 | 3.0 | 423.0 | 53.0 | 1.8 | NaN | NaN | ... | 1 | |
| 3 | 48.0 | 70.0 | 1.005 | 4.0 | 0.0 | 117.0 | 56.0 | 3.8 | 111.0 | 2.5 | ... | 1 | |
| 4 | 51.0 | 80.0 | 1.010 | 2.0 | 0.0 | 106.0 | 26.0 | 1.4 | NaN | NaN | ... | 1 | |

| | pc_normal | pcc_present | ba_present | htn_yes | dm_yes | cad_yes | appet_poor | \ |
|---|-----------|-------------|------------|---------|--------|---------|------------|---|
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | |
| 3 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |

| | pe_yes | ane_yes |
|---|--------|---------|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 2 | 0 | 1 |
| 3 | 1 | 1 |
| 4 | 0 | 0 |

[5 rows x 25 columns]

| | |
|-----|---------|
| age | float64 |
| bp | float64 |
| sg | float64 |

```

al          float64
su          float64
bgr         float64
bu          float64
sc          float64
sod         float64
pot         float64
hemo        float64
pcv         float64
wbcc        float64
rbcc        float64
class_ckd   int32
rbc_normal  int32
pc_normal   int32
pcc_present int32
ba_present  int32
htn_yes     int32
dm_yes      int32
cad_yes     int32
appet_poor  int32
pe_yes      int32
ane_yes     int32
dtype: object
(400, 25)

```

Summary Statistics:

| | age | bp | sg | al | su | bgr \ |
|-------|------------|------------|------------|------------|------------|------------|
| count | 391.000000 | 388.000000 | 353.000000 | 354.000000 | 351.000000 | 356.000000 |
| mean | 51.483376 | 76.469072 | 1.017408 | 1.016949 | 0.450142 | 148.036517 |
| std | 17.169714 | 13.683637 | 0.005717 | 1.352679 | 1.099191 | 79.281714 |
| min | 2.000000 | 50.000000 | 1.005000 | 0.000000 | 0.000000 | 22.000000 |
| 25% | 42.000000 | 70.000000 | 1.010000 | 0.000000 | 0.000000 | 99.000000 |

| | | | | | | |
|-----|-----------|------------|----------|----------|----------|------------|
| 50% | 55.000000 | 80.000000 | 1.020000 | 0.000000 | 0.000000 | 121.000000 |
| 75% | 64.500000 | 80.000000 | 1.020000 | 2.000000 | 0.000000 | 163.000000 |
| max | 90.000000 | 180.000000 | 1.025000 | 5.000000 | 5.000000 | 490.000000 |

| | bu | sc | sod | pot | ... | rbc_normal \ |
|-------|------------|------------|------------|------------|-----|--------------|
| count | 381.000000 | 383.000000 | 313.000000 | 312.000000 | ... | 400.000000 |
| mean | 57.425722 | 3.072454 | 137.528754 | 4.627244 | ... | 0.882500 |
| std | 50.503006 | 5.741126 | 10.408752 | 3.193904 | ... | 0.322418 |
| min | 1.500000 | 0.400000 | 4.500000 | 2.500000 | ... | 0.000000 |
| 25% | 27.000000 | 0.900000 | 135.000000 | 3.800000 | ... | 1.000000 |
| 50% | 42.000000 | 1.300000 | 138.000000 | 4.400000 | ... | 1.000000 |
| 75% | 66.000000 | 2.800000 | 142.000000 | 4.900000 | ... | 1.000000 |
| max | 391.000000 | 76.000000 | 163.000000 | 47.000000 | ... | 1.000000 |

| | pc_normal | pcc_present | ba_present | htn_yes | dm_yes | cad_yes \ |
|-------|------------|-------------|------------|------------|------------|------------|
| count | 400.000000 | 400.000000 | 400.000000 | 400.000000 | 400.000000 | 400.000000 |
| mean | 0.647500 | 0.105000 | 0.055000 | 0.367500 | 0.342500 | 0.085000 |
| std | 0.478347 | 0.306937 | 0.228266 | 0.482728 | 0.475140 | 0.279231 |
| min | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 50% | 1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 75% | 1.000000 | 0.000000 | 0.000000 | 1.000000 | 1.000000 | 0.000000 |
| max | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

| | appet_poor | pe_yes | ane_yes |
|-------|------------|------------|------------|
| count | 400.000000 | 400.000000 | 400.000000 |
| mean | 0.205000 | 0.190000 | 0.150000 |
| std | 0.404207 | 0.392792 | 0.357519 |
| min | 0.000000 | 0.000000 | 0.000000 |
| 25% | 0.000000 | 0.000000 | 0.000000 |
| 50% | 0.000000 | 0.000000 | 0.000000 |
| 75% | 0.000000 | 0.000000 | 0.000000 |

max 1.000000 1.000000 1.000000

[8 rows x 25 columns]

| | |
|-------------|-----|
| age | 9 |
| bp | 12 |
| sg | 47 |
| al | 46 |
| su | 49 |
| bgr | 44 |
| bu | 19 |
| sc | 17 |
| sod | 87 |
| pot | 88 |
| hemo | 52 |
| pcv | 71 |
| wbcc | 106 |
| rbcc | 131 |
| class_ckd | 0 |
| rbc_normal | 0 |
| pc_normal | 0 |
| pcc_present | 0 |
| ba_present | 0 |
| htn_yes | 0 |
| dm_yes | 0 |
| cad_yes | 0 |
| appet_poor | 0 |
| pe_yes | 0 |
| ane_yes | 0 |

dtype: int64

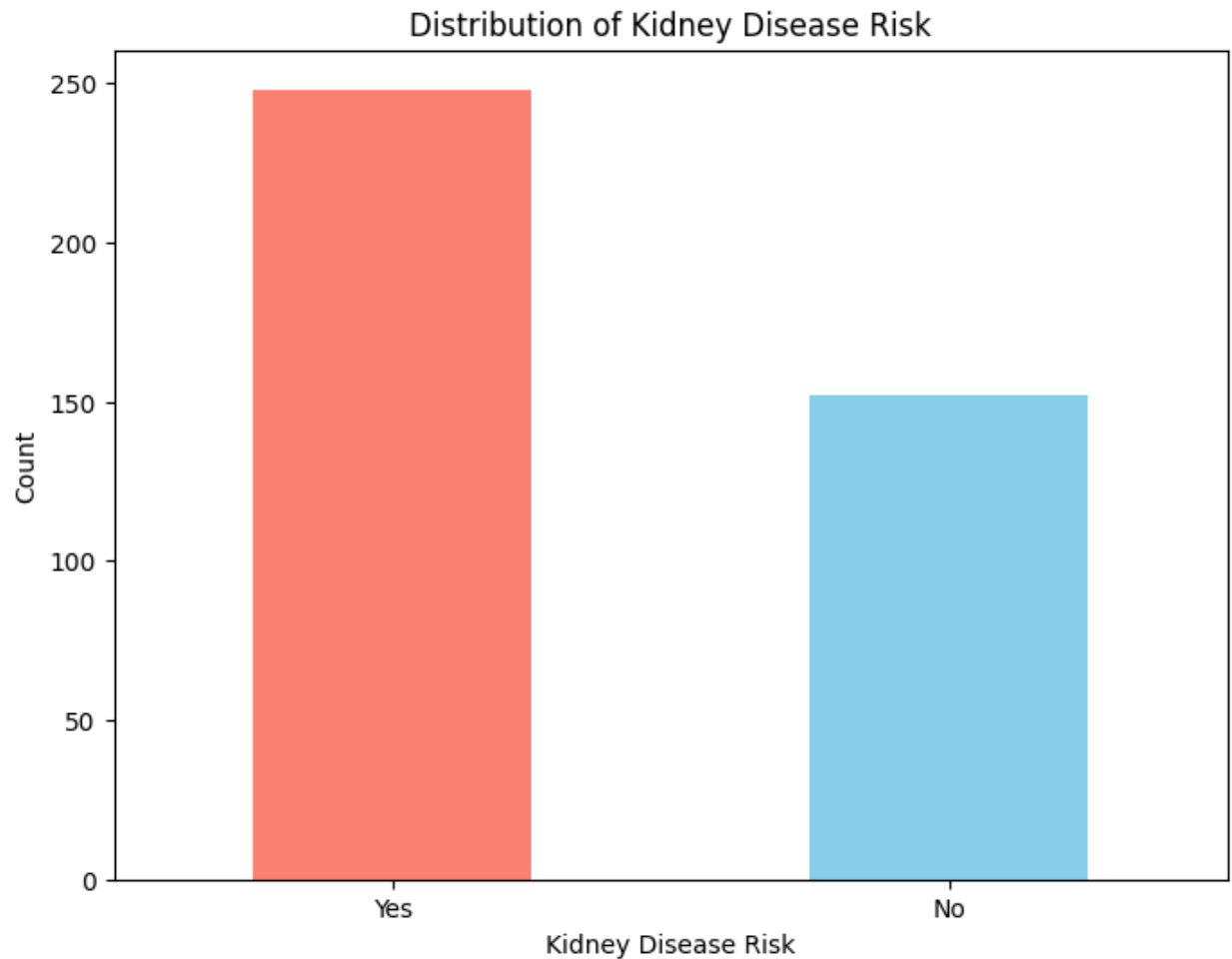
(400, 25)

class_ckd

1 248

0 152

Name: count, dtype: int64



4 Association Between Variables

Association between variables: By looking at the below heatmap we see that places of darker red or darker blue show greater correlation between the two variables. We see that the sugar variable is positively correlated to the blood glucous random variable. Intuitively this makes sense. Given the known association between diabetes and kidney disease, further exploring the relationship between blood glucose and sugar levels could be important. We can also look at the binary variable 'dm_yes' that indicates the presence of diabetes in the patient or not. We see positive correlation between diabetes and hypertension, meaning if someone has diabetes, there is a higher chance they also have

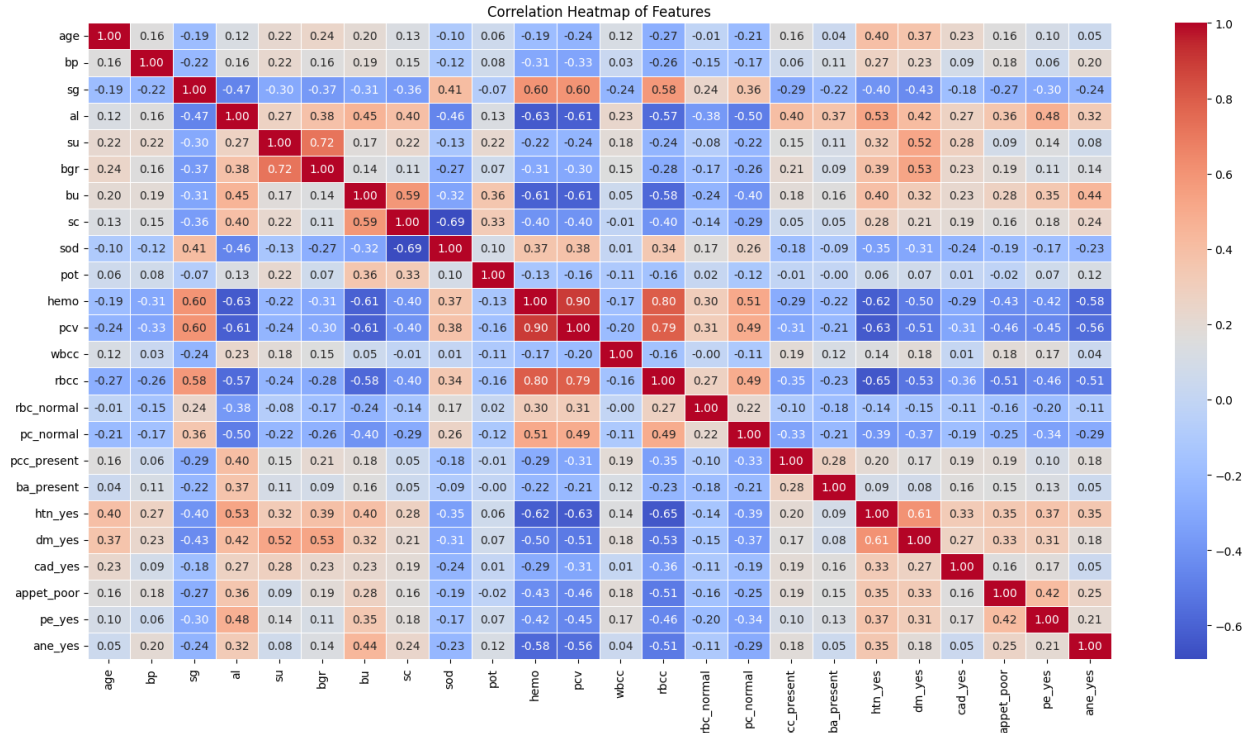
hypertension. Naturally, there is also positive correlation between diabetes indication and sugar and glucose levels. We also see strong positive correlation between hemoglobin and packed cell volume. As we have more blood cells, we have more hemoglobin, so this correlation makes sense. We also note that the blood urea variable is a common biomarker for reflecting kidney function. We note that it is negatively correlated to hemoglobin, packed cell volume, and red blood cell count, and positively correlated to sugar levels. We might be able to leverage this relationship in our feature selection and model building.

```
# Association between variables
import seaborn as sns

# remove 'class'
data_with_dummies_excluded = data_with_dummies.drop(['class_ckd'], axis=1)

# correlation matrix
correlation_matrix = data_with_dummies_excluded.corr()

# correlation heatmap
plt.figure(figsize=(20, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=.5)
plt.title('Correlation Heatmap of Features')
plt.show()
```



5 Missing Value Analysis and Handling

Missing Values: We see in the original data, that the binary columns have a low number of missing values, except for rbc. For these missing binary values, we have handled these in step 1. For all other columns except for rbc, we assign the missing values a value of 0. This change should not be very significant as there is only a few cases of missing values in these variables.

```
import pandas as pd
from ucimlrepo import fetch_ucirepo

# fetch dataset
chronic_kidney_disease = fetch_ucirepo(id=336)

# create pandas DataFrame for features (X)
X = pd.DataFrame(data=chronic_kidney_disease.data.features,
                  columns=chronic_kidney_disease.feature_names)
```

```

# create pandas DataFrame for targets (y)
y = pd.DataFrame(data=chronic_kidney_disease.data.targets,
                  columns=["class"])

# Combining features and targets into a single DataFrame
data = pd.concat([X, y], axis=1)

missing_values = data.isnull().sum()
print(missing_values)
print(data.shape)

```

| | |
|------|-----|
| age | 9 |
| bp | 12 |
| sg | 47 |
| al | 46 |
| su | 49 |
| rbc | 152 |
| pc | 65 |
| pcc | 4 |
| ba | 4 |
| bgr | 44 |
| bu | 19 |
| sc | 17 |
| sod | 87 |
| pot | 88 |
| hemo | 52 |
| pcv | 71 |
| wbcc | 106 |
| rbcc | 131 |
| htn | 2 |
| dm | 2 |

```
cad          2
appet        1
pe           1
ane          1
class        0
dtype: int64
(400, 25)
```

We see that the variables with the most missing values are red blood cell = 152, white blood cell count = 106, and red blood cell count = 131. In question 2, before transforming the data we have to handle the missing values for rbc. We use mode imputation to do this. We see that the most occurring value is normal rbc. We see that 12% of the data is abnormal rbc, and 38% of the data is missing.

```
# Count occurrences of each category
rbc_counts = data['rbc'].value_counts()
print(rbc_counts)

rbc_normal_counts = data_with_dummies['rbc_normal'].value_counts()
print(rbc_normal_counts)
```

```
rbc
normal      201
abnormal     47
Name: count, dtype: int64

rbc_normal
1      353
0       47
Name: count, dtype: int64
```

Now we can look at the missing values in the transformed data.

```

missing_values = data_with_dummies.isnull().sum()
print(missing_values)
print(data_with_dummies.shape)

# Count the number of missing values in each row
missing_values_per_row = data_with_dummies.isnull().sum(axis=1)

# Count the total number of rows with missing values
total_rows_with_missing_values = (missing_values_per_row > 0).sum()

print("Total rows with missing values:", total_rows_with_missing_values)

```

| | |
|-------------|-----|
| age | 9 |
| bp | 12 |
| sg | 47 |
| al | 46 |
| su | 49 |
| bgr | 44 |
| bu | 19 |
| sc | 17 |
| sod | 87 |
| pot | 88 |
| hemo | 52 |
| pcv | 71 |
| wbcc | 106 |
| rbcc | 131 |
| class_ckd | 0 |
| rbc_normal | 0 |
| pc_normal | 0 |
| pcc_present | 0 |
| ba_present | 0 |

```

htn_yes      0
dm_yes       0
cad_yes      0
appet_poor   0
pe_yes       0
ane_yes      0
dtype: int64
(400, 25)
Total rows with missing values: 197

```

We see that 197 out of 400 rows have missing values. We use soft imputation. We want to be able to capture complex relationships and dependencies. We also want to minimize the amount of bias in our data that can come from imputing with a single point, especially since there are many rows of missing values. Before we perform imputation we need to check if we should scale our data.

```

print(data_with_dummies.mean())
print(data_with_dummies.var())

```

```

age          51.483376
bp           76.469072
sg           1.017408
al           1.016949
su           0.450142
bgr          148.036517
bu           57.425722
sc           3.072454
sod          137.528754
pot           4.627244
hemo         12.526437
pcv          38.884498
wbcc         8406.122449
rbcc         4.707435

```


| | |
|----------------|--------------|
| class_ckd | 0.620000 |
| rbc_normal | 0.882500 |
| pc_normal | 0.647500 |
| pcc_present | 0.105000 |
| ba_present | 0.055000 |
| htn_yes | 0.367500 |
| dm_yes | 0.342500 |
| cad_yes | 0.085000 |
| appet_poor | 0.205000 |
| pe_yes | 0.190000 |
| ane_yes | 0.150000 |
| dtype: float64 | |
| age | 2.947991e+02 |
| bp | 1.872419e+02 |
| sg | 3.267971e-05 |
| al | 1.829740e+00 |
| su | 1.208221e+00 |
| bgr | 6.285590e+03 |
| bu | 2.550554e+03 |
| sc | 3.296053e+01 |
| sod | 1.083421e+02 |
| pot | 1.020102e+01 |
| hemo | 8.483161e+00 |
| pcv | 8.082198e+01 |
| wbcc | 8.669928e+06 |
| rbcc | 1.051288e+00 |
| class_ckd | 2.361905e-01 |
| rbc_normal | 1.039536e-01 |
| pc_normal | 2.288158e-01 |
| pcc_present | 9.421053e-02 |
| ba_present | 5.210526e-02 |
| htn_yes | 2.330263e-01 |

```

dm_yes          2.257581e-01
cad_yes         7.796992e-02
appet_poor      1.633835e-01
pe_yes          1.542857e-01
ane_yes         1.278195e-01
dtype: float64

```

We must scale the data since wbcc has a mean of 8406 which is much higher than the others. We must scale our data, but we have to exclude the binary variables since there are encoded with a 1 or 0.

```

from sklearn.preprocessing import StandardScaler

# Extract continuous variables from the DataFrame
continuous_column_names = ['age', 'bp', 'sg', 'al', 'su', 'bgr',
                           'bu', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wbcc', 'rbcc']
continuous_data = data_with_dummies[continuous_column_names]

# Initialize StandardScaler
scaler = StandardScaler()

# Scale the continuous variables
scaled_continuous_data = scaler.fit_transform(continuous_data)

# Create a DataFrame with scaled continuous variables and original binary variables
scaled_data = pd.DataFrame(scaled_continuous_data,
                           columns=continuous_column_names, index=data_with_dummies.index)
scaled_data[['class_ckd', 'rbc_normal', 'pc_normal', 'pcc_present', 'ba_present',
             'htn_yes', 'dm_yes', 'cad_yes', 'appet_poor', 'pe_yes', 'ane_yes']]
    = data_with_dummies[['class_ckd', 'rbc_normal', 'pc_normal',
                        'pcc_present', 'ba_present', 'htn_yes', 'dm_yes',
                        'cad_yes', 'appet_poor', 'pe_yes', 'ane_yes']]

```

```

from fancyimpute import SoftImpute
import pandas as pd

# Perform imputation using SoftImpute
data_imputed = SoftImpute().fit_transform(scaled_data)

# Original column names
original_column_names = [
    'age', 'bp', 'sg', 'al', 'su', 'bgr', 'bu', 'sc', 'sod', 'pot',
    'hemo', 'pcv', 'wbcc', 'rbcc', 'class_ckd', 'rbc_normal', 'pc_normal',
    'pcc_present', 'ba_present', 'htn_yes', 'dm_yes', 'cad_yes', 'appet_poor',
    'pe_yes', 'ane_yes'
]

# Convert the imputed array back to a DataFrame with original column names
data_imputed_df = pd.DataFrame(data_imputed, columns=original_column_names)

# Apply a threshold to binary variables
binary_column_names = ['class_ckd', 'rbc_normal', 'pc_normal', 'pcc_present',
                        'ba_present', 'htn_yes', 'dm_yes', 'cad_yes', 'appet_poor',
                        'pe_yes', 'ane_yes']

threshold = 0.5

data_imputed_df[binary_column_names] = (data_imputed_df[binary_column_names] >
                                         threshold).astype(int)

```

```

[SoftImpute] Max Singular Value of X_init = 42.867770
[SoftImpute] Iter 1: observed MAE=0.030561 rank=25
[SoftImpute] Iter 2: observed MAE=0.030592 rank=25
[SoftImpute] Iter 3: observed MAE=0.030620 rank=25
[SoftImpute] Iter 4: observed MAE=0.030647 rank=25
[SoftImpute] Iter 5: observed MAE=0.030670 rank=25

```

[SoftImpute] Iter 6: observed MAE=0.030692 rank=25
[SoftImpute] Iter 7: observed MAE=0.030712 rank=25
[SoftImpute] Iter 8: observed MAE=0.030730 rank=25
[SoftImpute] Iter 9: observed MAE=0.030746 rank=25
[SoftImpute] Iter 10: observed MAE=0.030761 rank=25
[SoftImpute] Iter 11: observed MAE=0.030775 rank=25
[SoftImpute] Iter 12: observed MAE=0.030787 rank=25
[SoftImpute] Iter 13: observed MAE=0.030799 rank=25
[SoftImpute] Iter 14: observed MAE=0.030810 rank=25
[SoftImpute] Iter 15: observed MAE=0.030819 rank=25
[SoftImpute] Iter 16: observed MAE=0.030828 rank=25
[SoftImpute] Iter 17: observed MAE=0.030837 rank=25
[SoftImpute] Iter 18: observed MAE=0.030844 rank=25
[SoftImpute] Iter 19: observed MAE=0.030851 rank=25
[SoftImpute] Iter 20: observed MAE=0.030858 rank=25
[SoftImpute] Iter 21: observed MAE=0.030863 rank=25
[SoftImpute] Iter 22: observed MAE=0.030869 rank=25
[SoftImpute] Iter 23: observed MAE=0.030874 rank=25
[SoftImpute] Iter 24: observed MAE=0.030879 rank=25
[SoftImpute] Iter 25: observed MAE=0.030883 rank=25
[SoftImpute] Iter 26: observed MAE=0.030887 rank=25
[SoftImpute] Iter 27: observed MAE=0.030891 rank=25
[SoftImpute] Iter 28: observed MAE=0.030894 rank=25
[SoftImpute] Iter 29: observed MAE=0.030897 rank=25
[SoftImpute] Iter 30: observed MAE=0.030900 rank=25
[SoftImpute] Iter 31: observed MAE=0.030903 rank=25
[SoftImpute] Iter 32: observed MAE=0.030905 rank=25
[SoftImpute] Iter 33: observed MAE=0.030907 rank=25
[SoftImpute] Iter 34: observed MAE=0.030910 rank=25
[SoftImpute] Iter 35: observed MAE=0.030912 rank=25
[SoftImpute] Iter 36: observed MAE=0.030914 rank=25
[SoftImpute] Iter 37: observed MAE=0.030915 rank=25

[SoftImpute] Iter 38: observed MAE=0.030917 rank=25
[SoftImpute] Iter 39: observed MAE=0.030918 rank=25
[SoftImpute] Iter 40: observed MAE=0.030920 rank=25
[SoftImpute] Iter 41: observed MAE=0.030921 rank=25
[SoftImpute] Iter 42: observed MAE=0.030922 rank=25
[SoftImpute] Iter 43: observed MAE=0.030924 rank=25
[SoftImpute] Iter 44: observed MAE=0.030925 rank=25
[SoftImpute] Iter 45: observed MAE=0.030926 rank=25
[SoftImpute] Iter 46: observed MAE=0.030927 rank=25
[SoftImpute] Iter 47: observed MAE=0.030928 rank=25
[SoftImpute] Iter 48: observed MAE=0.030929 rank=25
[SoftImpute] Iter 49: observed MAE=0.030930 rank=25
[SoftImpute] Iter 50: observed MAE=0.030930 rank=25
[SoftImpute] Iter 51: observed MAE=0.030931 rank=25
[SoftImpute] Iter 52: observed MAE=0.030932 rank=25
[SoftImpute] Iter 53: observed MAE=0.030932 rank=25
[SoftImpute] Iter 54: observed MAE=0.030933 rank=25
[SoftImpute] Iter 55: observed MAE=0.030934 rank=25
[SoftImpute] Iter 56: observed MAE=0.030934 rank=25
[SoftImpute] Iter 57: observed MAE=0.030935 rank=25
[SoftImpute] Iter 58: observed MAE=0.030935 rank=25
[SoftImpute] Iter 59: observed MAE=0.030936 rank=25
[SoftImpute] Iter 60: observed MAE=0.030936 rank=25
[SoftImpute] Iter 61: observed MAE=0.030936 rank=25
[SoftImpute] Iter 62: observed MAE=0.030937 rank=25
[SoftImpute] Iter 63: observed MAE=0.030937 rank=25
[SoftImpute] Iter 64: observed MAE=0.030937 rank=25
[SoftImpute] Iter 65: observed MAE=0.030938 rank=25
[SoftImpute] Iter 66: observed MAE=0.030938 rank=25
[SoftImpute] Iter 67: observed MAE=0.030938 rank=25
[SoftImpute] Iter 68: observed MAE=0.030939 rank=25
[SoftImpute] Iter 69: observed MAE=0.030939 rank=25

```
[SoftImpute] Iter 70: observed MAE=0.030939 rank=25
[SoftImpute] Iter 71: observed MAE=0.030939 rank=25
[SoftImpute] Iter 72: observed MAE=0.030939 rank=25
[SoftImpute] Iter 73: observed MAE=0.030940 rank=25
[SoftImpute] Iter 74: observed MAE=0.030940 rank=25
[SoftImpute] Iter 75: observed MAE=0.030940 rank=25
[SoftImpute] Iter 76: observed MAE=0.030940 rank=25
[SoftImpute] Iter 77: observed MAE=0.030940 rank=25
[SoftImpute] Iter 78: observed MAE=0.030941 rank=25
[SoftImpute] Iter 79: observed MAE=0.030941 rank=25
[SoftImpute] Iter 80: observed MAE=0.030941 rank=25
[SoftImpute] Iter 81: observed MAE=0.030941 rank=25
[SoftImpute] Iter 82: observed MAE=0.030941 rank=25
[SoftImpute] Iter 83: observed MAE=0.030941 rank=25
[SoftImpute] Iter 84: observed MAE=0.030941 rank=25
[SoftImpute] Iter 85: observed MAE=0.030941 rank=25
[SoftImpute] Iter 86: observed MAE=0.030942 rank=25
[SoftImpute] Iter 87: observed MAE=0.030942 rank=25
[SoftImpute] Iter 88: observed MAE=0.030942 rank=25
[SoftImpute] Iter 89: observed MAE=0.030942 rank=25
[SoftImpute] Iter 90: observed MAE=0.030942 rank=25
[SoftImpute] Iter 91: observed MAE=0.030942 rank=25
[SoftImpute] Iter 92: observed MAE=0.030942 rank=25
[SoftImpute] Iter 93: observed MAE=0.030942 rank=25
[SoftImpute] Iter 94: observed MAE=0.030942 rank=25
[SoftImpute] Iter 95: observed MAE=0.030942 rank=25
[SoftImpute] Stopped after iteration 95 for lambda=0.857355
```

Lets double check that our target variable has not changed.

```
#double check data set
data_imputed_df.head(15)
```

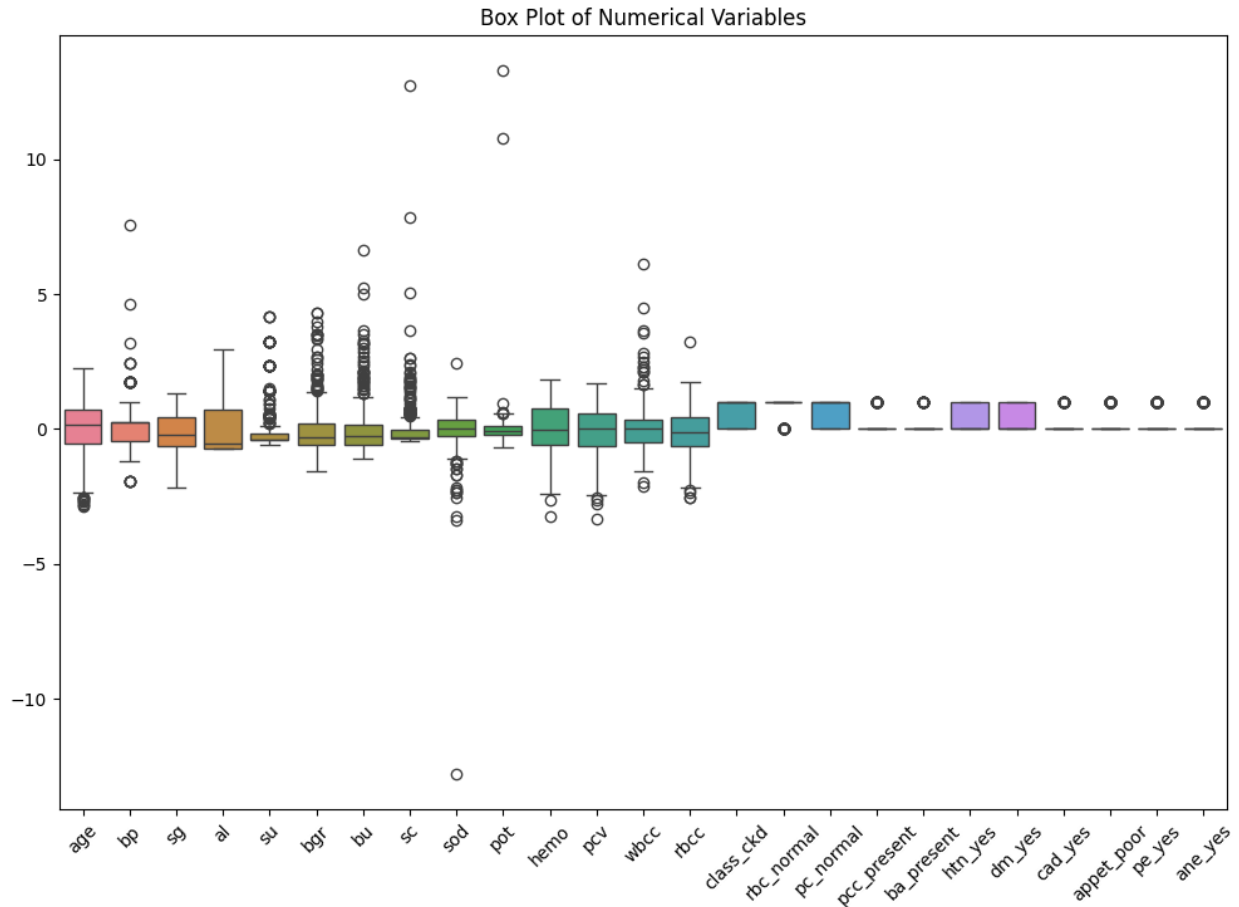
```
ckd_counts = data_with_dummies['class_ckd'].value_counts()
print(ckd_counts) # same output
```

```
class_ckd
1      248
0      152
Name: count, dtype: int64
```

6 Outlier Analysis

```
import seaborn as sns
import matplotlib.pyplot as plt

# Plot box plots for numerical variables
plt.figure(figsize=(12, 8))
sns.boxplot(data=data_imputed_df)
plt.xticks(rotation=45)
plt.title('Box Plot of Numerical Variables')
plt.show()
```



- Each variables are measures of biological variables which are in continuous scale in real life and removing them could distort the distributions.
- As shown in the boxplots, the abundance of outliers lined up closely together suggests getting rid of them would yield inaccurate outcomes or misinterpretations. With only 400 observations, removing outliers risks losing important variables within the sample population.
- Removing outliers could introduce bias and overfitting, especially if the model is trained on a cleaned dataset that does not reflect the true variability and complexity of the real-world data.
- Outliers can significantly affect correlation coefficients and relationships between continuous variables. Removing outliers may affect strengths or weakness of correlations, leading to incorrect conclusions about the associations between variables.

```
import pandas as pd
import numpy as np
```



```

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns

from sklearn.preprocessing import scale
from sklearn.decomposition import PCA, TruncatedSVD
from sklearn.cluster import KMeans
from scipy.cluster import hierarchy
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_samples, silhouette_score
from sklearn.metrics.cluster import rand_score
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LinearRegression
import statsmodels.api as sm
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix, roc_curve, roc_auc_score
from sklearn import neighbors

```

7 Sub-group analysis

```

# Select relevant columns for clustering
col_cluster = data_imputed_df[['age', 'bp', 'sg', 'al', 'su', 'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo']]

range_n_clusters = [3, 5, 7, 9]
for n_clusters in range_n_clusters:
    km = KMeans(n_clusters = n_clusters, n_init = 20, random_state=0)
    cluster_labels_km = km.fit_predict(col_cluster)

```

```

silhouette_avg_km = silhouette_score(col_cluster, cluster_labels_km)
# Compute the silhouette scores for each sample
sample_silhouette_values = silhouette_samples(col_cluster, cluster_labels_km)
fig, ax1 = plt.subplots(1, 1)
fig.set_size_inches(18, 7)
ax1.set_xlim([-0.25, 1])# change this based on the silhouette range

y_lower = 10

for i in range(n_clusters):
    # Aggregate the silhouette scores for samples belonging to
    # cluster i, and sort them
    ith_cluster_silhouette_values = sample_silhouette_values[cluster_labels_km == i]

    ith_cluster_silhouette_values.sort()

    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i

    color = cm.nipy_spectral(float(i) / n_clusters)
    ax1.fill_betweenx(
        np.arange(y_lower, y_upper),
        0,
        ith_cluster_silhouette_values,
        facecolor=color,
        edgecolor=color,
        alpha=0.7,
    )

    # Label the silhouette plots with their cluster numbers at the middle
    ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

```

```

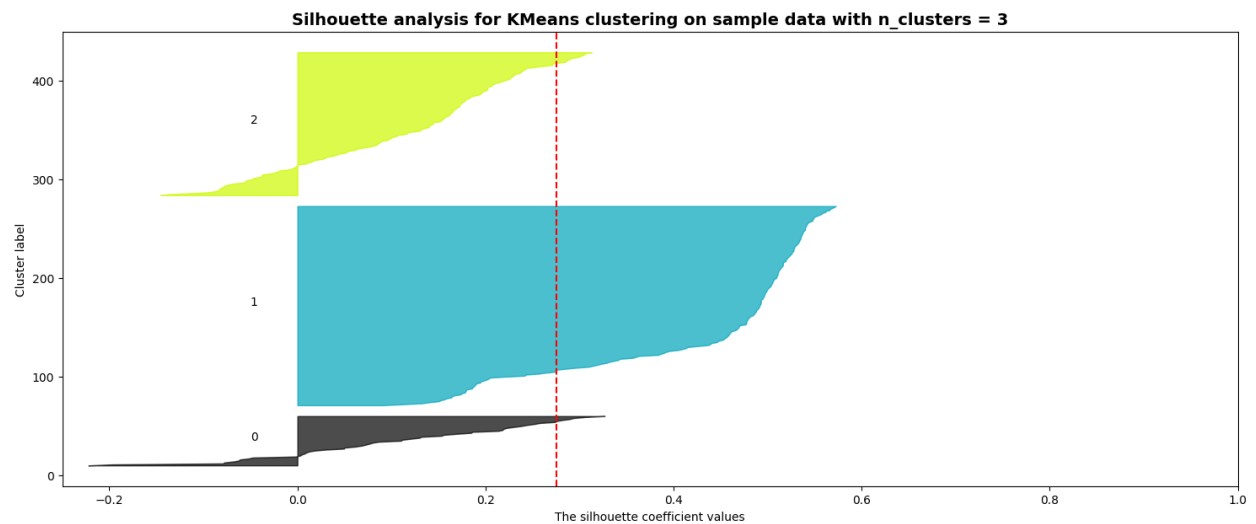
# Compute the new y_lower for next plot
y_lower = y_upper + 10

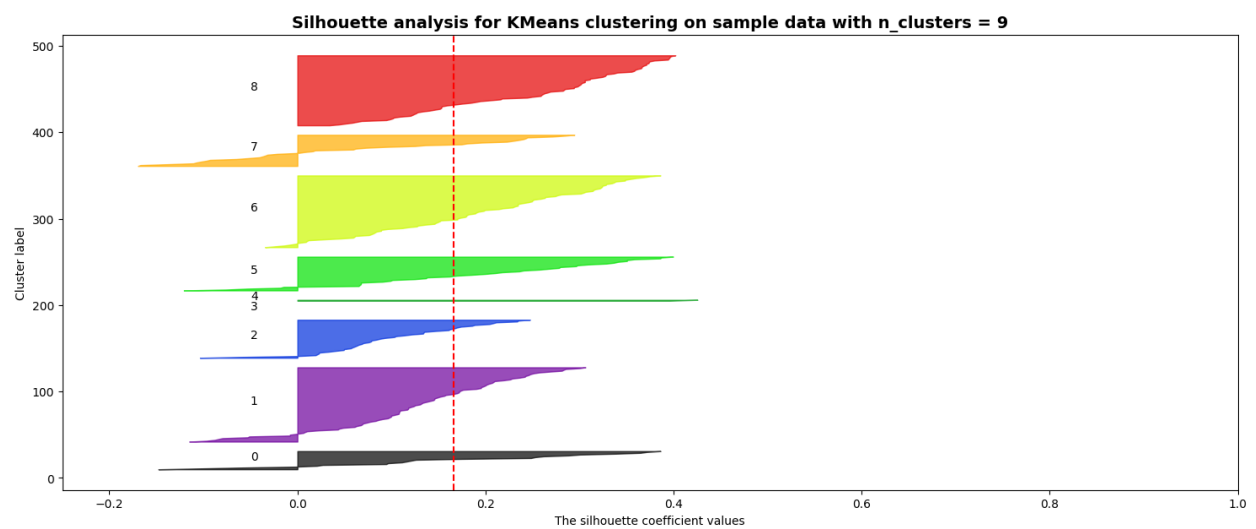
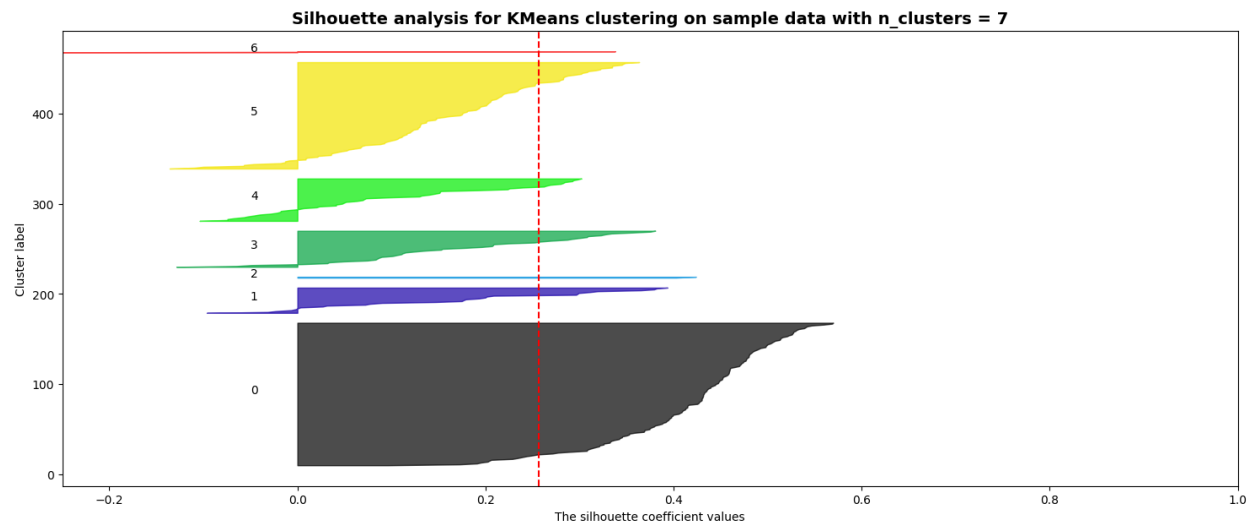
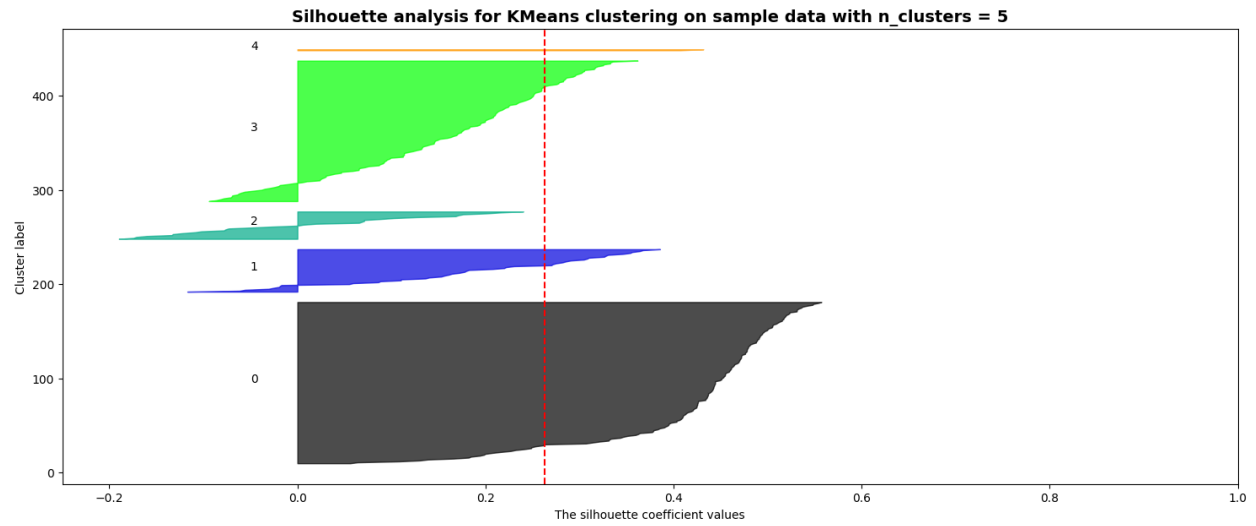
ax1.set_title("The silhouette plot for various cluster")
ax1.set_xlabel("The silhouette coefficient values")
ax1.set_ylabel("Cluster label")

# The vertical line for average silhouette score of all the values
ax1.axvline(x=silhouette_avg_km, color="red", linestyle="--")

plt.title(
    "Silhouette analysis for KMeans clustering on sample data with n_clusters = %d"
    % n_clusters,
    fontsize=14,
    fontweight="bold",
)

```





```

from sklearn.metrics import silhouette_score

# Loop over different numbers of clusters
for n_clusters in range_n_clusters:
    km = KMeans(n_clusters=n_clusters, n_init=20, random_state=0)
    cluster_labels_km = km.fit_predict(data_imputed_df)

    # Compute silhouette score
    silhouette_avg_km = silhouette_score(data_imputed_df, cluster_labels_km)
    print(f"Silhouette score for {n_clusters} clusters: {silhouette_avg_km}")

```

```

Silhouette score for 3 clusters: 0.2445335895621168
Silhouette score for 5 clusters: 0.2399834098139783
Silhouette score for 7 clusters: 0.22299493021091382
Silhouette score for 9 clusters: 0.11646996900994452

```

8 Data Splitting

```

x = data_imputed_df.drop(['class_ckd'], axis = 1)
y = data_imputed_df['class_ckd']

```

```

np.random.seed(1)

x_train, x_test, y_train, y_test = train_test_split(
    x,
    y,
    test_size=0.30,
    random_state=0,
    stratify=y
)

```

9 Classifier Choices

Logistic Regression

```
ckd_log = LogisticRegression()  
ckd_log.fit(x_train, y_train)
```

```
LogisticRegression()
```

```
pred_prob = ckd_log.predict_proba(x_test)
```

```
# Create a data frame with predicted probabilities (for default)  
# and the class labels in the test set.  
  
# get the prob being 1  
df = pd.DataFrame(  
    data = {'prob1': pred_prob[:,1], 'y_test': y_test}  
)  
df.head()
```

| | prob1 | y_test |
|-----|----------|--------|
| 32 | 0.999737 | 1 |
| 80 | 0.999051 | 1 |
| 198 | 0.999997 | 1 |
| 118 | 0.998953 | 1 |
| 175 | 0.993894 | 1 |

```
# Use cutoff = 0.5 and compute misclassification error, sensitivity, and specificity.  
df['y_test_pred'] = df.prob1.map(lambda x: 1 if x > 0.5 else 0)
```

```
cm = confusion_matrix(df.y_test, df.y_test_pred)
print('Confusion Matrix : \n', cm)
```

Confusion Matrix :

```
[[45  1]
 [ 1 73]]
```

```
# From confusion matrix, calculate accuracy
```

```
total = sum(sum(cm))
```

```
accuracy = (cm[0,0]+cm[1,1])/total
```

```
print ('Accuracy : ', accuracy)
```

```
sensitivity = cm[1,1]/(cm[1,0]+cm[1,1])
```

```
print('Sensitivity : ', sensitivity )
```

```
specificity = cm[0,0]/(cm[0,0]+cm[0,1])
```

```
print('Specificity : ', specificity)
```

Accuracy : 0.9833333333333333

Sensitivity : 0.9864864864864865

Specificity : 0.9782608695652174

KNN

```
k_range = range(1, 15)
```

```
scores = []
```

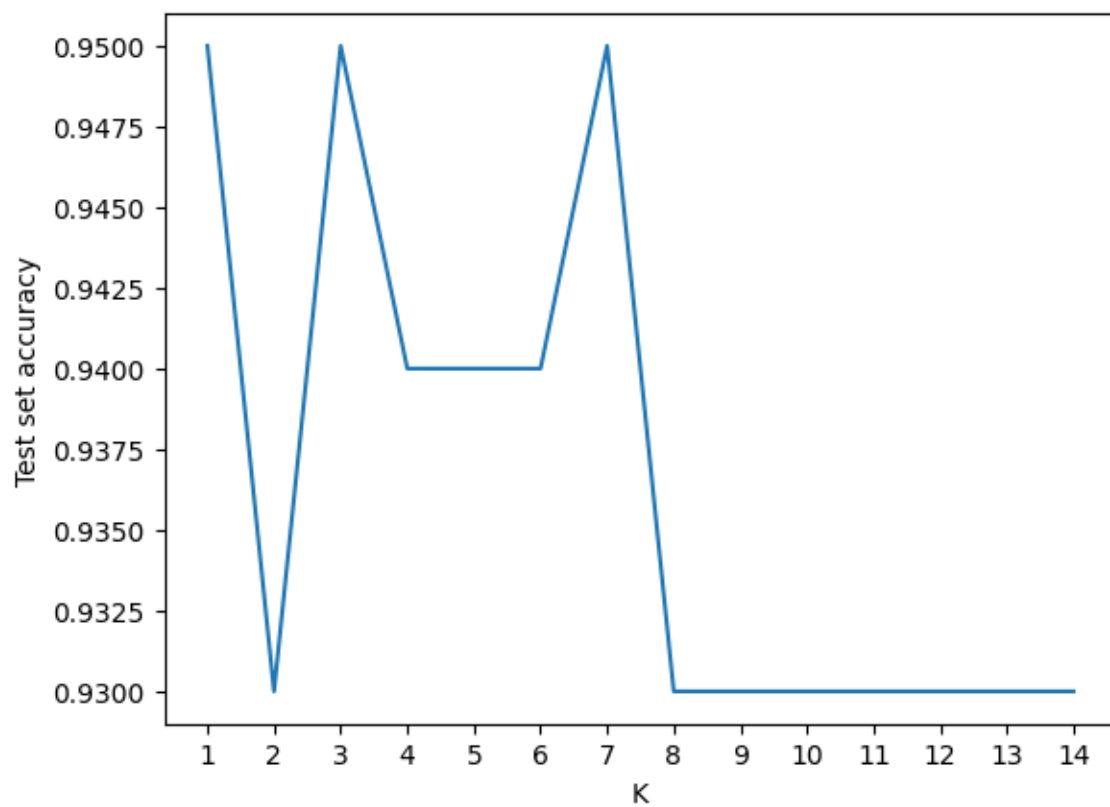
```
for k in k_range:
```

```
    knn = neighbors.KNeighborsClassifier(n_neighbors=k)
```

```
    knn.fit(x_train, y_train)
```

```
y_pred = knn.predict(x_test)
scores.append(round(metrics.accuracy_score(y_test, y_pred),2))
```

```
plt.plot(k_range, scores)
plt.xlabel('K')
plt.ylabel('Test set accuracy')
plt.xticks(range(1,15))
plt.show()
```



```
# Let K = 3
knn5 = neighbors.KNeighborsClassifier(
    n_neighbors = 3,
    algorithm='brute'
)
```



```
# train the model
knn5.fit(x_train, y_train)

# predict on the test set
pred5 = knn5.predict(x_test)
```

```
# accuracy on the test set
print(round(metrics.accuracy_score(y_test, pred5),2))
```

0.95

```
conf_matrix = confusion_matrix(y_test, pred5)
conf_matrix
```

```
array([[45,  1],
       [ 5, 69]], dtype=int64)
```

1. Logistic Regression: Logistic regression provides easily interpretable results, as the coefficients represent the impact of each feature on the predicted probability of the target class. Additionally, logistic regression possesses several features that enhance its effectiveness. For instance, when transform categorical variables to dummy variables enable logistic regression to handle categorical efficiently.
2. KNN: Since our data involves abundance of outliers, we chose KNN which is relatively robust to outliers since it relies on the majority vote of neighbors, which can help mitigate the influence of outliers on the predictions. Also, KNN is fairly easy to understand and implement, making it a good choice for exploratory data analysis.

10 Performance Metrics

One performance metric that we will look at will be the accuracy of the two models. Specifically we will look at the specificity and sensitivity of the models to get a better understanding of how they are classifying points. We can also look at the confusion matrix here.

The other performance metric that we will look at will be the root mean squared error of the models, where we want the lowest possible error between the two.

11 Feature Selection/Extraction

```
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, roc_curve, roc_auc_score
import statsmodels.api as sm
```

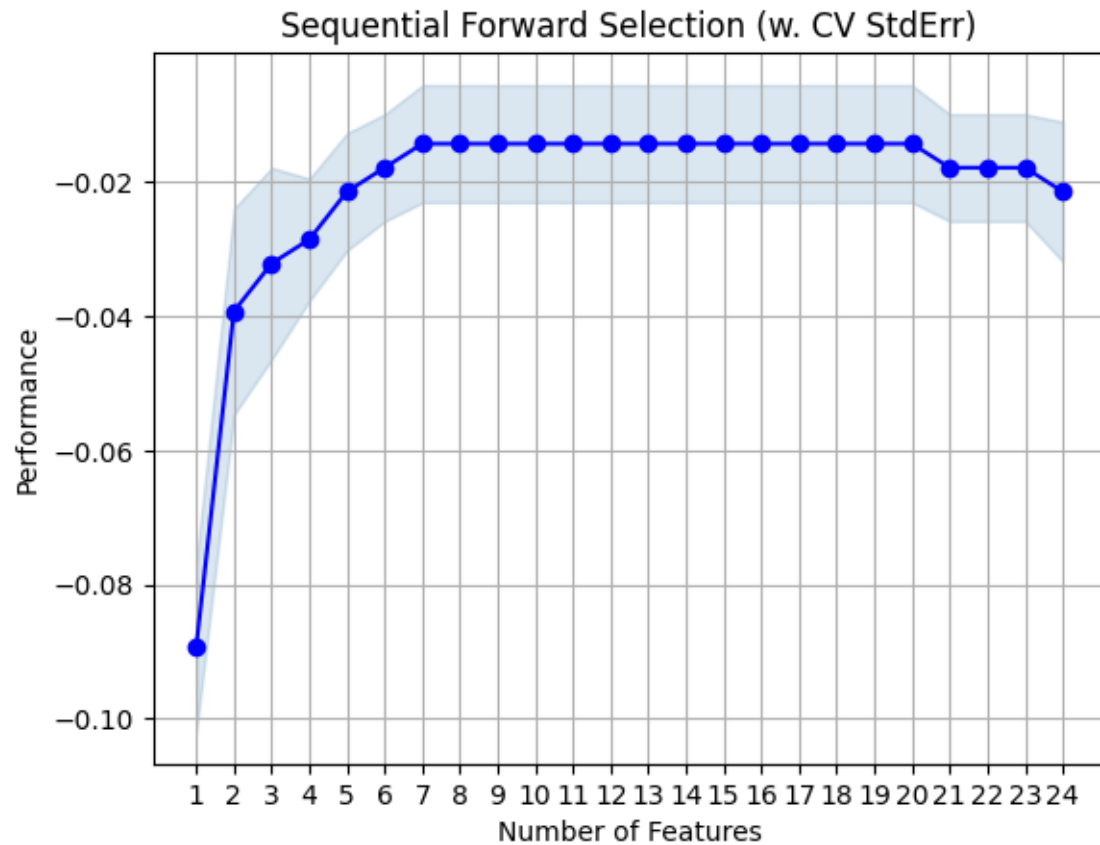
For logistic regression model

```
sfs = SFS(
    ckd_log,
    k_features=(1,24),
    forward=True,
    floating=False,
    scoring='neg_mean_squared_error',
    cv=5
)
```

```
sfs = sfs.fit(x_train, y_train)
```

```
fig = plot_sfs(sfs.get_metric_dict(), kind='std_err')

plt.title('Sequential Forward Selection (w. CV StdErr)')
plt.grid()
plt.show()
```



```
x_train.columns[list(sfs.k_feature_idx_)]
```

```
Index(['bp', 'sg', 'al', 'su', 'bgr', 'hemo', 'appet_poor'], dtype='object')
```

```
# Prediction on hold-out set
sel_col = x_train.columns[list(sfs.k_feature_idx_)]
X_train_sfs = x_train[sel_col]
X_test_sfs = x_test[sel_col]
sfs_m = LogisticRegression()
sfs_m.fit(X_train_sfs, y_train)
sfs_test = sfs_m.predict(X_test_sfs)
np.sqrt(metrics.mean_squared_error(y_test, sfs_test))
```

```
0.12909944487358055
```

```
cm_sfs = confusion_matrix(y_test, sfs_test)
print('Confusion Matrix : \n', cm_sfs)
```

Confusion Matrix :

```
[[45  1]
 [ 1 73]]
```

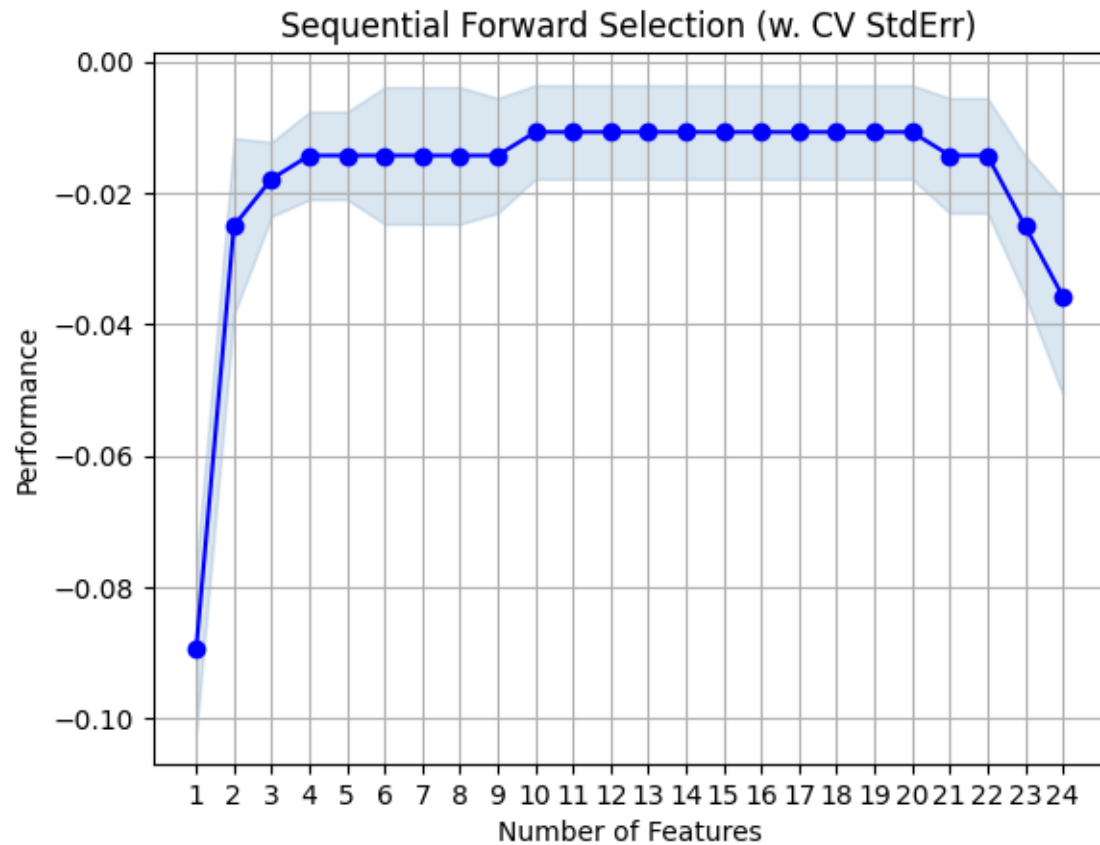
For KNN

```
sfs = SFS(
    knn5,
    k_features=(1,24),
    forward=True,
    floating=False,
    scoring='neg_mean_squared_error',
    cv=5
)
```

```
sfs = sfs.fit(x_train, y_train)
```

```
fig = plot_sfs(sfs.get_metric_dict(), kind='std_err')

plt.title('Sequential Forward Selection (w. CV StdErr)')
plt.grid()
plt.show()
```



```
x_train.columns[list(sfs.k_feature_idx_)]
```

```
Index(['sg', 'al', 'su', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wbcc',
      'appet_poor'],
      dtype='object')
```

```
# Prediction on hold-out set
sel_col = x_train.columns[list(sfs.k_feature_idx_)]
X_train_sfs = x_train[sel_col]
X_test_sfs = x_test[sel_col]
sfs_m = neighbors.KNeighborsClassifier(
    n_neighbors = 3,
    algorithm='brute'
)
sfs_m.fit(X_train_sfs, y_train)
```

```
sfs_test = sfs_m.predict(X_test_sfs)
np.sqrt(metrics.mean_squared_error(y_test, sfs_test))
```

0.12909944487358055

```
cm_sfs = confusion_matrix(y_test, sfs_test)
print('Confusion Matrix : \n', cm_sfs)
```

Confusion Matrix :

```
[[45  1]
 [ 1 73]]
```

12 Classifier Comparison

For Logistic Regression

```
x = data_imputed_df.drop(['class_ckd'], axis = 1)
#using most optimal features
x_lr = x[['bp', 'sg', 'al', 'su', 'bgr', 'hemo', 'appet_poor']]
y = data_imputed_df['class_ckd']
```

```
np.random.seed(1)

x_train, x_test, y_train, y_test = train_test_split(
    x_lr,
    y,
    test_size=0.30,
    random_state=0,
    stratify=y
)
```

```
ckd_log = LogisticRegression()
ckd_log.fit(x_train, y_train)
```

```
LogisticRegression()
```

```
pred_prob = ckd_log.predict_proba(x_test)
```

```
# Create a data frame with predicted probabilities (for default)
# and the class labels in the test set.

# get the prob being 1
df = pd.DataFrame(
    data = {'prob1': pred_prob[:,1], 'y_test': y_test}
)
df.head()
```

| | prob1 | y_test |
|-----|----------|--------|
| 32 | 0.999471 | 1 |
| 80 | 0.990577 | 1 |
| 198 | 0.999965 | 1 |
| 118 | 0.999423 | 1 |
| 175 | 0.976012 | 1 |

Now, find KS plot to find optimal threshold.

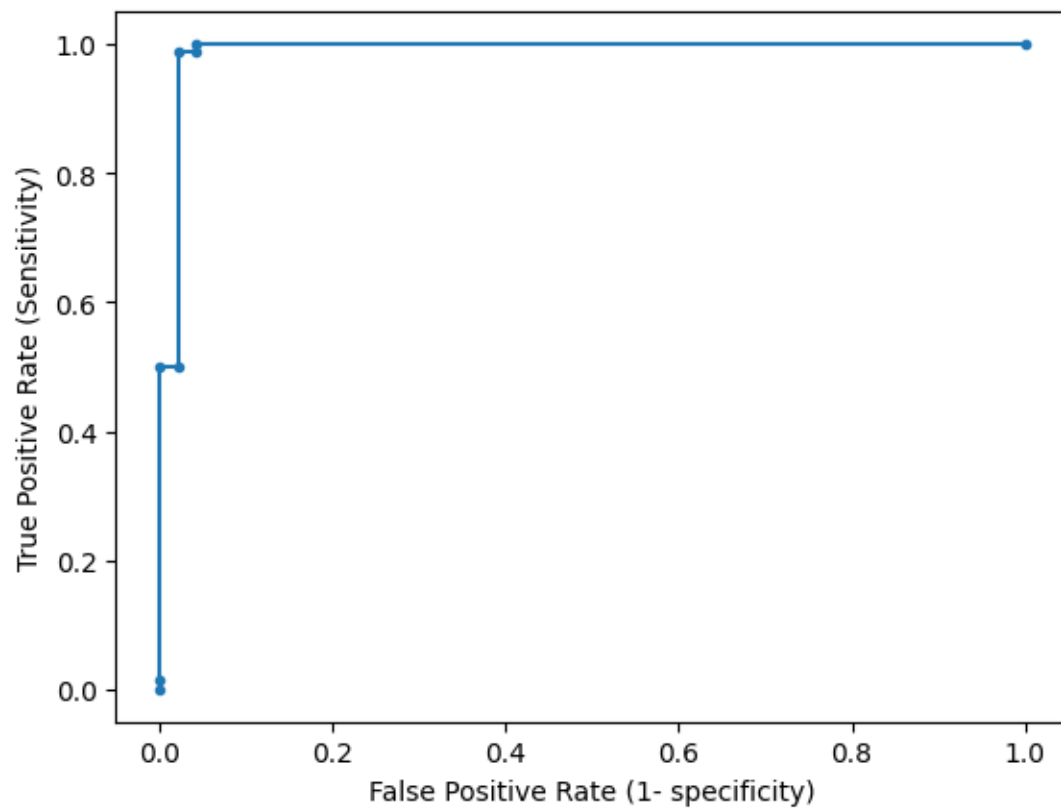
```
fpr, tpr, thresholds = roc_curve(df.y_test, df.prob1)
```

```
# AUC area under curve - if model is above line y = x
# then it is a good model (better than randomly assigning)
roc_auc_score(df.y_test, df.prob1)
```

```
0.9888366627497063
```

```
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.', label='Logistic')
plt.xlabel('False Positive Rate (1- specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```



```
indices = np.where(np.isclose(fpr, 0.02, atol=0.01))
print(thresholds[indices])
print(tpr[indices])
print(1-fpr[indices])
```

```
[0.99726002 0.51198128]
[0.5          0.98648649]
[0.97826087 0.97826087]
```



```
j_statistic = tpr - fpr
optimal_index = np.argmax(j_statistic)
optimal_threshold = thresholds[optimal_index]
optimal_threshold
```

0.5119812831944971

```
ind = np.where(np.isclose(thresholds, optimal_threshold, atol=0.001))
print(tpr[ind])
print(1-fpr[ind])
```

[0.98648649]

[0.97826087]

```
ks_statistic = np.max(tpr - fpr)
ks_threshold = thresholds[np.argmax(tpr - fpr)]
ks_threshold
```

0.5119812831944971

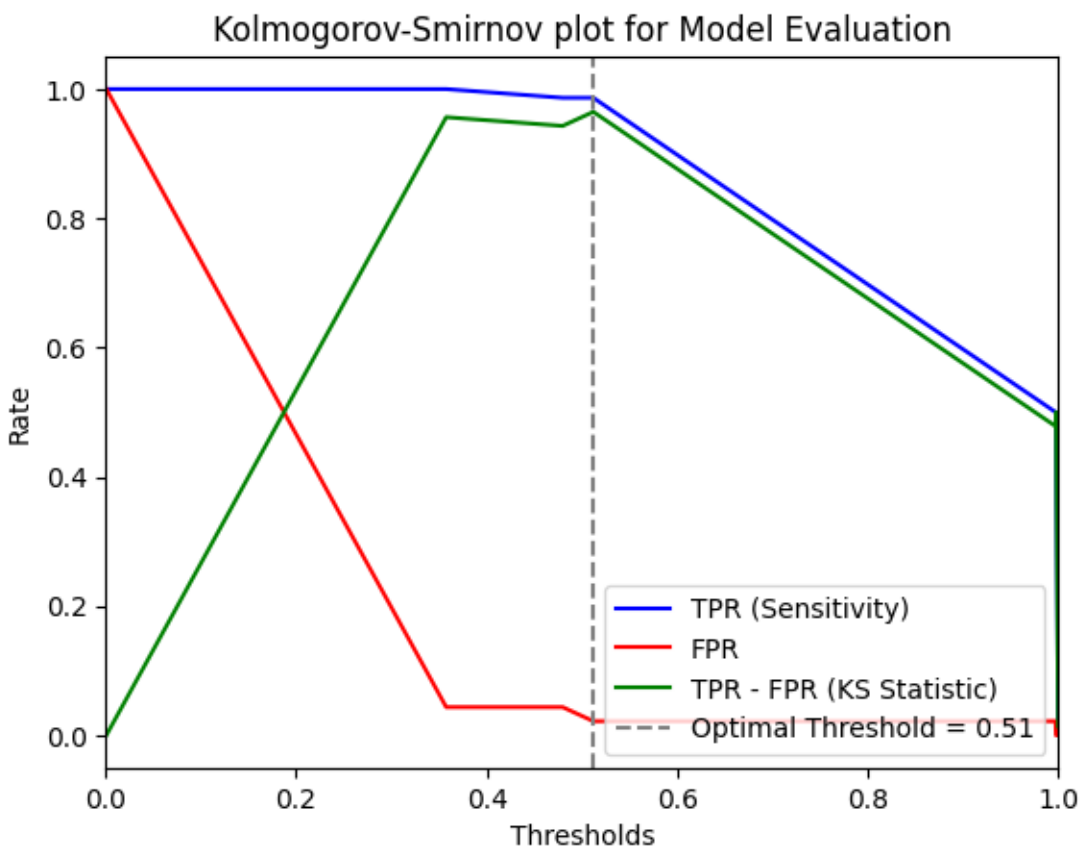
```
ind = np.where(np.isclose(thresholds, ks_threshold, atol=0.001)) ### atol is tolerance limit
print(tpr[ind])
print(1-fpr[ind])
```

[0.98648649]

[0.97826087]

```
plt.plot(thresholds, tpr, label='TPR (Sensitivity)', color='blue')
plt.plot(thresholds, fpr, label='FPR', color='red')
plt.plot(thresholds, tpr - fpr, label='TPR - FPR (KS Statistic)', color='green')
plt.axvline(x=ks_threshold, color='grey', linestyle='--',
            label=f'Optimal Threshold = {ks_threshold:.2f}')
```

```
plt.title('Kolmogorov-Smirnov plot for Model Evaluation')
plt.xlabel('Thresholds')
plt.ylabel('Rate')
plt.legend()
plt.xlim([0.0, 1.0])
#plt.gca().invert_xaxis()
plt.show()
```



```
# Use cutoff = optimal threshold and compute
# misclassification error, sensitivity, and specificity.
df['y_test_pred'] = df.probl.map(lambda x: 1 if x>=0.5119812831944971 else 0)
```

```
cm = confusion_matrix(df.y_test, df.y_test_pred)
print('Confusion Matrix : \n', cm)
# From confusion matrix, calculate accuracy
```

```

total = sum(sum(cm))

accuracy = (cm[0,0]+cm[1,1])/total
print ('Accuracy : ', accuracy)

sensitivity = cm[1,1]/(cm[1,0]+cm[1,1])
print('Sensitivity : ', sensitivity )

specificity = cm[0,0]/(cm[0,0]+cm[0,1])
print('Specificity : ', specificity)

```

Confusion Matrix :

```

[[45  1]
 [ 1 73]]

```

Accuracy : 0.9833333333333333

Sensitivity : 0.9864864864864865

Specificity : 0.9782608695652174

```

print('The Root Mean Squared Error is: \n',
      np.sqrt(metrics.mean_squared_error(df.y_test, df.y_test_pred)))

```

The Root Mean Squared Error is:

0.12909944487358055

For KNN

```

x = data_imputed_df.drop(['class_ckd'], axis = 1)
#using most optimal features
x_knn = x[['sg', 'al', 'su', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wbcc', 'appet_poor']]
y = data_imputed_df['class_ckd']

```

```

np.random.seed(1)

x_train, x_test, y_train, y_test = train_test_split(
    x_knn,
    y,
    test_size=0.30,
    random_state=0,
    stratify=y
)

```

Now, find the optimal k for knn classifier.

```

k_range = range(1, 15)
scores = []

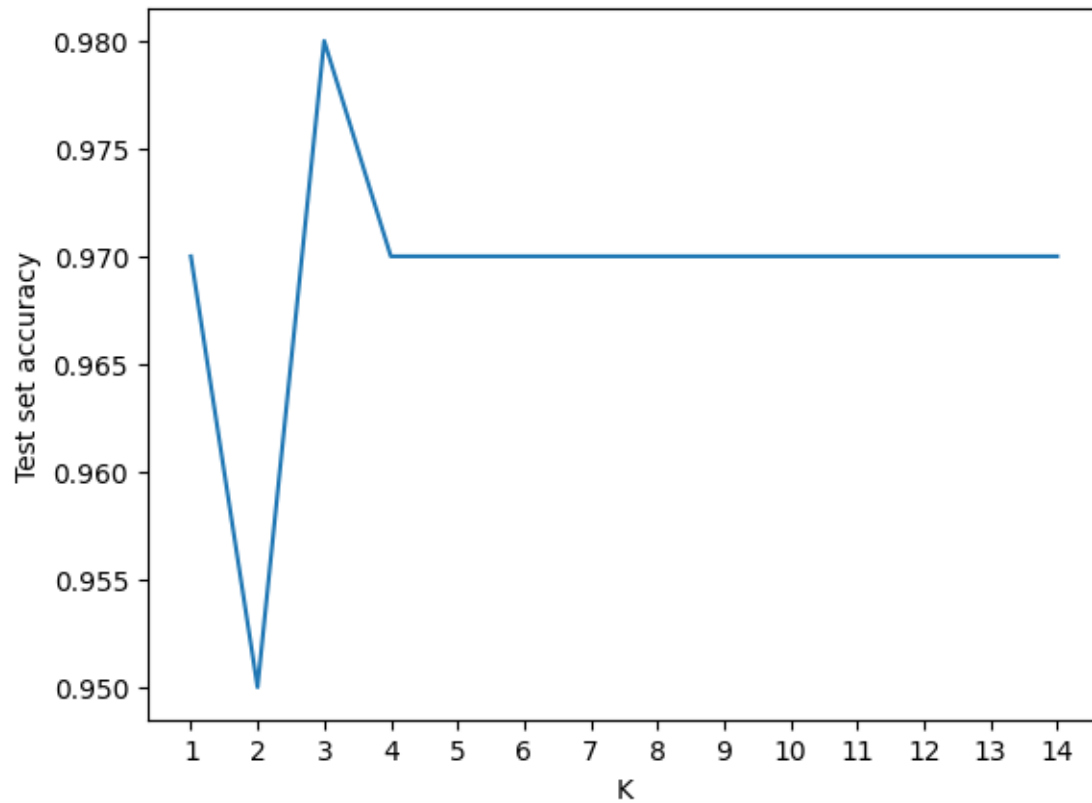
for k in k_range:
    knn = neighbors.KNeighborsClassifier(n_neighbors=k)
    knn.fit(x_train, y_train)
    y_pred = knn.predict(x_test)
    scores.append(round(metrics.accuracy_score(y_test, y_pred),2))

```

```

plt.plot(k_range, scores)
plt.xlabel('K')
plt.ylabel('Test set accuracy')
plt.xticks(range(1,15))
plt.show()

```



```
# Let K = 3
knn3 = neighbors.KNeighborsClassifier(
    n_neighbors =3,
    algorithm='brute'
)
```

```
# train the model
knn3.fit(x_train, y_train)

# predict on the test set
pred3 = knn3.predict(x_test)
```

```
# accuracy on the test set
print(round(metrics.accuracy_score(y_test, pred3),2))
```

0.98

```

cm = confusion_matrix(y_test, pred3)
print('Confusion Matrix : \n', cm)

total = sum(sum(cm))

accuracy = (cm[0,0]+cm[1,1])/total
print ('Accuracy : ', accuracy)

sensitivity = cm[1,1]/(cm[1,0]+cm[1,1])
print('Sensitivity : ', sensitivity )

specificity = cm[0,0]/(cm[0,0]+cm[0,1])
print('Specificity : ', specificity)

```

Confusion Matrix :

```
[[45  1]
```

```
[ 1 73]]
```

Accuracy : 0.9833333333333333

Sensitivity : 0.9864864864864865

Specificity : 0.9782608695652174

```

print('The Root Mean Squared Error is: \n',
      np.sqrt(metrics.mean_squared_error(y_test, pred3)))

```

The Root Mean Squared Error is:

0.12909944487358055

So, we see after training the models on the dataset with their optimal number of features and optimizing the classification algorithms according to the KS curve (for logistic regression) and finding the optimal k (for knn), we find that the two models have the same test set accuracy, sensitivity, specificity, and RMSE.

However, it is important to note that the logistic regression classifier is able to produce the same results whilst needing 7 predictors, whereas the knn classifier requires 10 predictors to achieve the same level of accuracy on the test set.

13 Interpretable Classifier Insight

```
from patsy import dmatrices, dmatrix
```

```
X = dmatrix(  
    'bp + sg + al + su + bgr + hemo + appet_poor',  
    data=data_imputed_df,  
    return_type='dataframe'  
)
```

```
y = data_imputed_df['class_ckd']
```

```
model = sm.Logit(y, X).fit()
```

Optimization terminated successfully.

Current function value: 0.084023

Iterations 10

```
model.summary()
```

| | | | |
|-------------------------|------------------|--------------------------|-----------|
| Dep. Variable: | class_ckd | No. Observations: | 400 |
| Model: | Logit | Df Residuals: | 392 |
| Method: | MLE | Df Model: | 7 |
| Date: | Mon, 15 Apr 2024 | Pseudo R-squ.: | 0.8735 |
| Time: | 17:45:52 | Log-Likelihood: | -33.609 |
| converged: | True | LL-Null: | -265.63 |
| Covariance Type: | nonrobust | LLR p-value: | 4.300e-96 |

| | coef | std err | z | P> z | [0.025 | 0.975] |
|-------------------|---------|---------|--------|-------|--------|--------|
| Intercept | 3.4841 | 0.746 | 4.672 | 0.000 | 2.022 | 4.946 |
| bp | 0.6735 | 0.481 | 1.401 | 0.161 | -0.269 | 1.616 |
| sg | -2.2660 | 0.523 | -4.334 | 0.000 | -3.291 | -1.241 |
| al | 2.0627 | 0.719 | 2.869 | 0.004 | 0.654 | 3.472 |
| su | 0.9194 | 1.030 | 0.892 | 0.372 | -1.100 | 2.939 |
| bgr | 1.1258 | 0.888 | 1.268 | 0.205 | -0.614 | 2.866 |
| hemo | -3.8583 | 0.838 | -4.606 | 0.000 | -5.500 | -2.216 |
| appet_poor | 0.3482 | 1.051 | 0.331 | 0.740 | -1.711 | 2.407 |

Possibly complete quasi-separation: A fraction 0.30 of observations can be perfectly predicted. This might indicate that there is complete quasi-separation. In this case some parameters will not be identified.

After running our logistic regression classifier on all the data, using the most optimal features, we can create the above summary table. From this table, we see that the predictors that are significant (p-value < 0.05) are: specific gravity (sg), albumin (al), and hemoglobin in grams (hemo). The predictors that are no longer significant are: blood pressure in mm/Hg (bp), sugar (su), blood glucose random in mgs/dL (bgr), and poor appetite(appet_poor).

We can interpret the predictors as follows:

For blood pressure: a one unit (mm/Hg) increase in blood pressure, when all other predictors are held constant, results in a 0.6735 increase in the log odds of having chronic kidney disease.

For specific gravity: when all other predictors are held constant, the log odds of having chronic kidney disease decreases by a factor of 2.2660 times the individual's specific gravity at the time of the test.

For albumin: when all other predictors are held constant, the log odds of having chronic kidney disease increases by a factor of 2.0627 times the individual's level of albumin.

For sugar: when all other predictors are held constant, the log odds of having chronic kidney disease increases by a factor of 0.9194 times the individual's sugar level.

For blood glucose random: a one unit (mgs/dL) increase in blood glucose random, when all other predictors are held constant, results in a 1.1258 increase in the log odds of having chronic kidney disease.

For hemoglobin: a one unit (grams) increase in hemoglobin, when all other predictors are held constant, results in a 3.8583 decrease in the log odds of having chronic kidney disease.

For poor appetite: when all other predictors are held constant, individuals who have a poor appetite have a 0.3482 increase in the log odds of having chronic kidney disease.

```
x = data_imputed_df[['bp', 'sg', 'al', 'su', 'bgr', 'hemo', 'appet_poor']]
y = data_imputed_df['class_ckd']
```

```
ckd_log = LogisticRegression()
ckd_log.fit(x, y)
```

```
LogisticRegression()
```

```
pred_prob = ckd_log.predict_proba(x)
```

```
df = pd.DataFrame(
    data = {'prob1': pred_prob[:,1], 'y': y}
)
df.head()
```

| | prob1 | y |
|---|----------|---|
| 0 | 0.198839 | 1 |
| 1 | 0.993484 | 1 |
| 2 | 0.999999 | 1 |
| 3 | 0.999983 | 1 |
| 4 | 0.998362 | 1 |

```
df['y_pred'] = df.prob1.map(lambda x: 1 if x>=0.5119812831944971 else 0)
```

```
cm = confusion_matrix(df.y, df.y_pred)
print('Confusion Matrix : \n', cm)
# From confusion matrix, calculate accuracy

total = sum(sum(cm))

accuracy = (cm[0,0]+cm[1,1])/total
print ('Accuracy : ', accuracy)

sensitivity = cm[1,1]/(cm[1,0]+cm[1,1])
print('Sensitivity : ', sensitivity )

specificity = cm[0,0]/(cm[0,0]+cm[0,1])
print('Specificity : ', specificity)
```

Confusion Matrix :

```
[[150  2]
```

```
[ 4 244]]
```

Accuracy : 0.985

Sensitivity : 0.9838709677419355

Specificity : 0.9868421052631579

When we run all the data through our logistic regression classifier, we see we have an accuracy of 98.5%, indicating that this is a good model fit.

15 Team Contributions

Anu did questions 1-5

Woojoo did questions 6-9

Derek did questions 10-13

We also collaboratively helped one another when stuck at a part or to determine the best way to proceed.

16 Link

https://github.com/Anu-Kumar2/STATS3DA3_A6/tree/main