Final Project

Jenna Orvitz(400312612), Noah Ripstein (400311716), Viransh Shah (400394334)

2024-04-18

```
!pip3 install ucimlrepo
```

```
Requirement already satisfied: ucimlrepo in /Users/NoahRipstein/miniconda3/envs/3d_39/lib/pytho
```

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.cm as cm
from sklearn.impute import SimpleImputer
import seaborn as sns
from patsy import dmatrices, dmatrix
from sklearn.preprocessing import StandardScaler
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.decomposition import PCA, TruncatedSVD
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, classification_report, silhouette_samples, silhou
from sklearn.metrics.cluster import rand_score
import statsmodels.api as sm
```

```python
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs
from sklearn.neighbors import KNeighborsClassifier
```

```python
from ucimlrepo import fetch_ucirepo

chronic_kidney_disease = fetch_ucirepo(id=336)

df = pd.concat([chronic_kidney_disease.data.features, chronic_kidney_disease.data.targets], ax

df.head()
```

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv	wbcc	rbc
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44.0	7800.0	5.2
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38.0	6000.0	Na
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31.0	7500.0	Na
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...	32.0	6700.0	3.9
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...	35.0	7300.0	4.6

1. Classification Problem Identification: Define and describe a classification problem based on the dataset.

Our goal is to predict whether individuals have Chronic Kidney Disease (CKD) based on various medical predictor variables. This classification problem involves distinguishing between two categories: individuals diagnosed with CKD and those without. We will use a clinical dataset to build a predictive model that can accurately identify these two states based on patient medical data.

2. Variable Transformation: Implement any transformations chosen or justify the absence of such modifications.

```
df.describe()
```

	age	bp	sg	al	su	bgr	bu	sc
count	391.000000	388.000000	353.000000	354.000000	351.000000	356.000000	381.000000	383.000000
mean	51.483376	76.469072	1.017408	1.016949	0.450142	148.036517	57.425722	3.072454
std	17.169714	13.683637	0.005717	1.352679	1.099191	79.281714	50.503006	5.741126
min	2.000000	50.000000	1.005000	0.000000	0.000000	22.000000	1.500000	0.400000
25%	42.000000	70.000000	1.010000	0.000000	0.000000	99.000000	27.000000	0.900000
50%	55.000000	80.000000	1.020000	0.000000	0.000000	121.000000	42.000000	1.300000
75%	64.500000	80.000000	1.020000	2.000000	0.000000	163.000000	66.000000	2.800000
max	90.000000	180.000000	1.025000	5.000000	5.000000	490.000000	391.000000	76.000000

```
df.dtypes
```

```
age      float64
bp       float64
sg       float64
al       float64
su       float64
rbc       object
pc        object
pcc       object
ba        object
bgr      float64
bu       float64
sc       float64
sod      float64
pot      float64
hemo     float64
pcv      float64
wbcc     float64
```

```
rbcc     float64
htn       object
dm        object
cad       object
appet     object
pe        object
ane       object
class     object
dtype: object
```

```
float64_columns = df.select_dtypes(
    include=['float64']
    ).columns
float64_columns
scaler = StandardScaler()
df[float64_columns] = scaler.fit_transform(df[float64_columns])
```

```
cat_columns = df.select_dtypes(
    include=['object']
    ).columns

for col in cat_columns:
    print(df[col].value_counts(normalize=True))
```

```
rbc
normal      0.810484
abnormal    0.189516
Name: proportion, dtype: float64
pc
normal      0.773134
abnormal    0.226866
Name: proportion, dtype: float64
```

```
pcc
notpresent    0.893939
present       0.106061
Name: proportion, dtype: float64
ba
notpresent    0.944444
present       0.055556
Name: proportion, dtype: float64
htn
no     0.630653
yes    0.369347
Name: proportion, dtype: float64
dm
no      0.653266
yes     0.344221
\tno    0.002513
Name: proportion, dtype: float64
cad
no     0.914573
yes    0.085427
Name: proportion, dtype: float64
appet
good    0.794486
poor    0.205514
Name: proportion, dtype: float64
pe
no     0.809524
yes    0.190476
Name: proportion, dtype: float64
ane
no     0.849624
yes    0.150376
```

```
Name: proportion, dtype: float64
class
ckd       0.620
notckd    0.375
ckd\t     0.005
Name: proportion, dtype: float64
```

```
for col in cat_columns:
    df[col] = df[col].astype('category').cat.codes
df.head(5)
```

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv
0	-0.203139	0.258373	0.454071	-0.012548	-0.410106	-1	1	0	0	-0.341498	...	0.569881
1	-2.594124	-1.936857	0.454071	2.208413	-0.410106	-1	1	0	0	NaN	...	-0.098536
2	0.613295	0.258373	-1.297699	0.727772	2.323069	1	1	0	0	3.473064	...	-0.878356
3	-0.203139	-0.473370	-2.173584	2.208413	-0.410106	1	0	1	0	-0.392022	...	-0.766953
4	-0.028189	0.258373	-1.297699	0.727772	-0.410106	1	1	0	0	-0.530963	...	-0.432744

Here, we performed two data transformation steps: 1. Transformation 1: Standardizing Numerical Features.

This step Z-transformed the numerical features to make them come from a distribution closer to a standard normal distribution with mean 0 and variance 1. This can improve performance of some classification algorithms, including logistic regression.

2. Transformation 2: Encoding Categorical Features.

This step converted columns containing categorical features into categorical variables within pandas. This is needed so that when we use our pandas dataframe as input to our classification models later, the libraries recognize the variables as categorical, rather than continuous.

3. Dataset Overview: Provide a detailed description of the dataset, covering variables, summaries, observation counts, data types, and distributions (at least three statements).

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 25 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   age     391 non-null    float64
 1   bp      388 non-null    float64
 2   sg      353 non-null    float64
 3   al      354 non-null    float64
 4   su      351 non-null    float64
 5   rbc     400 non-null    int8
 6   pc      400 non-null    int8
 7   pcc     400 non-null    int8
 8   ba      400 non-null    int8
 9   bgr     356 non-null    float64
 10  bu      381 non-null    float64
 11  sc      383 non-null    float64
 12  sod     313 non-null    float64
 13  pot     312 non-null    float64
 14  hemo    348 non-null    float64
 15  pcv     329 non-null    float64
 16  wbcc    294 non-null    float64
 17  rbcc    269 non-null    float64
 18  htn     400 non-null    int8
 19  dm      400 non-null    int8
 20  cad     400 non-null    int8
 21  appet   400 non-null    int8
 22  pe      400 non-null    int8
 23  ane     400 non-null    int8
 24  class   400 non-null    int8
```

```
dtypes: float64(14), int8(11)
memory usage: 48.2 KB
```

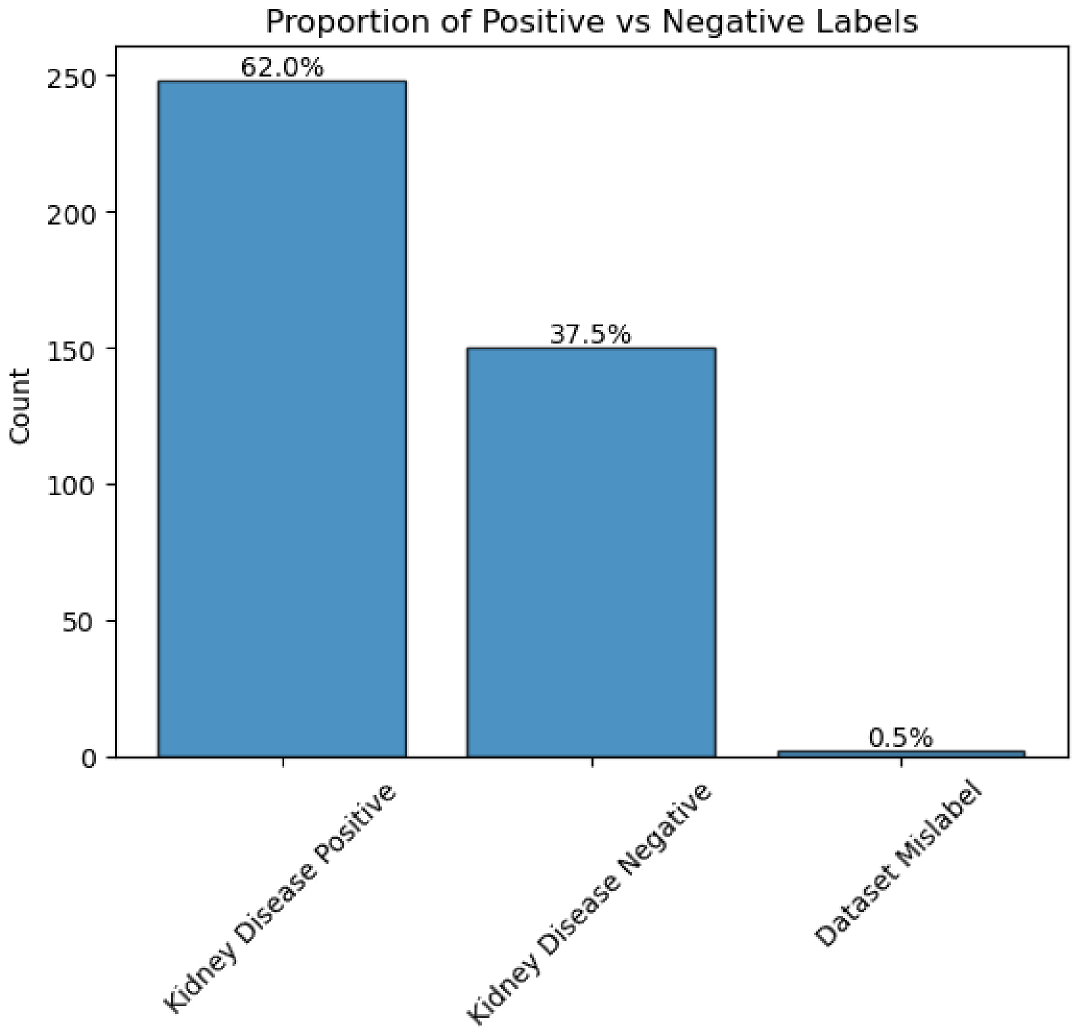
```
df.describe()
```

	age	bp	sg	al	su	rbc	pc	p
count	3.910000e+02	3.880000e+02	3.530000e+02	354.000000	351.000000	400.00000	400.000000	4
mean	9.994847e-17	-2.380684e-16	2.415443e-15	0.000000	0.000000	0.12250	0.485000	0.
std	1.001281e+00	1.001291e+00	1.001419e+00	1.001415	1.001428	0.93256	0.759089	0.
min	-2.885708e+00	-1.936857e+00	-2.173584e+00	-0.752868	-0.410106	-1.00000	-1.000000	-
25%	-5.530393e-01	-4.733701e-01	-1.297699e+00	-0.752868	-0.410106	-1.00000	0.000000	0.
50%	2.050779e-01	2.583733e-01	4.540705e-01	-0.752868	-0.410106	1.00000	1.000000	0.
75%	7.590867e-01	2.583733e-01	4.540705e-01	0.727772	-0.410106	1.00000	1.000000	0.
max	2.246163e+00	7.575807e+00	1.329955e+00	2.948733	4.145186	1.00000	1.000000	1.

```
df["class"].value_counts()
```

```
class
0    248
2    150
1      2
Name: count, dtype: int64
```

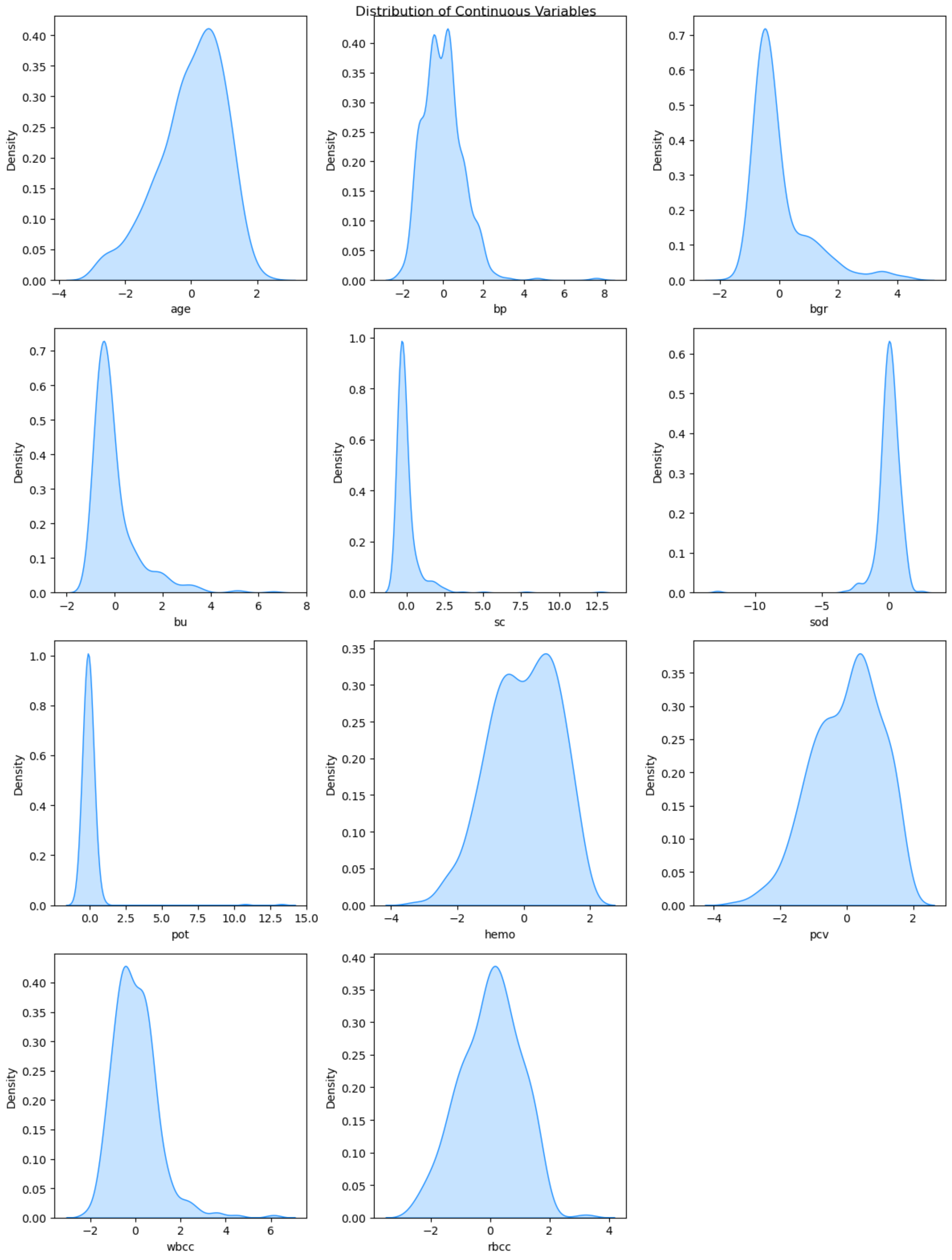
```
fig, ax = plt.subplots(1, 1)
bar_data = df["class"].value_counts()
ax.bar(range(len(bar_data)), bar_data, edgecolor="black", alpha=0.8)

ax.set_xticks([0, 1, 2])
ax.set_xticklabels(["Kidney Disease Positive", "Kidney Disease Negative", "Dataset Mislabel"],

for i, count in enumerate(bar_data):
    percentage = count / bar_data.sum() * 100
    ax.text(i, count, f"{percentage:.1f}%", ha="center", va="bottom")
ax.set_ylabel("Count")
ax.set_title("Proportion of Positive vs Negative Labels")
plt.show()
```



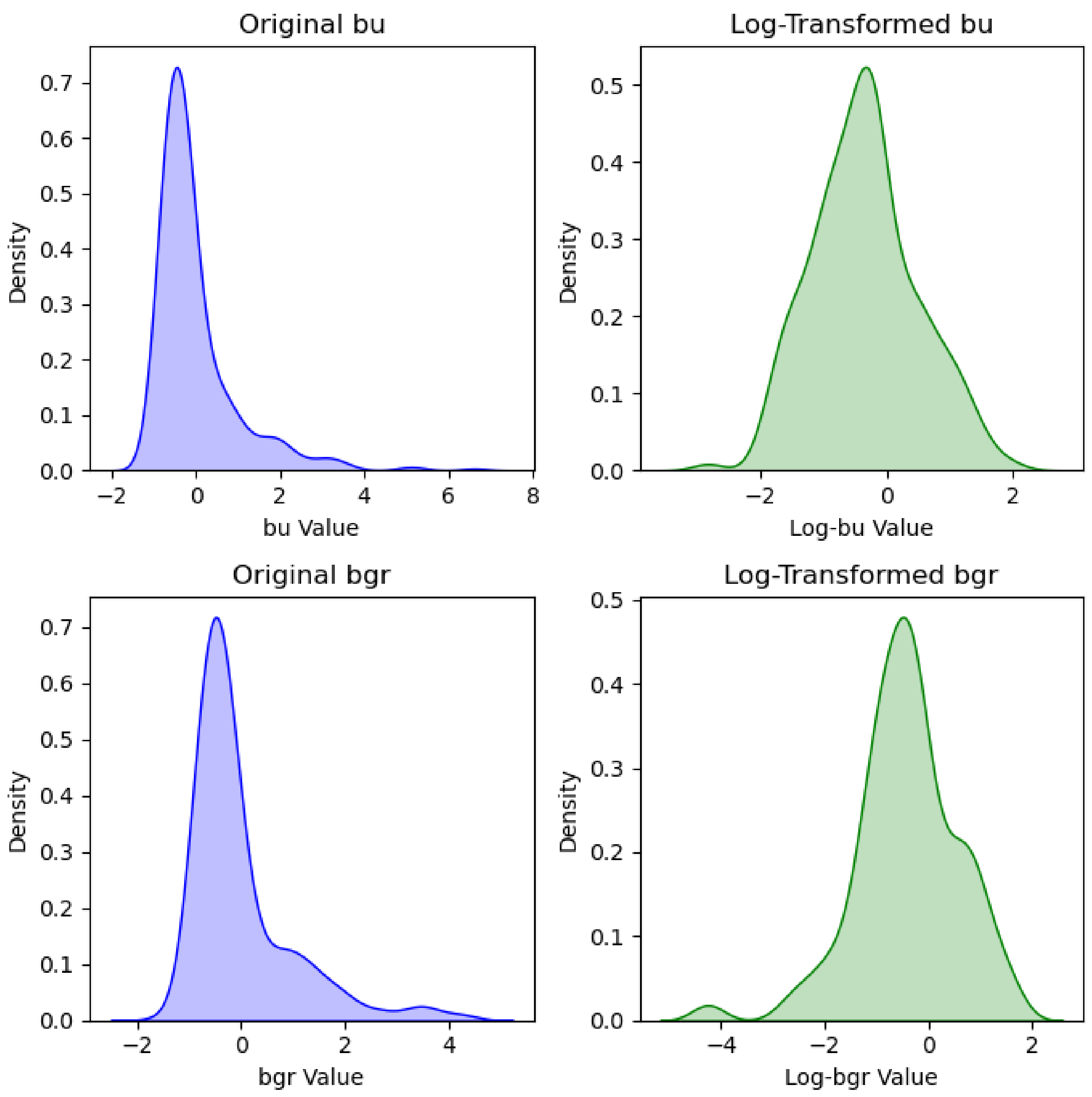
Visualizing distribution of continuous variables with Kernel Density Estimation

```
num_vars = ['age', 'bp', 'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wbcc', 'rbcc']


num_features = len(num_vars)
num_rows = 4  # Number of rows in the subplot grid
```

```python
num_cols = 3  # Number of columns in the subplot grid

fig, axes = plt.subplots(num_rows, num_cols, figsize=(4 * num_cols, 4 * num_rows))

for i, cont_feature in enumerate(df[num_vars]):
    row = i // num_cols  # Calculate the row index for the subplot
    col = i % num_cols  # Calculate the column index for the subplot

    ax_kde = axes[row, col]

    # Plot KDE for the feature
    sns.kdeplot(df[cont_feature], ax=ax_kde, fill=True, color="dodgerblue")

# Remove empty subplots
for i in range(num_features, num_rows * num_cols):
    fig.delaxes(axes.flatten()[i])

plt.suptitle("Distribution of Continuous Variables")
plt.tight_layout()
plt.show()
```


Distribution of Continuous Variables
Density
age
bp
bgr
bu
sc
sod
pot
hemo
pcv
wbcc
rbcc

```python
# Create a new DataFrame with selected variables and their transformations
data_log_vis = pd.DataFrame({
    'bu': df['bu'],
    'log_bu': np.log(df['bu'] + 1),  # Log transform with handling zero values
    'bgr': df['bgr'],
    'log_bgr': np.log(df['bgr'] + 1)
})

# Variables to plot
variables = ['bu', 'bgr']

# Create a figure with 2 rows and 2 columns
fig, axes = plt.subplots(2, 2, figsize=(7, 7))
axes = axes.flatten()  # Flatten to simplify indexing

for i, var in enumerate(variables):
    # Original Data Plot
    sns.kdeplot(data_log_vis[var], ax=axes[2*i], fill=True, color="blue")
    axes[2*i].set_title(f"Original {var}")
    axes[2*i].set_xlabel(f"{var} Value")
    axes[2*i].set_ylabel("Density")

    # Log-Transformed Data Plot
    sns.kdeplot(data_log_vis[f'log_{var}'], ax=axes[2*i+1], fill=True, color="green")
    axes[2*i+1].set_title(f"Log-Transformed {var}")
    axes[2*i+1].set_xlabel(f"Log-{var} Value")
    axes[2*i+1].set_ylabel("Density")

plt.tight_layout()
plt.show()
```



Observations: 1. The dataset has an imbalance in the number of kidney disease positive vs negative examples. Our visual exploratory data analysis also revealed that there are two mislabeled variables in the dataset's target column. The column in the dataset should include only "positive" or "negative" Kidney disease status, but there were a few examples with a third label. We discuss this more in the outliers section. 2. Many of the variables look roughly noramlly distributed, except that the blood glucode random and blood urea features are long-tailed. This has implications for feature engineering: we expect that log-transforming these features will make them closer to a normal distribution; this is likely to improve performance on classifiers such as logistic regression. We visualized these variables log-transformed to confirm that they look closer to a normal distribution after the transformation 3. Most variables are continuous, although the specific gravity, albumin and sugar levels are categorical.

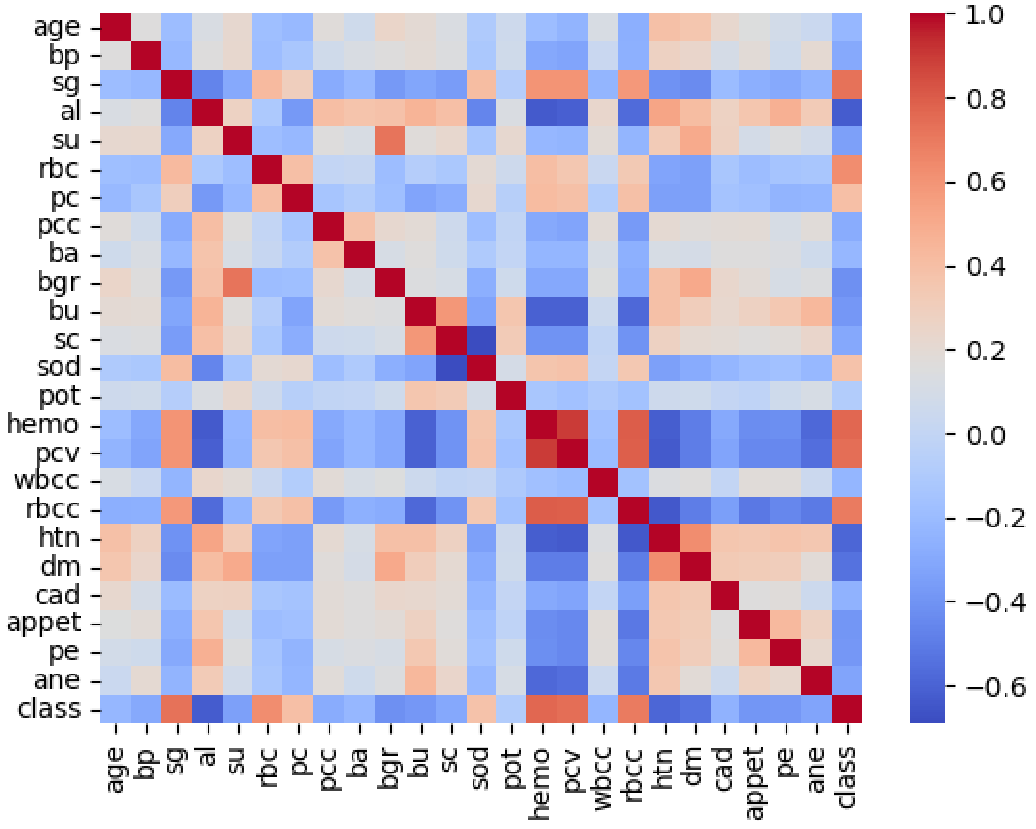
In this stage of exploratory data analysis, we are unsure whether we will end up using the log-transformed variables. This will depend on our results we obtain later.

4. Association Between Variables: Analyze variable relationships and their implications for feature selection or extraction (at least three statements)

```
correlation = df.corr()
sns.heatmap(correlation, cmap='coolwarm')


correlation
```

	age	bp	sg	al	su	rbc	pc	pcc	ba
age	1.000000	0.159480	-0.191096	0.122091	0.220866	-0.181683	-0.209743	0.169865	0.065425
bp	0.159480	1.000000	-0.218836	0.160689	0.222576	-0.194643	-0.129873	0.074018	0.126518
sg	-0.191096	-0.218836	1.000000	-0.469760	-0.296234	0.421101	0.299093	-0.290210	-0.220317
al	0.122091	0.160689	-0.469760	1.000000	0.269305	-0.110803	-0.375461	0.403257	0.366845
su	0.220866	0.222576	-0.296234	0.269305	1.000000	-0.187230	-0.221037	0.156997	0.115534
rbc	-0.181683	-0.194643	0.421101	-0.110803	-0.187230	1.000000	0.393821	0.002845	0.019199
pc	-0.209743	-0.129873	0.299093	-0.375461	-0.221037	0.393821	1.000000	-0.136040	-0.088435
pcc	0.169865	0.074018	-0.290210	0.403257	0.156997	0.002845	-0.136040	1.000000	0.376102
ba	0.065425	0.126518	-0.220317	0.366845	0.115534	0.019199	-0.088435	0.376102	1.000000
bgr	0.244992	0.160193	-0.374710	0.379464	0.717827	-0.193079	-0.175899	0.215386	0.109492
bu	0.196985	0.188517	-0.314295	0.453528	0.168583	-0.071404	-0.323372	0.192276	0.167696
sc	0.132531	0.146222	-0.361473	0.399198	0.223244	-0.122191	-0.279445	0.060680	0.063784
sod	-0.100046	-0.116422	0.412190	-0.459896	-0.131776	0.197653	0.218343	-0.183387	-0.100474
pot	0.058377	0.075151	-0.072787	0.129038	0.219450	0.061364	-0.058745	-0.003962	0.001224
hemo	-0.192928	-0.306540	0.602582	-0.634632	-0.224775	0.402049	0.418814	-0.295985	-0.233115
pcv	-0.242119	-0.326319	0.603560	-0.611891	-0.239189	0.350038	0.391230	-0.326328	-0.230173
wbcc	0.118339	0.029753	-0.236215	0.231989	0.184893	0.029804	-0.079035	0.184171	0.115111

	age	bp	sg	al	su	rbc	pc	pcc	ba
rbcc	-0.268896	-0.261936	0.579476	-0.566437	-0.237448	0.339400	0.390282	-0.371968	-0.266713
htn	0.389724	0.277324	-0.410243	0.525234	0.321166	-0.321229	-0.344689	0.206843	0.111083
dm	0.354065	0.235513	-0.436692	0.406456	0.500133	-0.345661	-0.345482	0.173907	0.099610
cad	0.221807	0.098398	-0.195717	0.272713	0.276542	-0.129224	-0.154193	0.184861	0.157115
appet	0.148648	0.184732	-0.268856	0.359009	0.089770	-0.190258	-0.172015	0.193949	0.155157
pe	0.085726	0.062676	-0.298504	0.477127	0.144712	-0.143371	-0.244199	0.113742	0.141271
ane	0.041271	0.204279	-0.243082	0.322958	0.077908	-0.135308	-0.233601	0.178299	0.064608
class	-0.222361	-0.297019	0.729117	-0.625585	-0.345589	0.630148	0.397401	-0.283455	-0.222438



Observations: 1. Correlations between the following: White bloodcell count-packed cell volume (Hemo and pcv features), red bloodcell count-hemoglobin (hemo and rbcc features), Packed cell volume-red blood cell count (pcv and rbcc features) have the three highest positive correlations. 2. Correlations between the following: Serum creatinine and sodium (sc and sod features), Hemoglobin and hypertension (hemo and htn features), Packed cell volume and hypertension (pcv and htn features), Hemoglobin and anemia (hemo and ane features), packed cell volume and anemia (pcv and ane features) have the highest negative correlations. 3. Highly correlated features can lead to overfitting or redundant information. We can get rid of redundant features which leads to simpler models. 4. The packed cell volume feature is highly correlated with many variables. This means that on it's own, it adds very little information. For that reason, we opted to remove that feature. We did some initial testing to verify that it did not degrade performance of our classifiers, and found that it had no impact on performance. Including it, then, would simply add what amounts to random noise, making the model less robust for future predictions.

```
df = df.drop(["pcv"], axis=1)
```

5. Missing Value Analysis and Handling: Implement your strategy for identifying and addressing missing values in the dataset, or provide reasons for not addressing them.

```
# Missing Value Analysis
missing_values = df.isnull().sum()


print(missing_values)
```

```
age         9
bp         12
sg         47
al         46
su         49
rbc         0
pc          0
pcc         0
ba          0
bgr        44
bu         19
sc         17
sod        87
pot        88
```

```
hemo      52
wbcc     106
rbcc     131
htn        0
dm         0
cad        0
appet      0
pe         0
ane        0
class      0
dtype: int64
```

```
# Mean imputer for numerical values and most frequent imputer for categorical values
num_vars = ['age', 'bp', 'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wbcc', 'rbcc']
num_vars = ['age', 'bp', 'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo', 'wbcc', 'rbcc']
cat_vars = ['sg', 'al', 'su']

imputer_num = SimpleImputer(strategy='mean')
imputer_cat = SimpleImputer(strategy='most_frequent')

df[num_vars] = imputer_num.fit_transform(df[num_vars])
df[cat_vars] = imputer_cat.fit_transform(df[cat_vars])
```

- For numerical features (age, blood pressure, blood glucose random, blood urea, serum creatinine, sodium, potassium, hemoglobin, packed cell volume, white blood cell count, red blood cell count), we'll use mean imputation.
- For categorical features (specific gravity, albumin, sugar), we'll use mode imputation.
- Binary features (red blood cells, pus cell, pus cell clumps, bacteria, hypertension, diabetes mellitus, coronary artery disease, appetite, pedal edema, anemia) already have no missing values.

6. Outlier Analysis: Implement your approach for identifying and managing outliers, or provide reasons for not addressing them.

```
display(df["dm"].value_counts())
display(df["class"].value_counts())
```

```
dm
 1    260
 2    137
-1      2
 0      1
Name: count, dtype: int64
```

```
class
0    248
2    150
1      2
Name: count, dtype: int64
```

```
# I noticed dm has 1s and 2s, so I converted them to 0s and 1s
# Class has 0s and 2s, so I converted them to 0s and 1s

df['dm'] = df['dm'].replace({2:1, 1:0, -1:0})
# df['dm'] = df['dm'].replace({'2':1, '1':0,})
df['class'] = df['class'].replace({2:1})
```

```
display(df["dm"].value_counts())
display(df["class"].value_counts())
```

```
dm
0    263
1    137
Name: count, dtype: int64
```

```
class
```

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

We found two errors in the dataset: features which are reported as binary in the data card which have more than two values in the dataset.

We found that for both the target variable and diabetes mellitus, which were each supposed to be binary variables, there were a few examples of additional categories. These additional categories were merged into the most frequent category.

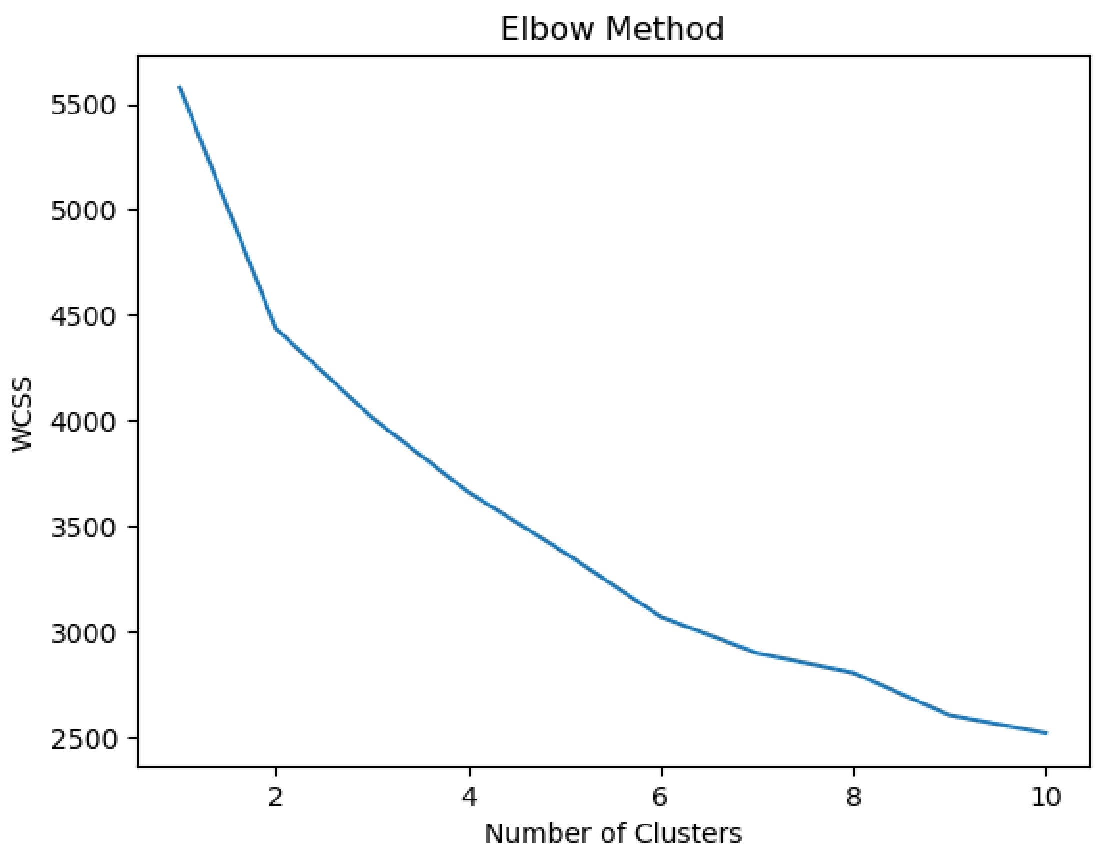
We ran our classification algorithms without any in-depth analysis regarding removal of outliers. Before doing so, we decided that if our classification performance is strong (accuracy above 90%), then we would likely not include any outlier analysis. We did find strong performance, and below is our justification for not removing outliers.

Since both our Random Forest and Logistic Regression classifiers are achieving extremley high accuracy, sensitivity, and specificity without any loss of performance due to the presence of outliers, we believe it's best to keep the outliers in the dataset. Removing them might cause me to lose valuable information and reduce the variability of the data, potentially affecting my ability to generalize to unseen data. Additionally, as Random Forest and Logistic Regression models are quite robust to outliers, we don't think their presence would significantly impact my performance. Also, we are cautious about making decisions regarding outlier removal to avoid introducing bias into the dataset and ensuring that any data manipulation doesn't compromise the integrity of my analysis results. Overall, if removing outliers doesn't lead to noticeable improvements in my performance, we would prefer to stick with the original dataset and keep the outliers intact.

7. Sub-group Analysis: Explore potential sub-groups within the data, employing appropriate data science methods to find the sub-groups of patients and visualize the sub-groups. The sub-group analysis must not include the labels (for CKD patients and healthy controls).

```python
# Split data into features and target variable
X = df.drop('class', axis=1)
y = df['class']
```

```python
# Determine the optimal number of clusters using the elbow method
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)

# Plot the elbow method graph to find the optimal number of clusters
plt.plot(range(1, 11), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.show()

# Based on the elbow method, choose the number of clusters
num_clusters = 2  # Adjust as needed

kmeans = KMeans(n_clusters=num_clusters, n_init=20, random_state=0)
kmeans.fit(X)
silhouette_avg = silhouette_score(X, kmeans.labels_)
print("Average Silhouette Score:", silhouette_avg)
```

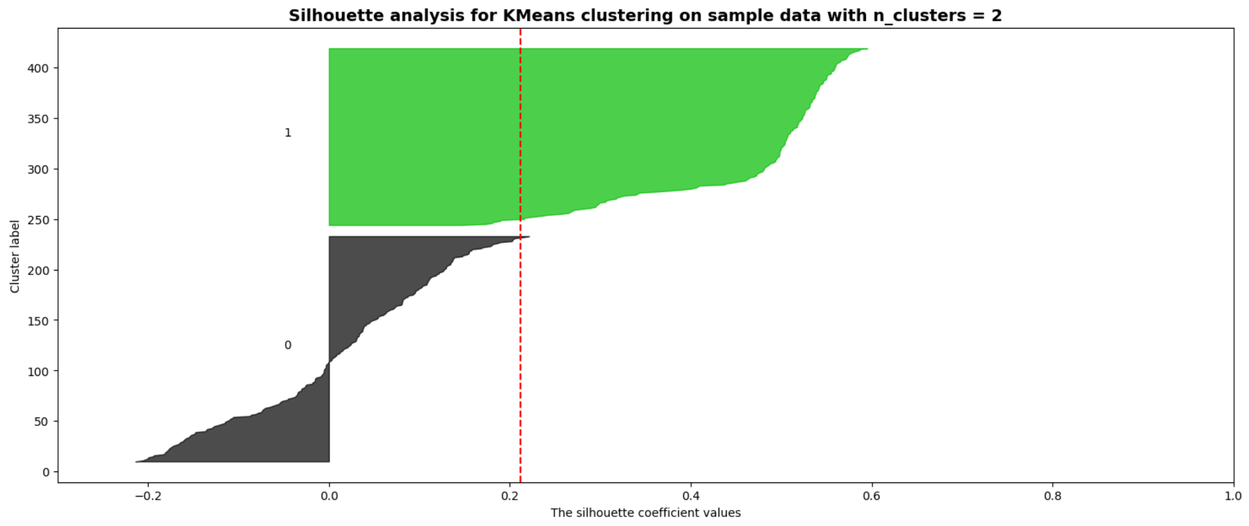


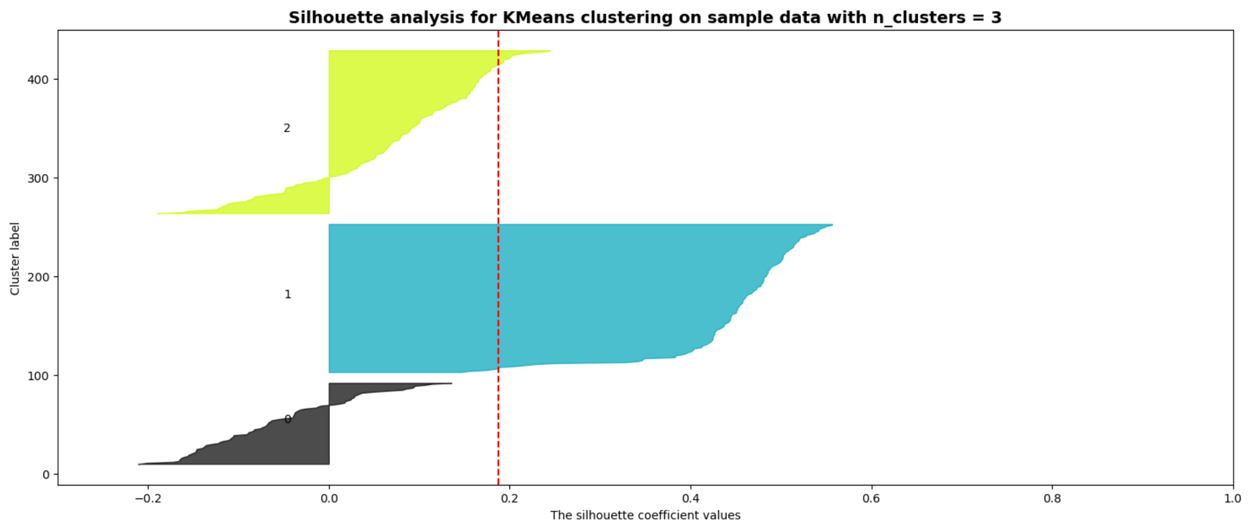
```
Average Silhouette Score: 0.21162760671350317
```

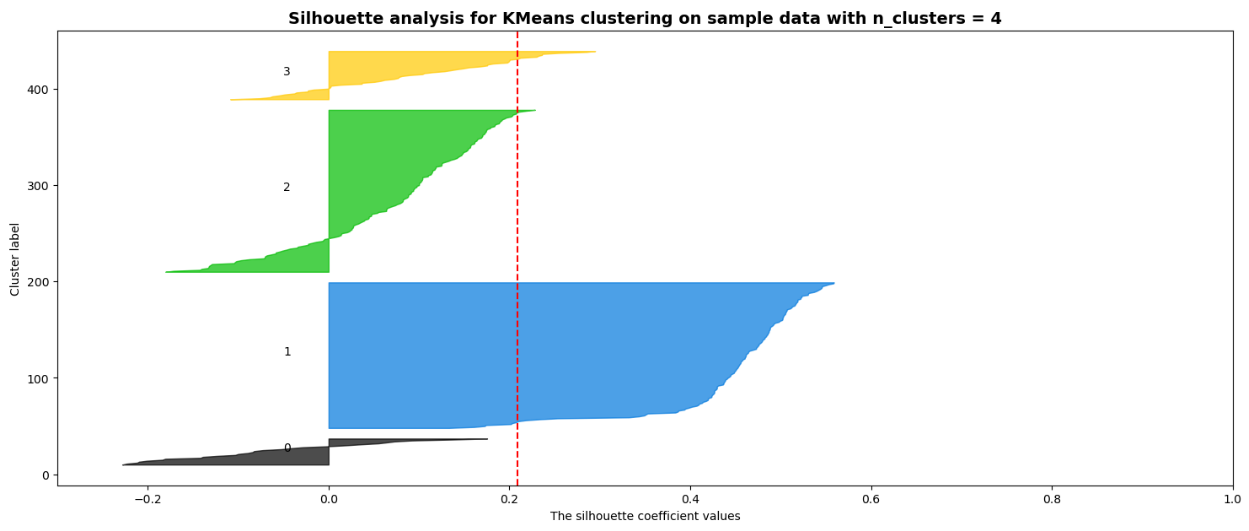
```
# Determine optimal number of clusters using silhouette scores
range_n_clusters = [2, 3, 4]
optimal_k = (0,0)
for n_clusters in range_n_clusters:
    km = KMeans(n_clusters = n_clusters, n_init = 20, random_state=0)
    cluster_labels_km = km.fit_predict(X)
    # average silhouette score
    silhouette_avg_km = silhouette_score(X, cluster_labels_km)
    # compute the silhouette scores for each sample
    sample_silhouette_values = silhouette_samples(X, cluster_labels_km)
    fig, ax1 = plt.subplots(1, 1)
    fig.set_size_inches(18, 7)
    ax1.set_xlim([-0.3, 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(
        y=np.arange(y_lower, y_upper),
        x1=0,
        x2=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 cluster silhouette scores
    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")
```

```
    # 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",
    )
    optimal_k = (n_clusters,silhouette_avg_km) if optimal_k[1] < silhouette_avg_km else optimal

plt.show()
print("Optimal number of clusters:", optimal_k[0])
```







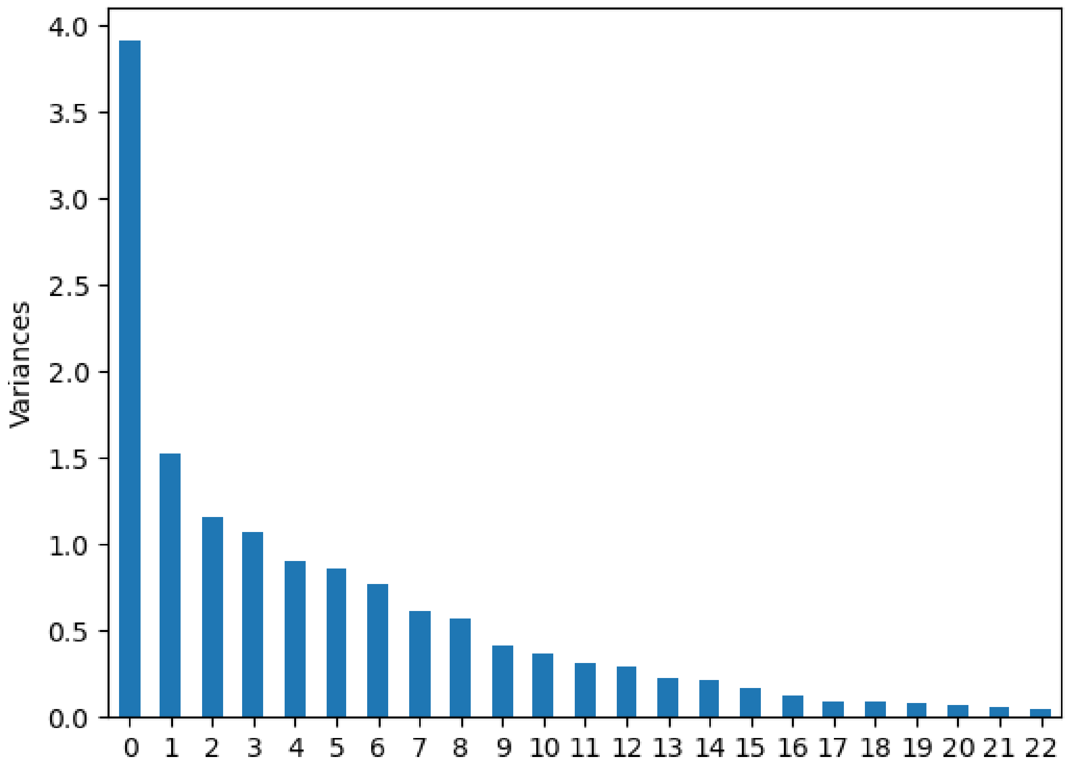
```
Optimal number of clusters: 2
```

```
# Apply k-means clustering with optimal k
kmeans_optimal = KMeans(n_clusters=optimal_k[0], n_init=20, random_state=0)
kmeans_optimal.fit(X)
cluster_counts = pd.Series(kmeans_optimal.labels_).value_counts().sort_index()
print("Number of observations within each cluster:")
print(cluster_counts)


# Perform PCA
pca = PCA()
```

```
df2_plot = pd.DataFrame(pca.fit_transform(X))
print("Variances: ",df2_plot.iloc[:,:].std(axis=0, ddof=0).to_numpy())


df2_plot.iloc[:,:].var(axis=0, ddof=0).plot(kind='bar', rot=0)
plt.ylabel('Variances')
```

```
Number of observations within each cluster:
0    224
1    176
Name: count, dtype: int64
Variances:  [1.97824553 1.23616004 1.07685887 1.03490043 0.9498406  0.92415524
 0.88020906 0.78520753 0.75648553 0.64444749 0.60909821 0.56257017
 0.54068757 0.47359118 0.45735511 0.40608239 0.35604038 0.30341904
 0.2989895  0.28835064 0.26487148 0.23276191 0.20995873]
```

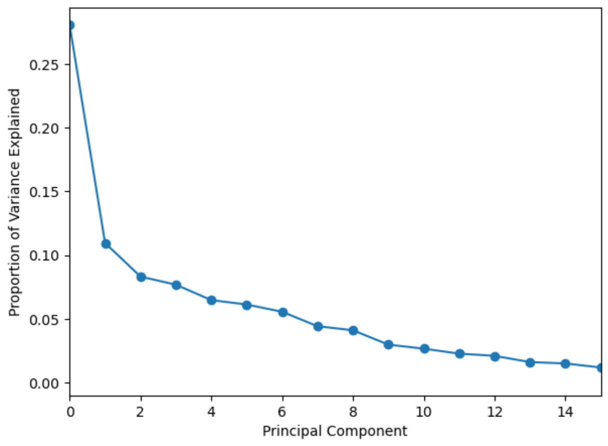
```
Text(0, 0.5, 'Variances')
```

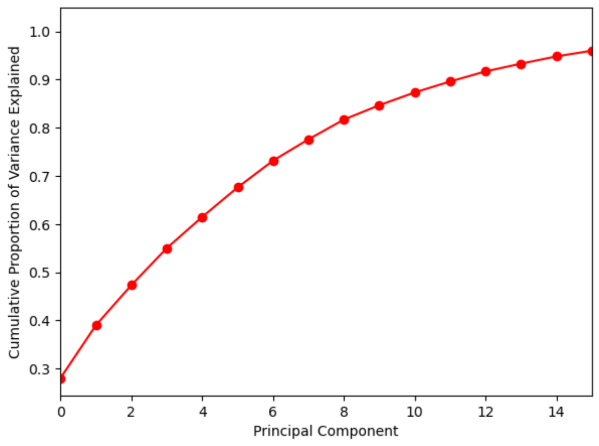


```
pd.DataFrame([df2_plot.iloc[:,:].std(axis=0, ddof=0).to_numpy(),
              pca.explained_variance_ratio_[:],
              np.cumsum(pca.explained_variance_ratio_[:])],
             index=['Standard Deviation', 'Proportion of Variance', 'Cumulative Proportion'],
             columns=['PC'+str(i) for i in range(1,24)])
```

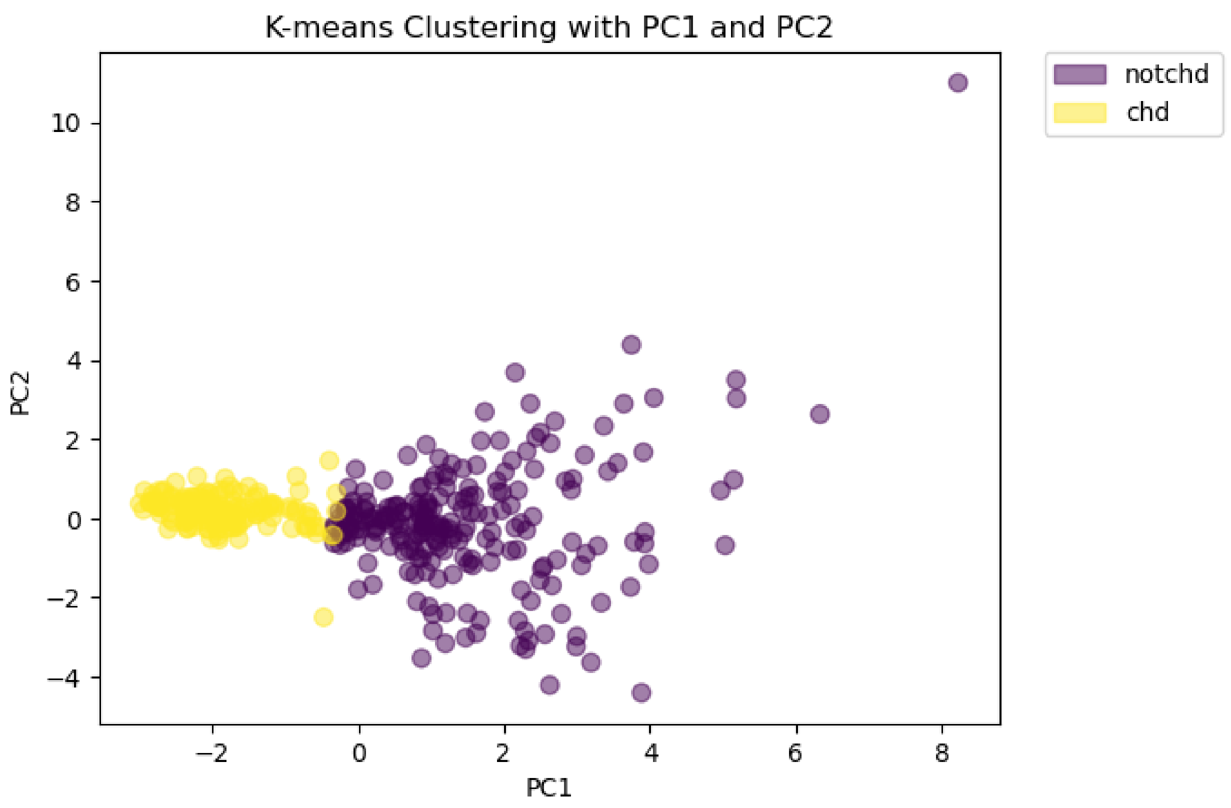
	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
Standard Deviation	1.978246	1.236160	1.076859	1.034900	0.949841	0.924155	0.880209	0.785208
Proportion of Variance	0.280628	0.109577	0.083155	0.076801	0.064695	0.061244	0.055558	0.044212
Cumulative Proportion	0.280628	0.390206	0.473361	0.550162	0.614857	0.676101	0.731658	0.775870

```
fig , (ax1,ax2) = plt.subplots(1,2, figsize=(15,5))

# Left plot
ax1.plot(pca.explained_variance_ratio_, '-o')
ax1.set_ylabel('Proportion of Variance Explained')
ax1.set_ylim(ymin=-0.01)

# Right plot
ax2.plot(np.cumsum(pca.explained_variance_ratio_), '-ro')
ax2.set_ylabel('Cumulative Proportion of Variance Explained')
ax2.set_ylim(ymax=1.05)

for ax in fig.axes:
    ax.set_xlabel('Principal Component')
    ax.set_xlim(0,15)
```





```
# Visualization of k-means cluster assignments using first two principal components
cmap = plt.cm.viridis

plt.scatter(df2_plot.iloc[:, 0], df2_plot.iloc[:, 1], c=kmeans_optimal.labels_, cmap=cmap, alpl
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('K-means Clustering with PC1 and PC2')
handles = []
labels = pd.factorize(y.unique())
norm = mpl.colors.Normalize(vmin=0.0, vmax=1.0)

for i, v in zip(labels[0], labels[1]):
    handles.append(mpl.patches.Patch(color=cmap(norm(i)), label='chd' if v else 'notchd', alpha

plt.legend(handles=handles, bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

plt.show()
```



```
loadings = pd.DataFrame(pca.components_.T, index=['PC'+str(i) for i in range(1,24)], columns=X
loadings
```

	age	bp	sg	al	su	rbc	pc	pcc	ba
PC1	0.187481	-0.142848	-0.332067	0.342348	-0.028598	-0.715559	-0.219458	-0.331568	0.109907
PC2	0.191835	-0.057524	-0.263973	0.177803	-0.525457	0.477178	-0.572327	-0.009737	0.045376
PC3	-0.305444	0.194334	-0.061244	0.324916	-0.001024	-0.046148	-0.122113	0.226060	-0.616412
PC4	0.301262	-0.142218	0.419900	-0.296534	-0.048340	-0.035415	-0.086280	-0.301424	-0.160720
PC5	0.213803	-0.459014	0.169075	0.371432	0.167873	0.190477	0.047790	0.135095	0.025224
PC6	-0.191137	0.055962	0.597954	0.125475	-0.031671	-0.155125	-0.416759	-0.115325	-0.207036
PC7	-0.172196	-0.094542	0.256790	-0.023357	-0.006085	0.093823	-0.200085	-0.306737	0.306594
PC8	0.059672	-0.035402	0.046848	-0.049790	0.000174	-0.045076	-0.035190	-0.046063	-0.030545
PC9	0.034915	-0.014265	0.036999	-0.035364	-0.017840	-0.005851	-0.031697	-0.030322	-0.017143
PC10	0.243771	-0.441432	0.049703	0.239993	0.296319	0.158132	0.091798	0.026811	-0.281345
PC11	0.331909	0.350027	0.212712	0.168158	-0.175769	-0.133963	0.065876	0.102201	-0.165886
PC12	0.276794	0.468200	0.062802	0.256852	0.323272	0.086364	-0.127712	0.118008	0.228832

	age	bp	sg	al	su	rbc	pc	pcc	ba
PC13	-0.224488	-0.290847	0.044785	0.015266	-0.518689	-0.168231	0.238112	0.132806	-0.097497
PC14	0.096974	0.099964	0.312560	0.418159	-0.381446	-0.015415	0.408705	0.071475	0.324271
PC15	-0.379467	-0.085367	0.057243	0.263912	0.169933	-0.005894	-0.104212	-0.031622	0.206046
PC16	0.091764	-0.195640	0.128557	-0.230602	0.019990	-0.307274	-0.339152	0.751861	0.243807
PC17	-0.300522	-0.069126	0.005050	0.188518	0.120493	0.063612	-0.043900	0.018577	0.110443
PC18	0.170662	-0.025807	-0.073268	-0.008285	-0.014891	-0.063309	-0.017943	-0.045158	-0.134117
PC19	0.154400	-0.092539	-0.076945	0.045352	0.043479	-0.031860	0.027395	0.025995	-0.089955
PC20	0.054064	-0.007259	-0.011827	0.016842	0.027190	-0.017680	-0.007664	-0.035311	-0.049205
PC21	0.089668	0.000769	-0.017763	-0.070699	-0.038262	-0.026814	-0.029931	0.006906	-0.082575
PC22	0.087658	0.013662	0.037247	-0.074564	-0.028892	-0.042924	0.027939	0.012466	-0.080151
PC23	0.083520	0.055551	0.004774	-0.037824	-0.067158	0.018269	0.015566	0.026223	-0.108663

```python
# Identify variable with most significant influence on all PCs
sub_groups = set()
for i in range(0,23):
    most_influential_variable = loadings.iloc[i,:].idxmax()
    print("Variable with most significant influence " + '(PC'+str(i+1)+'):', most_influential_
    sub_groups.add(most_influential_variable)

print("\nSub Groups: ",sub_groups)
```

```
Variable with most significant influence (PC1): al
Variable with most significant influence (PC2): rbc
Variable with most significant influence (PC3): bu
Variable with most significant influence (PC4): bgr
Variable with most significant influence (PC5): al
Variable with most significant influence (PC6): sg
Variable with most significant influence (PC7): ba
Variable with most significant influence (PC8): htn
Variable with most significant influence (PC9): ane
Variable with most significant influence (PC10): sc

Variable with most significant influence (PC11): bp

Variable with most significant influence (PC12): hemo

Variable with most significant influence (PC13): hemo

Variable with most significant influence (PC14): bu

Variable with most significant influence (PC15): pot

Variable with most significant influence (PC16): pcc

Variable with most significant influence (PC17): bgr

Variable with most significant influence (PC18): wbcc

Variable with most significant influence (PC19): htn

Variable with most significant influence (PC20): pe

Variable with most significant influence (PC21): rbcc

Variable with most significant influence (PC22): wbcc

Variable with most significant influence (PC23): pe


Sub Groups:  {'rbc', 'bgr', 'bu', 'al', 'bp', 'pcc', 'hemo', 'pe', 'ane', 'sc', 'rbcc', 'sg',
```

- Variable Influence on Principal Components: After performing PCA, each principal component represents a linear combination of the original variables. The loadings of the original variables on each principal component indicate their influence on that component. The code iterates through each principal component and identifies the variable with the highest loading for that component. This variable is considered to have the most significant influence on that principal component.
- Identification of Subgroups: By identifying the variable with the highest loading for each principal component, we essentially identify the key features that contribute the most to the variance captured by each component. These identified variables serve as the subgroups. Each subgroup represents a set of variables that are most influential in defining the variance along a particular principal component axis.
- Interpretation of Subgroups: The identified subgroups provide insights into the underlying structure of the data. They represent the dimensions along which the data vary the most. For example, if 'al' (albumin) is identified as the most influential variable for a principal component, it suggests that variations in albumin levels contribute significantly to the variance captured by that component. These subgroups help in understanding the key factors driving

the patterns observed in the data and aid in interpretation. For example, variables like 'al' (albumin) and 'rbc' (red blood cell count) are highlighted as influential across multiple principal components, suggesting their importance in distinguishing different clusters.

```
# Compare true labels with k-means cluster assignments
adjusted_rand_index = round(adjusted_rand_score(y, kmeans_optimal.labels_), 2)
rand = rand_score(kmeans_optimal.labels_, y).round(2)
print("Rand Index:", rand)
print("Adjusted Rand Index:", adjusted_rand_index)
```

```
Rand Index: 0.87
Adjusted Rand Index: 0.74
```

Adjusted Rand Index/Rand Index: The comparison of true labels with KMeans cluster assignments using the Adjusted Rand Index and Rand Index provides a measure of clustering quality. A higher value indicates better agreement between the true labels and the clusters identified by KMeans.

8. Data Splitting: Segregate 30% of the data for testing, using a random seed of 1. Use the remaining 70% for training and model selection.

```
np.random.seed(1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
```

9. Classifier Choices: Identify the two classifiers you have chosen and justify your selections.

```
# Classifier Choices
rf = RandomForestClassifier()
lr = LogisticRegression()

rf.fit(X_train, y_train)
lr.fit(X_train, y_train)

# Model Evaluation, Presenting Additional Info
rf_pred = rf.predict(X_test)
```

```
rf_y_prob = rf.predict_proba(X_test)
lr_pred = lr.predict(X_test)
lr_y_prob = lr.predict_proba(X_test)
probT_rf = pd.DataFrame(
    data = {'prob0': rf_y_prob[:,1], 'y_test': y_test}
    )
probT_lr = pd.DataFrame(
    data = {'prob0': lr_y_prob[:,1], 'y_test': y_test}
    )
probT_rf['y_test_pred'] = probT_rf.prob0.map(lambda x: 1 if x>0.5 else 0)
probT_lr['y_test_pred'] = probT_lr.prob0.map(lambda x: 1 if x>0.5 else 0)
```

We chose K-Nearest Neighbours (KNN) because it is a relativley simple classification algorithm, which we wanted to use as a baseline to compare to other, more sophisticated methods.

We chose logistic regression because it is relativley simple and easy to interpret, but more sophisticated and powerful than KNN. It was important that we chose an interpretable model so that we could understand what factors cause the model to make predictions, and ensure that they make sense.

Beyond what we learned in class we also chose to include a random forest classifier because it is known for not overfitting and being able to handle multi dimensional data. It also works well when there is both numerical and catagorical data which we have in this case. Random forests are similar to the decision trees we learned about in class, but they use bootstrapping and random feature selection to make predictions more robust. A random forest is a collection of decision trees which are generated using bootstrapping. Each decision tree is trained on a random subset of the data and a random subset of features. During prediction, the results from all trees are majority-voted for classification to determine the predicted class.

10. Performance Metrics: Outline the two metrics for comparing the performance of the classifiers.

```
# Accuracy, Sensitivity, Specificity
def evaluate(y_test, y_test_pred):
    cm = confusion_matrix(y_test,y_test_pred)
    print('Confusion Matrix : \n', cm)
    total = sum(sum(cm))
    accuracy = (cm[0,0]+cm[1,1])/total
    print ('Accuracy : ', accuracy)
    sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
    print('Sensitivity : ', sensitivity )
    specificity = cm[1,1]/(cm[1,0]+cm[1,1])
    print('Specificity : ', specificity)
    print(classification_report(y_test, y_test_pred, zero_division=0.0))
```

Accuracy, sensitivity, and specificity are essential metrics derived from the confusion matrix used to evaluate the performance of classification models.

Accuracy measures the overall correctness of the model's predictions by calculating the ratio of correctly predicted observations to the total observations in the dataset.

Sensitivity, also known as recall or true positive rate, assesses the model's ability to correctly identify positive cases, calculated as the proportion of true positives to the sum of true positives and false negatives.

Specificity evaluates the model's capacity to correctly identify negative cases, represented as the ratio of true negatives to the sum of true negatives and false positives.

These metrics provide valuable insights into different aspects of the classifiers's behavior, helping to compare classifiers.

11. Feature Selection/Extraction: Implement methods to enhance the performance of at least one classifier in (9). The answer for this question can be included in (12).

```
# Feature selection using Backward selection (called Recursive Feature Elimination (RFE) in sk
from sklearn.feature_selection import RFE

rfe_selector = RFE(lr, n_features_to_select=10, step=1)
```

```
rfe_selector.fit(X_train, y_train)
selected_features_rfe = X_train.columns[rfe_selector.support_]
```

```
print("Selected Features Using Backward Selection")
print(selected_features_rfe.to_list())
```

```
Selected Features Using Backward Selection
['bp', 'sg', 'al', 'su', 'rbc', 'bgr', 'sc', 'sod', 'hemo', 'pe']
```

We opted to use backward selection to select a subset of the most important features to retrain our models on.

Backward selection starts with all candidate features fits a logistic regression model to the full dataset. It then iteratively removes the least significant feature at each step. The process continues until a predefined stopping condition is met, in this case, that there are 10 features remaining (James et al. (2023), page 235).

12. Classifier Comparison: Utilize the selected metrics to compare the classifiers based on the test set. Discuss your findings (at least two statements).

Using all the features

```
# Classifier Choices and Instantiation
rf = RandomForestClassifier()
lr = LogisticRegression()

# Fit models to training data
rf.fit(X_train, y_train)
lr.fit(X_train, y_train)

# Model Evaluation
rf_pred = rf.predict(X_test)
rf_y_prob = rf.predict_proba(X_test)
lr_pred = lr.predict(X_test)
```

```python
lr_y_prob = lr.predict_proba(X_test)
probT_rf = pd.DataFrame(
    data = {'prob0': rf_y_prob[:,1], 'y_test': y_test}
    )
probT_lr = pd.DataFrame(
    data = {'prob0': lr_y_prob[:,1], 'y_test': y_test}
    )
probT_rf['y_test_pred'] = probT_rf.prob0.map(lambda x: 1 if x>0.5 else 0)
probT_lr['y_test_pred'] = probT_lr.prob0.map(lambda x: 1 if x>0.5 else 0)

print('Random Forest Classifier:\n')
evaluate(probT_rf.y_test, rf_pred)

print('Logistic Regression Classifier:\n')
evaluate(probT_lr.y_test, lr_pred)
```

```
Random Forest Classifier:

Confusion Matrix :
 [[70  0]
 [ 0 50]]
Accuracy :  1.0
Sensitivity :  1.0
Specificity :  1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        70
           1       1.00      1.00      1.00        50

    accuracy                           1.00       120
   macro avg       1.00      1.00      1.00       120
weighted avg       1.00      1.00      1.00       120
```

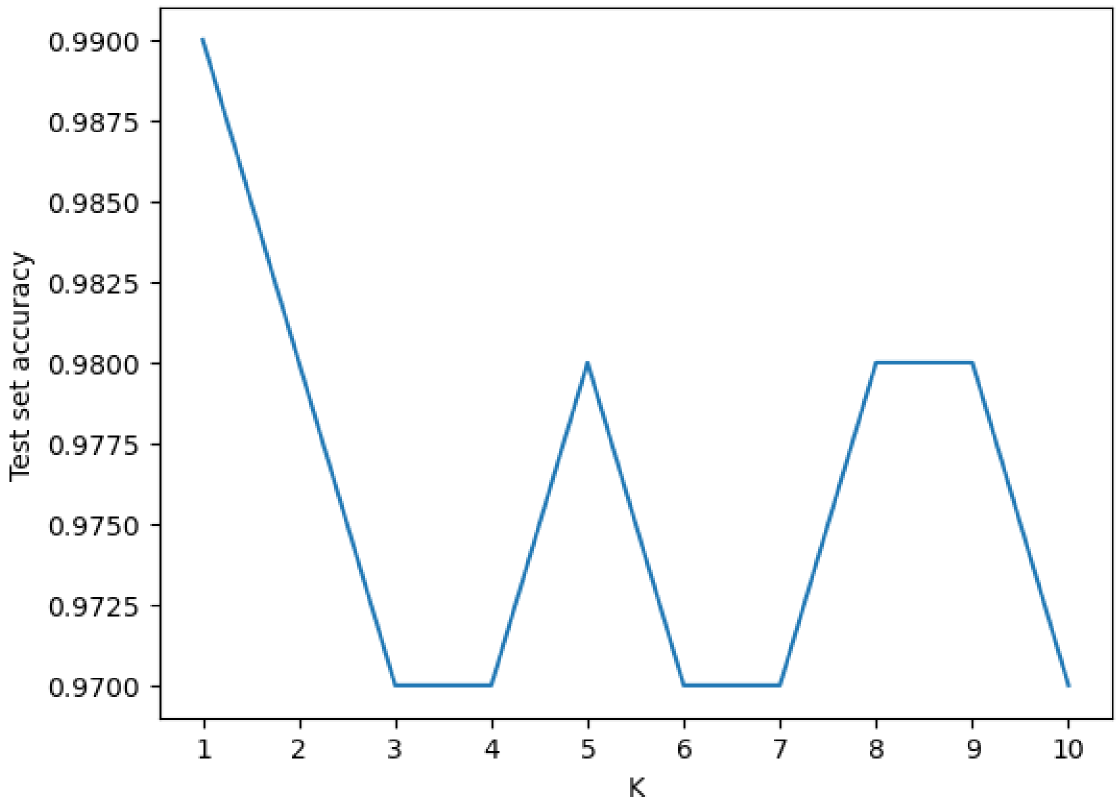
```
Logistic Regression Classifier:

Confusion Matrix :
 [[70  0]
 [ 0 50]]
Accuracy :  1.0
Sensitivity :  1.0
Specificity :  1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        70
           1       1.00      1.00      1.00        50

    accuracy                           1.00       120
   macro avg       1.00      1.00      1.00       120
weighted avg       1.00      1.00      1.00       120
```

Before performing KNN classification, we will first select the optimal value of K. This will be done using the technique learned in class: we will try a number of values, and select the one which yields best performance

```python
k_range = range(1, 11)
scores = []

for k in k_range:
    knn = 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))

print(f"Best value of k: {k_range[np.argmax(np.array(scores))]}")
```

```
Best value of k: 1
```

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



K=1 has the best performance, so we select it

```
# KNN:
# KNN Classifier
knn = KNeighborsClassifier(n_neighbors=1)

# Fit model to training data
knn.fit(X_train, y_train)
```

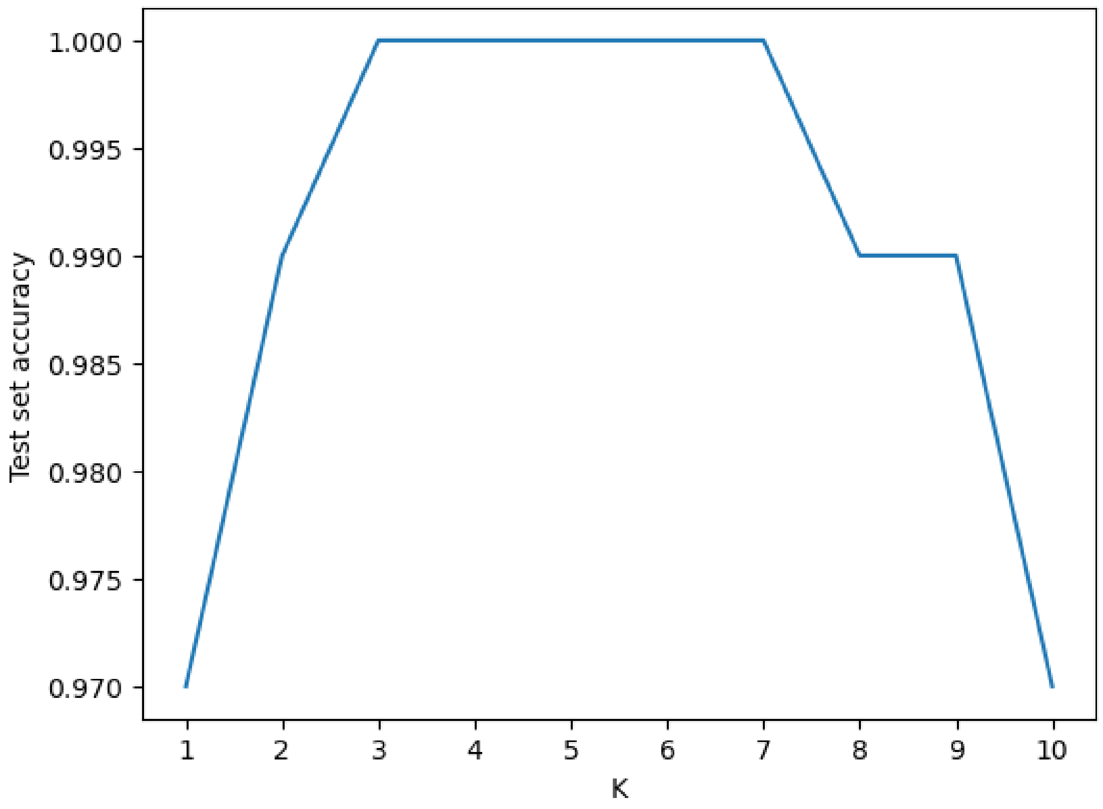
```
# Model Evaluation
knn_pred = knn.predict(X_test)
knn_y_prob = knn.predict_proba(X_test)

probT_knn = pd.DataFrame(
    data={'prob0': knn_y_prob[:, 1], 'y_test': y_test}
)

probT_knn['y_test_pred'] = probT_knn.prob0.map(lambda x: 1 if x > 0.5 else 0)

print('KNN Classifier:\n')
evaluate(probT_knn.y_test, knn_pred)
```

```
KNN Classifier:


Confusion Matrix :
 [[69  1]
 [ 0 50]]
Accuracy :  0.9916666666666667
Sensitivity :  0.9857142857142858
Specificity :  1.0
              precision    recall  f1-score   support


           0       1.00      0.99      0.99        70
           1       0.98      1.00      0.99        50


    accuracy                           0.99       120
   macro avg       0.99      0.99      0.99       120
weighted avg       0.99      0.99      0.99       120
```

Now using the selected features

```
k_range = range(1, 11)
scores = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train[selected_features_rfe], y_train)
    y_pred = knn.predict(X_test[selected_features_rfe])
    scores.append(round(metrics.accuracy_score(y_test, y_pred),2))

print(f"Best value of k: {k_range[np.argmax(np.array(scores))]}")

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

```
Best value of k: 3
```



This time, K=3 has the best performance, so we select it

```
# KNN:
# KNN2 Classifier
knn2 = KNeighborsClassifier(n_neighbors=3)

# Fit model to training data
knn2.fit(X_train[selected_features_rfe], y_train)

# Model Evaluation
knn2_pred = knn2.predict(X_test[selected_features_rfe])
knn2_y_prob = knn2.predict_proba(X_test[selected_features_rfe])

probT_knn2 = pd.DataFrame(
    data={'prob0': knn2_y_prob[:, 1], 'y_test': y_test}
)
```

```python
probT_knn2['y_test_pred'] = probT_knn2.prob0.map(lambda x: 1 if x > 0.5 else 0)

print('KNN2 Classifier:\n')
evaluate(probT_knn2.y_test, knn2_pred)
```

```
KNN2 Classifier:

Confusion Matrix :
 [[70  0]
 [ 0 50]]
Accuracy :  1.0
Sensitivity :  1.0
Specificity :  1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        70
           1       1.00      1.00      1.00        50

    accuracy                           1.00       120
   macro avg       1.00      1.00      1.00       120
weighted avg       1.00      1.00      1.00       120
```

```python
# Classifier Choices and Instantiation
rf2 = RandomForestClassifier()
lr2 = LogisticRegression()

# Fit models to training data
rf2.fit(X_train[selected_features_rfe], y_train)
lr2.fit(X_train[selected_features_rfe], y_train)

# Model Evaluation
```

```
rf_pred = rf2.predict(X_test[selected_features_rfe])
rf_y_prob = rf2.predict_proba(X_test[selected_features_rfe])
lr_pred = lr2.predict(X_test[selected_features_rfe])
lr_y_prob = lr2.predict_proba(X_test[selected_features_rfe])
probT_rf = pd.DataFrame(
    data={'prob0': rf_y_prob[:, 1], 'y_test': y_test}
)
probT_lr = pd.DataFrame(
    data={'prob0': lr_y_prob[:, 1], 'y_test': y_test}
)
probT_rf['y_test_pred'] = probT_rf.prob0.map(lambda x: 1 if x > 0.5 else 0)
probT_lr['y_test_pred'] = probT_lr.prob0.map(lambda x: 1 if x > 0.5 else 0)

print('Random Forest Classifier:\n')
evaluate(probT_rf.y_test, rf_pred)

print('Logistic Regression Classifier:\n')
evaluate(probT_lr.y_test, lr_pred)
```

```
Random Forest Classifier:


Confusion Matrix :
 [[70  0]
 [ 0 50]]
Accuracy :  1.0
Sensitivity :  1.0
Specificity :  1.0
              precision    recall  f1-score   support


           0       1.00      1.00      1.00        70
           1       1.00      1.00      1.00        50
```

```
    accuracy                           1.00       120
   macro avg       1.00      1.00      1.00       120
weighted avg       1.00      1.00      1.00       120

Logistic Regression Classifier:

Confusion Matrix :
 [[70  0]
 [ 0 50]]
Accuracy :  1.0
Sensitivity :  1.0
Specificity :  1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        70
           1       1.00      1.00      1.00        50

    accuracy                           1.00       120
   macro avg       1.00      1.00      1.00       120
weighted avg       1.00      1.00      1.00       120
```

We were happy to see very strong performance with all of the methods we tested. We achieved perfect classification on the test set using logistic regression and random forest classification when we used all of the features in the dataset. The KNN classifier had one misclassification, leading to the following metrics:

Accuracy: 0.99 Sensitivity: 0.99 Specificity: 1.00

We found that in this case, the optimal value of K is 1. When training on the full dataset, increasing the value beyond 1 decreases accuracy. There are some fluctuations in accuracy as K increases, but the trend remains downwards. When K equals 1, the target datapoint is classified based on its nearest neighbor. While K=1 may offer optimal performance, it increases the classifier's sensitivity to errors caused by outliers in the training set. This occurs because an outlier might be in close proximity to the target datapoint, despite the majority of nearby datapoints belonging to a different class.

When using only the subset of features obtained from backward selection, we found that the logistic regression and random forest classifiers maintained their optimal performance. This is a sign that the backward selection process worked well, as no valuable information was lost. Our KNN classifier also achived perfect performance on the test set once we trained it only on the features selcted using backward selection. Additionally, the optimal value of K changed under these features. We found an optimal value of k=3. This is promising in terms of generalization performance because a higher k value than 1 reduces the model's sensitivity to noise and outliers in the training data, enhancing its ability to generalize to unseen data.

13. Interpretable Classifier Insight: After re-training the interpretable classifier with all available data, analyze and interpret the significance of predictor variables in the context of the data and the challenge (at least two statements).

```
# retrain Logistic regression on whole dataset:
lr = LogisticRegression()
lr.fit(X[selected_features_rfe], y)

# Interpret Logistic Regression coefficients
coefficients = pd.DataFrame(lr.coef_, columns=X_train[selected_features_rfe].columns)
coefficients_transposed = coefficients.transpose()
coefficients_transposed.columns = ['Coefficient']
coefficients_transposed = coefficients_transposed.assign(Abs_Coefficient=abs(coefficients_trans
coefficients_transposed = coefficients_transposed.sort_values(by='Abs_Coefficient', ascending=F
coefficients_transposed = coefficients_transposed.drop('Abs_Coefficient', axis=1)
print("Interpretable Classifier Insight:")
print(coefficients_transposed.head(10))
```

```
Interpretable Classifier Insight:
      Coefficient
hemo     2.010680
```

```
al      -1.849457
rbc      1.796728
pe      -1.080013
sg       1.055752
sod      0.897158
sc      -0.852279
bgr     -0.674604
bp      -0.587153
su      -0.490523
```

For logistic regression, the value and the sign of the predictor variables are important in interpreting the results. In order to best interpret the results, we will first provide a brief overview of how to interpret them, and then point to a specific example and highlight what conclusions we may draw from our interpretation of it.

A positive coefficient for a given predictor variable implies a positive relationship between that variable and the outcome: when predictor variable increases so does the outcome. Similarly, negative implies a negative relationship. The particular value of a variable's coefficient indicates how strongly the predictor affects the outcome. A higher magnitude indicates that this variable implies a higher probility of either a positive or negative outcome (depending on the sign). Additionally, in logistic regression, we can use p-values to show whether each predictor varaible is statistically significant. The p-value in logistic regression comes from a hypothesis test, where the null hypothesis is that the true slope of the varible's coefficient is 0. This null hypothesis holds that changing the variable has no impact on the probability of a patient having kidney disease. The lower the p-value, the more likely the variable is to not be zero and be useful in predicting the outcome.

In this case, for example, the albumin predictor (al feature) has a coefficient -1.849457. This means that if patient albumin levels increases by one unit then the log-odds of the patient having CKD decreases (since sign is negative) by -1.849457.

From an interpretability perspective this highlights that our logistic regression model predicts that as a patient's levels of albumin decrease, the risk of the kidney disease becomes higher. Low Albumin is an established marker for kidney damage within the biomedial literature (Cheng et al.

(2023)). We take it as a promising sign that our model identified a pattern which is well-established through other means.

Similar interpretations can be drawn with respect to the hemoglobin (hemo) feature, which has a positive coefficient of 2.010680. For a patient, each increase in 1 unit of hemoglobin results in our logistic regression model predicting that the log-odds of the patient having CKD increases (since sign is positive) by 2.010680. Other research has shown that those with higher hemoglobin are at higher risk of CKD (Oh et al. (2012)).

Similar conclusions can be drawn for other features based on their sign and magnitude.

14. Sub-group Improvement Strategy: If sub-groups were identified, propose and implement a method to improve one classifier performance further. Compare the performance of the new classifer with the results in (12).

Contributions

Jenna: - Created/set up repository and jupyter notebook - started working on questions 1-4 - started working on 11, - made general edits - q11-13 with Noah

Viransh: - References added, - done questions 5-10

Noah: - Finished question 3, - added visualizations and discussion of normality/log-transformation, - added writing and detail for questions 1, 2 and 4 - q11-13 with Jenna - Proofread and edited all writing in final stages

Github Link

Github link (https://github.com/JennaOrvitz/Stats3DA3FinalProject/tree/main)

References

Cheng, Tong, Xiaoyu Wang, Yong Han, Jianbing Hao, Haofei Hu, and Lirong Hao. 2023. "The Level of Serum Albumin Is Associated with Renal Prognosis and Renal Function Decline in Patients with Chronic Kidney Disease." *BMC Nephrology* 24 (1): 57–57.

James, Gareth., Daniela. Witten, Trevor. Hastie, Robert. Tibshirani, and Jonathan. Taylor. 2023. *An Introduction to Statistical Learning with Applications in Python*. 1st ed. 2023. Springer Texts in Statistics. Cham: Springer International Publishing.

Oh, Se Won, Seon Ha Baek, Yong Chul Kim, Ho Suk Goo, Ho Jun Chin, Ki Young Na, Dong Wan Chae, and Suhnggwon Kim. 2012. "Higher Hemoglobin Level Is Associated with Subtle Declines in Renal Function and Presence of Cardiorenal Risk Factors in Early CKD Stages." *Nephrology, Dialysis, Transplantation* 27 (1): 267–75.

Rubini, Soundarapandian, L., and P. Eswaran. 2015. "Chronic Kidney Disease." UCI Machine Learning Repository.

Sanmarchi, Francesco, Claudio Fanconi, Davide Golinelli, Davide Gori, Tina Hernandez-Boussard, and Angelo Capodici. 2023. "Predict, Diagnose, and Treat Chronic Kidney Disease with Machine Learning: A Systematic Literature Review - Journal of Nephrology." *SpringerLink*. Springer International Publishing. https://link.springer.com/article/10.1007/s40620-023-01573-4.